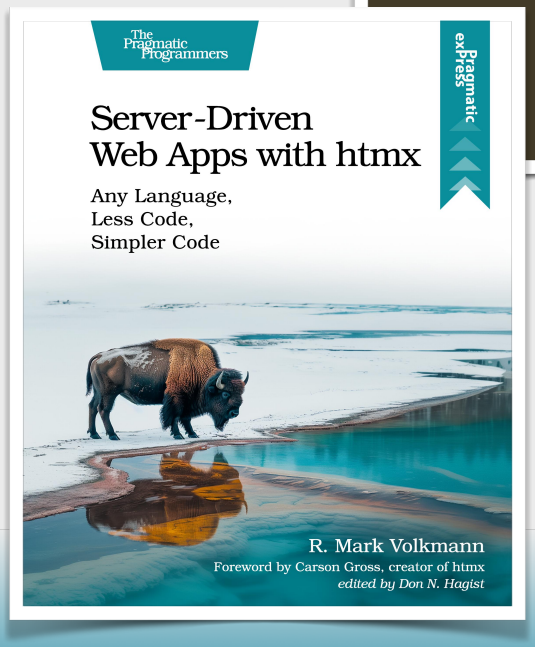
JavaScript Decorators

Slides at https://github.com/mvolkmann/talks/
Code at https://github.com/mvolkmann/web-component-book-code/ in ch04

R. Mark Volkmann
Object Computing, Inc.
https://objectcomputing.com
r.mark.volkmann@gmail.com

About Me

- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 44 years of professional software development experience
- Writer and speaker
- **Blog** at https://mvolkmann.github.io/blog/
- Author of Manning book "**Svelte ... in Action**"
- Author of Pragmatic Bookshelf book "**Server-Driven Web Apps with htmx**"
- Currently writing a book on web components



What is a Decorator

- A **design pattern** and **syntax** that allows you to wrap a piece of code to modify its behavior without changing its source code
- Applied to functions, classes, properties, and methods
- Supported by many programming languages including C#, Java, JavaScript, Kotlin, Python, Rust, Scala, and Swift

currently JavaScript decorators cannot be applied to functions

Why Use Decorators

- There's nothing a decorator can do that cannot be achieved in another way
- But decorators **reduce boilerplate code**
- They also **make code more declarative**, stating goals rather than how to achieve them

Why Avoid Decorators

- Decorators introduce a kind of **magic**
- Simply applying a decorator to a programming language construct, changes its functionality in a way that is more difficult to follow than walking through a sequence of function calls
- The code that implements a decorator you use
  may not be readily available and
  may not be well-documented

Why Learn How To Implement

- You are more likely to use decorators implemented by others than to implement your own
- However, learning how to implement decorators provides a valuable understanding of what is happening behind the scenes

Proposals

- **ECMAScript**: JavaScript programming language specification
  - maintained by TC39 committee within “ECMA International” standards organization
- “**Legacy decorators**”: described in stage 1 TC39 Decorators proposal
  - published in 2015
  - use of this style of decorators is now discouraged
- “**Standard decorators**”: described in stage 3 TC39 Decorators proposal
  - published in 2022
  - preferred over legacy decorators
- Decorators could become an official part of **ECMAScript 2026**

This talk focuses on standard decorators.

Where Used Today

- **TypeScript** 5+ compiler (tsc)
- **Vite**: build tool and development server
- **Deno**: JavaScript runtime; alternative to Node.js
- **Angular**: has used decorators since 2016
- **Web component libraries**: Lit, Stencil, and FAST

- Not yet in web browsers, Node.js or Bun
  - but code that uses decorators can be compiled to code that does not, and used in these environments

Actions

- A decorator can **access**, **initialize**, or **replace** its target
- **When applied to a class**,
  it can return a new class that extends the original
- **When applied to a method**,
  it can return a new method that optionally calls the original
- **When applied to a field**,
  it can return a new initial value, or a pair of getter and setter methods
- **To replace the target**, the decorator must
  return a value of the same type as the target
- If nothing is returned, the target is unchanged

Getters and setters are special methods that can be implemented in class to define a property.

Getter methods are automatically executed when the property they define is accessed.

Setter methods are automatically executed when the property they define is modified.

Applying and Processing

- **Applying**
  - a decorator is applied by adding the syntax `@decoratorName` before its target
  - no arguments can be passed
    - but arguments can be passed to a decorator factory, described later
  - any number of decorators can be applied to the same target
- **Processing**
  - decorators are processed when a class definition is parsed, not when each instance of the class is created
  - decorators that are applied to a class are processed after the decorators applied to its members are processed

A First Decorator ...

game.js

```
export class Game {
  #score = 0;

  get score() {
    return this.#score;
  }

  set score(value) {
    if (value < 0) {
      throw new Error(`score cannot be negative`);
    }
    this.#score = value;
  }
}
```

private field

getter method

setter method

can replace "`set score`" method with this

```
@nonNegative
set score(value) {
  this.#score = value;
}
```

This works fine, but imagine there are many classes with many numeric properties that want to enforce this same restriction. The code to do that would be required in many setter methods.
Decorators provide a way to express this more succinctly and remove boilerplate code by placing the code in one place, only in the decorator implementation.

```
import { Game } from "./game";

const game = new Game();
try {
  game.score = 7;
  game.score = -1;
} catch (e) {
  console.error((e as Error).message);
}
```

throws

... A First Decorator

- Implemented in TypeScript, so lots of types
- Uses a generic type to capture the type of object in which the decorator was applied
  - in our case this is **`Game`**
- Target must be a function that takes a **`number`** and returns nothing
  - expressed by **`target`** parameter type
- **`context`** parameter holds information about the context in which the decorator was applied
  - in our case, **`context.name`** will be set to a **`Symbol`** whose value is "**`score`**"
- Returns new setter method that replaces original and delegates to it

```
export function nonNegative<This>(
  target: (value: number) => void,
  context: ClassSetterDecoratorContext<This>
) {
  return function (this: This, newValue: number) {
    if (newValue < 0) {
      const name = String(context.name);
      throw new Error(`${name} cannot be negative`);
    }
    target.call(this, newValue);
  };
}
```

TypeScript will complain if this decorator is applied to anything other than a method that takes a number and returns nothing.

Auto-accessor Keyword

- Decorators proposal defines **`accessor`** keyword which can precede a class field declaration
- Removes need to manually write basic getter and setter methods that access a private field
- Primary motivation is to also apply a decorator see next slide
- These class definitions are equivalent

```
class Game {
  #score = 0;
  get score() {
    return this.#score;
  }
  set score(val) {
    this.#score = val;
  }
}
```

```
class Game {
  accessor score = 0;
}
```

Accessor Decorators

- Can log every get and set of the field
  - for example, see **`logAccess`** decorator later
- Can validate new field values
  - for example, see **`rangeValidation`** decorator later
- Can lazily compute the field value when its value is requested
- Can notify a UI library (such as Lit) to re-render part of the UI because the field value changed

Decorator Context

- Each decorator is implemented by a function that has two parameters, the target to which the decorator will be applied and a context object
- Target parameter value is always **`undefined`** for class fields
  - prevents decorators from accessing or modifying the field value
- Type of context object differs based on target type
- All context types have names that begin with **`Class`** and end with **`DecoratorContext`**

Context Types

CONTEXT MATTERS

Target Kind	Context TypeScript Type
class	`ClassDecoratorContext`
field	`ClassFieldDecoratorContext`
method	`ClassMethodDecoratorContext`
getter	`ClassGetterDecoratorContext`
setter	`ClassSetterDecoratorContext`
accessor	`ClassAccessorDecoratorContext`

Context `kind` & `name` Properties

- String properties present in all context objects
- **`kind`** is a string whose value is one from the first column in table
  - can be used to verify that a decorator is applied to the expected kind of target
- **`name`** holds the name of the target as a **`Symbol`**
- For example, when a decorator is applied to a class
  - context **`kind`** property is set to "**`class`**"
  - context **`name`** property is set to the class name

Context `static` & `private` Properties

- Boolean properties present in all context objects except those of type **`ClassDecoratorContext`**
- **`static`** property can be used to determine if the decorator is being applied at the class or instance level
- **`private`** property can be used to determine if the decorator has permission to access the member

Context `addInitializer` Property

- Present in all context objects
- Takes a function as its argument that is called **after** the target to which the decorator is applied is fully defined
- Important in situations where calling the function before the target is fully defined would be inappropriate
- Can be called any number of times on the same context object
- Functions passed are called in the same order in which they are added, each time the decorator is applied
- Used in **`customElement`** decorator shown later

Context `metadata` Property

- Present in all context objects
- Enables multiple decorators to store data
  in an object that is associated with the class
- The data can be accessed later for many purposes
- Much more on this later!

Context `access` Property

- Present in all context objects
  except those of type **`ClassDecoratorContext`**
- Value is an object that can have the methods **`has`**, **`get`**, and **`set`**
  - **`has`** method tests whether a given object has a member with the same name
  - **`get`** method gets the same member value from a given object
  - **`set`** method sets the same field in a given object to a specified value
- Seems difficult to find good used cases for this property

`logAccess` Decorator

- Can be applied to an accessor field
- Logs all target field accesses, including getting and setting value
- Returns an object with **`get`** and **`set`** methods that replace those generated by **`accessor`** keyword

```
export function logAccess<This, Value>(
  target: ClassAccessorDecoratorTarget<This, Value>,
  { kind, name }: ClassAccessorDecoratorContext<This, Value>
) {
  if (kind !== "accessor") {
    throw new Error(
      "This decorator can only be applied to " +
        'a field with the "accessor" keyword.'
    );
  }
  const nameString = String(name); // name is a Symbol
  return {
    get(this: This) {
      const value = target.get.call(this);
      console.log(
        `Getting ${nameString} field value ${value}.`
      );
      return value;
    },
    set(this: This, value: Value) {
      console.log(
        `Setting ${nameString} field to ${value}.`
      );
      target.set.call(this, value);
    },
  };
}
```

`get` and **`set`** methods use "fake this" specifies type of **`this`** inside the function

Using `logAccess`

```
class Dog {
  name = "";

  @logAccess
  accessor age = 0;

  constructor(name: string) {
    this.name = name;
  }
}

const dog = new Dog('Comet');
dog.age = 5;
console.log(dog.age);
```

outputs “Setting age field to 5.”

outputs “Geting age field value 5.”

`timeMethod` Decorator

- Measures and outputs time required for each execution of a target method
- Returns new method that replaces decorated method and delegates to it
- Call to **`console.time`** starts a timer
- Call to **`console.timeEnd`** stops timer and outputs elapsed time

```
export function timeMethod<This, Return>(
  originalMethod: (...args: any[]) => Return,
  { kind, name }: ClassMethodDecoratorContext<This>
) {
  if (kind !== "method") {
    throw new Error(
      "This decorator can only be applied to a method."
    );
  }
  const nameString = String(name);
  return function (this: This, ...args: any[]): Return {
    console.time(nameString);
    const result = originalMethod.call(this, ...args);
    console.timeEnd(nameString);
    return result;
  };
}
```

Using `timeMethod`

- Applies decorator to static method **`fibonacci`** which computes the nth value in Fibonacci sequence
- **`fibonacci`** method calls function **`fib`**, which is recursive
- Recall that decorators cannot be applied to functions
- We wouldn't want to apply the decorator to the **`fib`** function anyway because we don't want to time each recursive call
- We only want to output the total time required to compute final result

```
function fib(n: number): number {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}

class MathLab {
  @timeMethod
  static fibonacci(n: number): number {
    return fib(n);
  }
}

console.log(
  "fibonacci(20) =",
  MathLab.fibonacci(20)
);
```

outputs “fibonacci: 0.171ms” or similar time, and “fibonacci(20) = 6765”

`countInstances` Decorator

- Counts instances created from target class
- Returns a class that extends the target class by adding the static field **`_instanceCount`** and static getter **`instanceCount`**
- Every time a new instances is created, the count is incremented
- TypeScript type for “any class” is **`new (+++...+++args: any[]) => {}`**
  - used to specify that the generic type **`Value`** is a class that has a constructor which takes any number of arguments with any types

```
export function countInstances<
  Value extends new (...args: any[]) => {}
>(
  target: Value,
  { kind }: ClassDecoratorContext<T>
) {
  if (kind !== "class") {
    throw new Error(
      "This decorator can only " +
      "be applied to a class."
    );
  }
  return class extends target {
    private static _instanceCount = 0;

    constructor(...args: any[]) {
      super(...args);
      (this.constructor as any).
        _instanceCount++;
    }

    static get instanceCount() {
      return this._instanceCount;
    }
  };
}
```

Using `countInstances`

- Applies decorator to class **`Dog`**

```
@countInstances
export class Dog {
  name = "";
  constructor(name: string) {
    this.name = name;
  }
}

const dogs = [
  new Dog("Ramsay"),
  new Dog("Oscar"),
  new Dog("Comet"),
  new Dog("Greta")
];
console.log((Dog as any).instanceCount);
```

outputs “4”

Decorator Factories

- A function that returns a decorator function
- Enables passing arguments that are used to configure a decorator
- Applied with the syntax `@decoratorFactoryName(arguments)`

`rangeValidation` Decorator Factory

- Adds range validation to a number field
- Defines **`validate`** function that is scoped to decorator function
  - throws an error if the value passed is outside the range described by **`min`** and **`max`** parameters
- The decorator returns an object with **`init`** and **`set`** methods
  - both call **`validate`** function

```
export function rangeValidation(
  min: number, max: number
) {
  return function <This, Value extends number>(
    target: ClassAccessorDecoratorTarget<This, Value>,
    context: ClassAccessorDecoratorContext<This, Value>
  ): ClassAccessorDecoratorResult<This, Value> {
    function validate(value: Value) {
      if (value < min || value > max) {
        const name = String(context.name);
        throw new Error(
          `${name} ${value} is outside ` +
          `range ${min} to ${max}`
        );
      }
    }

    return {
      init(initialValue: Value): Value {
        validate(initialValue);
        return initialValue;
      },
      set(this: This, newValue: Value) {
        validate(newValue);
        target.set.call(this, newValue);
      },
    };
  };
```

calls generated setter method

Using `rangeValidation`

- Applies decorator factory to field **`age`** in class **`Dog`**

```
class Dog {
  name = "";

  @rangeValidation(0, 20)
  accessor age = 0;

  constructor(name: string) {
    this.name = name;
  }
}

const comet = new Dog("Comet");
try {
  comet.age = 5;
  comet.age = 50;
} catch (e) {
  console.error((e as Error).message);
}
```

allowed

throws

outputs “Uncaught Error: age 50 is outside range 0 to 20”

`customElement` Decorator Factory

- Registers a web component class with a custom element name
- Does not return a new class definition
- A similar decorator factory is provided by the Lit library
  - but it's implementation is more complicated because it supports both legacy and standard decorators

```
export function customElement(name: string) {
  return (
    target: CustomElementConstructor,
    context: ClassDecoratorContext
  ) => {
    if (context.kind !== "class") {
      throw new Error(
        "This decorator can only " +
        "be applied to a class."
      );
    }
    if (!(target.prototype instanceof HTMLElement)) {
      throw new Error(
        "This decorator can only be applied " +
          "to a class that extends HTMLElement."
      );
    }
    context.addInitializer(() => {
      if (!customElements.get(name)) {
        customElements.define(name, target);
      }
    });
  };
}
```

avoids registering a custom element name that is already registered

`addInitializer` takes a function as its argument that is called **after** the class to which the decorator is applied is fully defined

Using `customElement`

- Applies decorator factory to field **`age`** in class **`Dog`**

```
class Dog {
  name = "";

  @rangeValidation(0, 20)
  accessor age = 0;

  constructor(name: string) {
    this.name = name;
  }
}

const comet = new Dog("Comet");
try {
  comet.age = 5;
  comet.age = 50;
} catch (e) {
  console.error((e as Error).message);
}
```

allowed

throws

outputs “Uncaught Error: age 50 is outside range 0 to 20”

Using `CustomElement`

- Applies decorator factory to class **`HelloWorld`**

hello-world.js

```
@customElement("hello-world")
export class HelloWorld extends HTMLElement {
  connectedCallback() {
    this.innerHTML = "<p>Hello, World!</p>";
  }
}
```

index.html

```
<!doctype html>
<html lang="en">
  <head>
    <script src="hello-world.js" type="module"></script>
  </head>
  <body>
    <hello-world></hello-world>
  </body>
</html>
```

Applying Multiple Decorators

- Multiple decorators can be applied to the same target
- Each can be specified on its own line or they can be specified on the same line
- When decorator factories are used, those are called in the order in which they are applied, top to bottom

```
@decoratorA
@decoratorB
class SomeClass {
  ...
}
```

```
@decoratorA @decoratorB class SomeClass {
  ...
}
```

- Once all the factories have returned a decorator function, all the decorator functions are called in reverse order, bottom to top
  - allows a decorator function to be applied to the result of a decorator function that runs before it
- Decorators that are applied after one that does not return a value are applied to the previous target value

Using Context Metadata

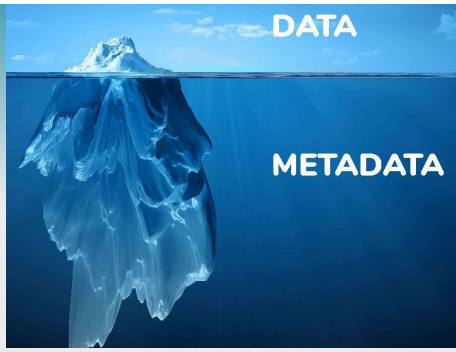
- When any decorator is applied to a class or its members, a **`metadata`** property is added to the class
  - its value is a plain, empty JavaScript object
- The context object passed to every decorator contains a **`metadata`** property whose value is the same object for all decorators applied to the same class or class member
  - allows multiple decorators to add data to a common metadata object
- After a class is fully initialized, its metadata object can be accessed using the key **`Symbol.metadata`**
- In classes where no decorators were applied, the value of that class property will be **`undefined`**

Polyfill

- **`Symbol.metadata`** is a recent addition to JavaScript which may not exist in environments such as web browsers and Node.js
- The following code is a polyfill that ensures that it is defined
- This *MUST* be executed *before* the definitions of any classes that use decorators are encountered

```
(Symbol as any).metadata ??= Symbol("Symbol.metadata");
```

Metadata Use Case



- One scenario is validating the fields of an object on demand rather than when they are modified
  - like in **`rangeValidation`** decorator factory presented earlier
- Decorators can be applied to class fields to specify their validation constraints
- Examples include required fields, numeric fields whose value must be in a given range, string fields with minimum lengths, and string fields that must match a regular expression
- Login dialogs typically require these kinds of validations
  - username and password are required
  - password may require at least eight characters and may need to match a regular expression which requires it to contain specific characters

Describing Validations

- Each validation decorator adds an object containing a function and an error message to the class metadata
- These functions accept any value and return a Boolean indicating whether the value is valid
- TypeScript type for these objects

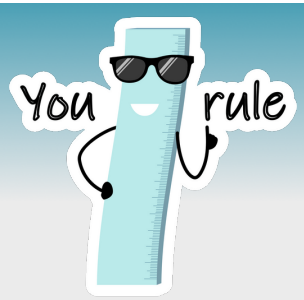
```
type ValidationRule = {
  validate: (value: any) => boolean;
  message: string;
};
```

Restricting Application

- The validation decorators can only be applied to fields and accessors
- This function throws an error when a decorator is applied to anything else

```
function fieldOrAccessor({ kind }: DecoratorContext) {
  if (kind !== "field" && kind !== "accessor") {
    throw new Error(
      "This decorator can only be applied to a class field or accessor."
    );
  }
}
```

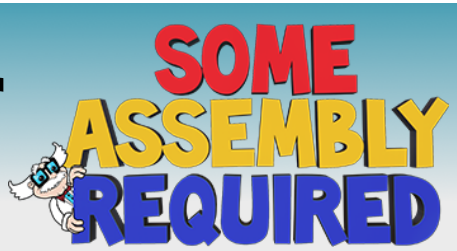
Adding a Validation Rule



- Code required to add a **`ValidationRule`** object to the metadata for a class is non-trivial
- This function handles that and is used by each of the validation decorators that follow

```
function addValidationRule(context: DecoratorContext, rule: ValidationRule) {
  const { metadata } = context;
  let constraints = metadata["constraints"] as Record<string, ValidationRule[]>;
  if (!constraints) constraints = metadata.constraints = {};
  const name = String(context.name);
  constraints[name] ??= []; <3>
  constraints[name].push(rule);
}
```

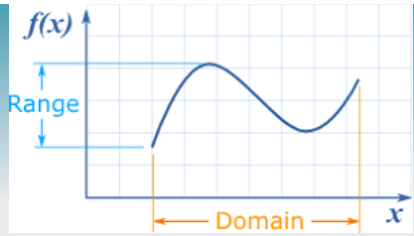
`required` Decorator



- Specifies that a field must have a value other than **`undefined`**, **`null`**, or an empty string

```
export function required(target: unknown, context: DecoratorContext) {
  fieldOrAccessor(context);
  addValidationRule(context, {
    validate: (v: unknown) => v !== undefined && v !== null && v !== "",
    message: `${String(context.name)} is required`,
  });
}
```

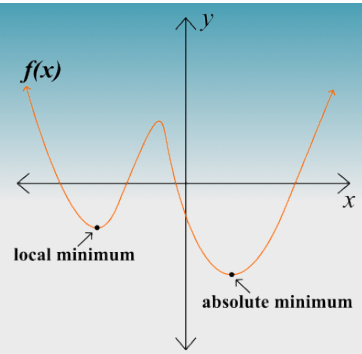
`range` Decorator



- Specifies that a numeric field must have a value in given, inclusive range

```
export function range(min: number, max: number) {
  return (target: unknown, context: DecoratorContext) => {
    fieldOrAccessor(context);
    addValidationRule(context, {
      validate: (v: number) => min <= v && v <= max,
      message: `${String(context.name)} must be between ${min} and ${max}`,
    });
  };
}
```

`minLength` Decorator



- Specifies that a string field must have at least a given length

```
export function minLength(len: number) {
  return (target: unknown, context: DecoratorContext) => {
    fieldOrAccessor(context);
    addValidationRule(context, {
      validate: (v: string) => v.length >= len,
      message: `${String(context.name)} must be at least ${len} characters`,
    });
  };
}
```

`regex` Decorator

- Specifies that a string field must match a given regular expression

```
export function regex(pattern: string) {
  return (target: unknown, context: DecoratorContext) => {
    fieldOrAccessor(context);
    addValidationRule(context, {
      validate: (v: string) => new RegExp(pattern).test(v),
      message: `${String(context.name)} must match pattern ${pattern}`,
    });
  };
}
```

Applying Validation Decorators

- Each of the validation decorators we've defined are applied to members of this **`Residence`** class

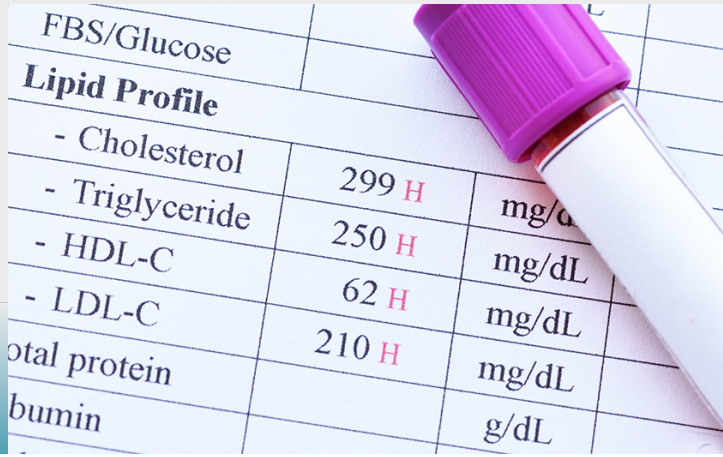
```
export class Residence {
  @required
  @minLength(3)
  accessor city = "";

  @regex("^[0-9]{5}$")
  accessor zip = "";

  @range(0, 100)
  accessor years = 0;
}
```

Getting Validation Results ...

- **`validate`** function on next slide takes any object
- If the constructor of the object has an associated **`metadata`** object that has a **`constraints`** property,
  those validation rules are evaluated
- Returns an object with **`valid`** and **`errors`** properties
  - **`valid`** is a Boolean that indicates whether any errors were found
  - **`errors`** is an array of error message strings



... Getting Validation Results

```
export function validate(instance: Record<string, any>) {
  const metadata = instance.constructor[Symbol.metadata] ?? {};
  const constraints = metadata["constraints"] ?? {};
  const errors: string[] = [];
  for (const [prop, rules] of Object.entries(constraints)) {
    const value = instance[prop];
    for (const rule of rules) {
      if (!rule.validate(value)) {
        errors.push(`${rule.message} (value is ${JSON.stringify(value)})`);
      }
    }
  }
  return {
    valid: errors.length === 0,
    errors,
  };
}
```

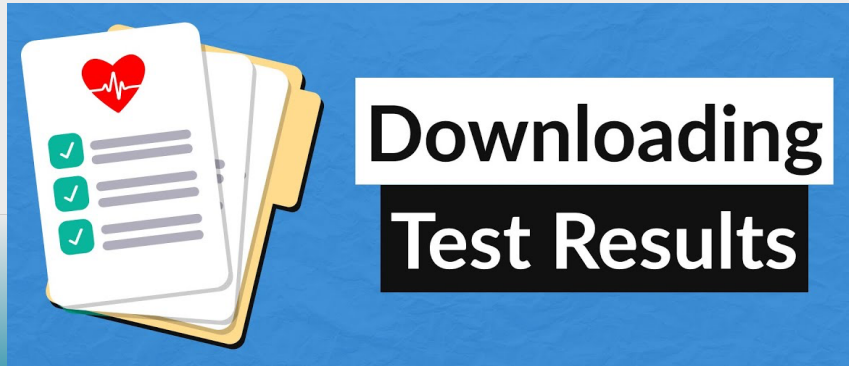
Performing Validation

```
const residence = new Residence();
residence.years = -3;
console.error(validate(residence));

residence.city = "St. Charles";
residence.zip = "63304";
residence.years = 27;
console.error(validate(residence));
```

```
{
  valid: false,
  errors: [
    'city must be at least 3 characters (value is "")',
    'city is required (value is "")',
    'zip must match pattern ^[0-9]{5}$ (value is "")',
    'years must be between 0 and 100 (value is -3)'
  ]
}
```

```
{ valid: true, errors: [] }
```



Using with TypeScript

- The **`tsc`** command compiles **`.ts`** files that can use standard decorators to **`.js`** files that do not
- To enable this, create the files below
- Enter **`npm run dev`** to start a local HTTP server and browse **`localhost:5173`**

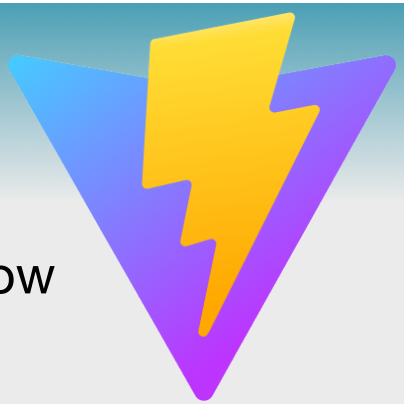
package.json

```
{
  "name": "some-name",
  "type": "module",
  "scripts": {
    "build": "tsc",
    "clean": "rimraf dist",
    "start": "node dist/demo.js"
  },
  "devDependencies": {
    "rimraf": "^6.1.2",
    "typescript": "^5.9.3"
  }
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "lib": ["ES2022", "DOM"],
    "outDir": "./dist",
    "skipLibCheck": true,
    "strict": true,
    "target": "ES2022"
  }
}
```

Using with Vite



- To use standard decorators in a Vite project, create the files below
- Enter **`npm run dev`** to start a local HTTP server and browse **`localhost:5173`**

package.json

```
{
  "name": "some-name",
  "type": "module",
  "scripts": {
    "dev": "vite"
  },
  "type": "module",
  "devDependencies": {
    "typescript": "^5.9.3",
    "vite": "^7.3.0"
  }
}
```

tsconfig.json

```
{
  "compilerOptions": {
    "target": "ESNext",
    "strict": true
  }
}
```

vite.config.ts

```
import { defineConfig } from "vite";

export default defineConfig({
  esbuild: {
    target: "es2022",
    include: /\.[jt]s$/,
    loader: "ts",
  },
});
```

Resources

- Decorators proposal
  - https://github.com/tc39/proposal-decorators
- Decorators in TypeScript 5.0
  - https://devblogs.microsoft.com/typescript/announcing-typescript-5-0/#decorators
- ECMAScript specification
  - https://ecma-international.org/publications-and-standards/standards/ecma-262/

Wrap Up

- Decorators reduce boilerplate code
  and make code more declarative
- Arguments cannot be specified when applying a decorator,
  but they can be specified when applying a decorator factory,
  which returns a configured decorator
- Decorators will be an official JavaScript feature spon,
  but they can be used today with TypeScript and Vite

My Latest Effort

- See npm package **wrec** which greatly simplifies creating web components
  - https://www.npmjs.com/package/wrec
- Name is acronym for **W**eb **RE**active **C**omponents

```
import {html, Wrec} from 'wrec';

class HelloWorld extends Wrec {
  static properties = {
    name: {type: String, value: 'World'}
  };

  static html = html`<div>Hello, <span>this.name</span>!</div>`;
}

HelloWorld.register();
```

```
<hello-world name="Mark"></hello-world>
```