Introduction

This book is about `wasm-bindgen`, a Rust library and CLI tool that facilitate high-level interactions between Wasm modules and JavaScript. The `wasm-bindgen` tool and crate are only one part of the Rust and WebAssembly ecosystem. If you're not familiar already with `wasm-bindgen` it's recommended to start by reading the Game of Life tutorial. If you're curious about `wasm-pack`, you can find that documentation here.

The `wasm-bindgen` tool is sort of half polyfill for features like the component model proposal and half features for empowering high-level interactions between JS and wasm-compiled code (currently mostly from Rust). More specifically this project allows JS/wasm to communicate with strings, JS objects, classes, etc, as opposed to purely integers and floats. Using `wasm-bindgen` for example you can define a JS class in Rust or take a string from JS or return one. The functionality is growing as well!

Currently this tool is Rust-focused but the underlying foundation is language-independent, and it's hoping that over time as this tool stabilizes that it can be used for languages like C/C++!

Notable features of this project includes:

- Importing JS functionality in to Rust such as DOM manipulation, console logging, or performance monitoring.
- Exporting Rust functionality to JS such as classes, functions, etc.
- Working with rich types like strings, numbers, classes, closures, and objects rather than simply `u32` and floats.
- Automatically generating TypeScript bindings for Rust code being consumed by JS.

With the addition of `wasm-pack` you can run the gamut from running Rust on the web locally, publishing it as part of a larger application, or even publishing Rust-compiled-to-WebAssembly on NPM!

Examples of using `wasm-bindgen`, `js-sys`, and `web-sys`

This subsection contains examples of using the `wasm-bindgen`, `js-sys`, and `web-sys` crates. Each example should have more information about what it's doing.

These examples all assume familiarity with `wasm-bindgen`, `wasm-pack`, and building a Rust and WebAssembly project. If you're unfamiliar with these check out the Game of Life tutorial or wasm pack tutorials to help you get started.

The source code for all examples can also be found online to download and run locally. For best results, consider checking out the latest release before testing them out. Most examples are configured with Webpack/`wasm-pack` and can be built with `pnpm run serve`. Other examples which don't use Webpack are accompanied with instructions or a `build.sh` showing how to build it.

Note that most examples currently use Webpack to assemble the final output artifact, but this is not required! You can review the deployment documentation for other options of how to deploy Rust and WebAssembly.

Hello, World!

View full source code or view the compiled example online

This is the "Hello, world!" example of `#[wasm_bindgen]` showing how to set up a project, export a function to JS, call it from JS, and then call the `alert` function in Rust.

`Cargo.toml`

The `Cargo.toml` lists the `wasm-bindgen` crate as a dependency.

Also of note is the `crate-type = ["cdylib"]` which is largely used for wasm final artifacts today.

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "hello_world"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = { path = "../../" }

[lints]
workspace = true
```

src/lib.rs

Here we define our Rust entry point along with calling the `alert` function.

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    fn alert(s: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    alert(&format!("Hello, {name}!"));
}
```

index.js

Our JS entry point is quite small!

```
import { greet } from './pkg';

greet('World');
```

Webpack-specific files

Note: Webpack is required for this example, and if you're interested in options that don't use a JS bundler see other examples.

And finally here's the Webpack configuration and `package.json` for this project:

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const webpack = require('webpack');
const WasmPackPlugin = require("@wasm-tool/wasm-pack-plugin");

module.exports = {
    entry: './index.js',
    output: {
                path:  path.resolve(__dirname,  '..',  'dist',
'hello_world'),
        filename: 'index.js',
    },
    plugins: [
        new HtmlWebpackPlugin(),
        new WasmPackPlugin({
            crateDirectory: __dirname
        }),
    ],
    mode: 'development',
    experiments: {
        asyncWebAssembly: true
   }
};
```

package.json

```
{
  "scripts": {
```

```
    "build": "webpack",
    "serve": "webpack serve"
  },
  "devDependencies": {
    "@wasm-tool/wasm-pack-plugin": "catalog:",
    "html-webpack-plugin": "catalog:",
    "webpack": "catalog:",
    "webpack-cli": "catalog:",
    "webpack-dev-server": "catalog:"
  }
}
```

`console.log`

View full source code or view the compiled example online

This example shows off how to use `console.log` in a variety of ways, all the way from bare-bones usage to a `println!`-like macro with `web_sys`.

src/lib.rs

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen(start)]
fn run() {
    bare_bones();
    using_a_macro();
    using_web_sys();
}

// First up let's take a look of binding `console.log` manually, without the
// help of `web_sys`. Here we're writing the `#[wasm_bindgen]` annotations
// manually ourselves, and the correctness of our program relies on the
// correctness of these annotations!

#[wasm_bindgen]
extern "C" {
      // Use `js_namespace` here to bind `console.log(..)` instead of just
    // `log(..)`
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);

    // The `console.log` is quite polymorphic, so we can bind it with multiple
      // signatures. Note that we need to use `js_name` to ensure we always call
    // `log` in JS.
    #[wasm_bindgen(js_namespace = console, js_name = log)]
    fn log_u32(a: u32);
```

```
    // Multiple arguments too!
    #[wasm_bindgen(js_namespace = console, js_name = log)]
    fn log_many(a: &str, b: &str);
}

fn bare_bones() {
    log("Hello from Rust!");
    log_u32(42);
    log_many("Logging", "many values!");
}

// Next let's define a macro that's like `println!`, only it
works for
// `console.log`. Note that `println!` doesn't actually work
on the Wasm target
// because the standard library currently just eats all
output. To get
// `println!`-like behavior in your app you'll likely want a
macro like this.

macro_rules! console_log {
     // Note that this is using the `log` function imported
above during
    // `bare_bones`
    ($($t:tt)*) => (log(&format_args!($($t)*).to_string()))
}

fn using_a_macro() {
    console_log!("Hello {}!", "world");
    console_log!("Let's print some numbers...");
    console_log!("1 + 3 = {}", 1 + 3);
}

// And finally, we don't even have to define the `log`
function ourselves! The
// `web_sys` crate already has it defined for us.
```

```
fn using_web_sys() {
    use web_sys::console;

    console::log_1(&"Hello using web-sys".into());

    let js: JsValue = 4.into();
          console::log_2(&"Logging   arbitrary   values   looks
like".into(), &js);
}
```

Small Wasm files

View full source code or view the compiled example online

One of `wasm-bindgen`'s core goals is a pay-only-for-what-you-use philosophy, so if we don't use much then we shouldn't be paying much! As a result `#[wasm_bindgen]` can generate super-small executables

Currently this code...

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn add(a: u32, b: u32) -> u32 {
    a + b
}
```

generates a 710 byte Wasm binary:

```
$ ls -l add_bg.wasm
-rw-rw-r-- 1 alex alex 710 Sep 19 17:32 add_bg.wasm
```

If you run wasm-opt, a C++ tool for optimize WebAssembly, you can make it even smaller too!

```
$ wasm-opt -Os add_bg.wasm -o add.wasm
$ ls -l add.wasm
-rw-rw-r-- 1 alex alex 172 Sep 19 17:33 add.wasm
```

And sure enough, using the wasm2wat tool it's quite small!

```
$ wasm2wat add.wasm
(module
  (type (;0;) (func (param i32 i32) (result i32)))
  (func (;0;) (type 0) (param i32 i32) (result i32)
    get_local 1
    get_local 0
    i32.add)
  (table (;0;) 1 1 anyfunc)
  (memory (;0;) 17)
  (global (;0;) i32 (i32.const 1049118))
```

```
  (global (;1;) i32 (i32.const 1049118))
  (export "memory" (memory 0))
  (export "__indirect_function_table" (table 0))
  (export "__heap_base" (global 0))
  (export "__data_end" (global 1))
  (export "add" (func 0))
  (data (i32.const 1049096) "invalid malloc request"))
```

Also don't forget to compile in release mode for the smallest binaries! For larger applications you'll likely also want to turn on LTO to generate the smallest binaries:

```
[profile.release]
lto = true
```

Without a Bundler

View full source code

This example shows how the `--target web` flag can be used load code in a browser directly. For this deployment strategy bundlers like Webpack are not required. For more information on deployment see the dedicated documentation.

First, you'll need to add `web-sys` to your Cargo.toml.

```
[dependencies.web-sys]
version = "0.3.4"
features = [
  'Document',
  'Element',
  'HtmlElement',
  'Node',
  'Window',
]
```

Then, let's take a look at the code and see how when we're using `--target web` we're not actually losing any functionality!

```
use wasm_bindgen::prelude::*;

// Called when the Wasm module is instantiated
#[wasm_bindgen(start)]
fn main() -> Result<(), JsValue> {
      // Use `web_sys`'s global `window` function to get a
handle on the global
    // window object.
     let window = web_sys::window().expect("no global `window`
exists");
      let document = window.document().expect("should have a
document on window");
     let body = document.body().expect("document should have a
body");
```

```
    // Manufacture the element we're gonna append
    let val = document.create_element("p")?;
    val.set_inner_html("Hello from Rust!");

    body.append_child(&val)?;

    Ok(())
}

#[wasm_bindgen]
pub fn add(a: u32, b: u32) -> u32 {
    a + b
}
```

Otherwise the rest of the deployment magic happens in `index.html`:

```
<html>
  <head>
            <meta   content="text/html;charset=utf-8"   http-
equiv="Content-Type"/>
  </head>
  <body>
    <!-- Note the usage of `type=module` here as this is an
ES6 module -->
    <script type="module">
      // Use ES module import syntax to import functionality
from the module
      // that we have compiled.
      //
      // Note that the `default` import is an initialization
function which
        // will "boot" the module and make it ready to use.
Currently browsers
      // don't support natively imported WebAssembly as an ES
module, but
          //  eventually  the  manual  initialization  won't  be
```

```
required!
      import init, { add } from './without_a_bundler.js';

      async function run() {
        // First up we need to actually load the Wasm file, so
we use the
        // default export to inform it where the Wasm file is
located on the
        // server, and then we wait on the returned promise to
wait for the
        // Wasm to be loaded.
        //
                    //  It  may  look  like  this:  `await
init('./without_a_bundler_bg.wasm');`,
          // but there is also a handy default inside `init`
function, which uses
        // `import.meta` to locate the Wasm file relatively to
js file.
        //
        // Note that instead of a string you can also pass in
any of the
        // following things:
        //
        // * `WebAssembly.Module`
        //
        // * `ArrayBuffer`
        //
        // * `Response`
        //
          // * `Promise` which returns any of the above, e.g.
`fetch("./path/to/wasm")`
        //
        // This gives you complete control over how the module
is loaded
        // and compiled.
        //
```

```
         // Also note that the promise, when resolved, yields
the Wasm module's
         // exports which is the same as importing the `*_bg`
module in other
        // modes
        await init();

          // And afterwards we can use all the functionality
defined in wasm.
        const result = add(1, 2);
        console.log(`1 + 2 = ${result}`);
        if (result !== 3)
          throw new Error("wasm addition doesn't work!");
      }

      run();
    </script>
  </body>
</html>
```

Note: You cannot directly open `index.html` in your web browser due to CORS limitations. Instead, you can set up a quick development environment using Python's built-in HTTP server:

```
wasm-pack build --target web
python3 -m http.server 8080
```

If you don't have Python installed, you can also use miniserve which is installable via Cargo:

```
cargo install miniserve
miniserve . --index "index.html" -p 8080
```

And that's it! Be sure to read up on the deployment options to see what it means to deploy without a bundler.

Using the older `--target no-modules`

View full source code

The older version of using `wasm-bindgen` without a bundler is to use the `--target no-modules` flag to the `wasm-bindgen` CLI.

While similar to the newer `--target web`, the `--target no-modules` flag has a few caveats:

- It does not support local JS snippets
- It does not generate an ES module
- It does not support `--split-linked-modules` outside of a document, e.g. inside a worker

With that in mind the main difference is how the wasm/JS code is loaded, and here's an example of loading the output of `wasm-pack` for the same module as above.

```
<html>
  <head>
            <meta   content="text/html;charset=utf-8"   http-
equiv="Content-Type"/>
  </head>
  <body>
    <!-- Include the JS generated by `wasm-pack build` -->
    <script src='without_a_bundler_no_modules.js'></script>

    <script>
       // Like with the `--target web` output the exports are
immediately
      // available but they won't work until we initialize the
module. Unlike
       // `--target web`, however, the globals are all stored
on a
         // `wasm_bindgen` global. The global itself is the
initialization
       // function and then the properties of the global are
```

```
all the exported
      // functions.
      //
       // Note that the name `wasm_bindgen` can be configured
with the
      // `--no-modules-global` CLI flag
      const { add } = wasm_bindgen;

      async function run() {
        await wasm_bindgen();

        const result = add(1, 2);
        console.log(`1 + 2 = ${result}`);
      }

      run();
    </script>
  </body>
</html>
```

Synchronous Instantiation

View full source code

This example shows how to synchronously initialize a WebAssembly module as opposed to asynchronously. In most cases, the default way of asynchronously initializing a module will suffice. However, there might be use cases where you'd like to lazy load a module on demand and synchronously compile and instantiate it. Note that this only works off the main thread and since compilation and instantiation of large modules can be expensive you should only use this method if it's absolutely required in your use case. Otherwise you should use the default method.

For this deployment strategy bundlers like Webpack are not required. For more information on deployment see the dedicated documentation.

First let's take a look at our tiny lib:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(value: &str);
}

#[wasm_bindgen]
pub fn greet(name: &str) {
    log(&format!("Hello, {name}!"));
}
```

Next, let's have a look at the `index.html`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
         <meta   name="viewport"   content="width=device-width,
```

```
initial-scale=1.0" />
    <title>Document</title>
  </head>
  <body>
    <script>
      /**
       * First off we spawn a Web Worker. That's where our lib
will be used. Note that
       * we set the `type` to `module` to enable support for
ES modules.
       */
      const worker = new Worker("worker.js", { type: "module"
});

      /**
       * Here we listen for messages from the worker.
       */
      worker.onmessage = ({ data }) => {
        const { type } = data;

        switch (type) {
          case "FETCH_WASM": {
            /**
               * The worker wants to fetch the bytes for the
module and for that we can use the `fetch` API.
                    * Then we convert the response into an
`ArrayBuffer` and transfer the bytes back to the worker.
             *
                     * @see https://developer.mozilla.org/en-
US/docs/Web/API/Fetch_API
                     * @see https://developer.mozilla.org/en-
US/docs/Glossary/Transferable_objects
             */
            fetch("synchronous_instantiation_bg.wasm")
              .then((response) => response.arrayBuffer())
              .then((bytes) => {
```

```
              worker.postMessage(bytes, [bytes]);
            });
          break;
        }
        default: {
          break;
        }
      }
    };
  </script>
</body>
</html>
```

Otherwise the rest of the magic happens in `worker.js`:

```
import * as wasm from "./synchronous_instantiation.js";

self.onmessage = ({ data: bytes }) => {
  /**
   * When we receive the bytes as an `ArrayBuffer` we can use that to
   * synchronously initialize the module as opposed to asynchronously
   * via the default export. The synchronous method internally uses
   * `new WebAssembly.Module()` and `new WebAssembly.Instance()`.
   */
  wasm.initSync({ module: bytes });

  /**
   * Once initialized we can call our exported `greet()` functions.
   */
  wasm.greet("Dominic");
};
```

```
/**
 * Once the Web Worker was spawned we ask the main thread to
fetch the bytes
 * for the WebAssembly module. Once fetched it will send the
bytes back via
 * a `postMessage` (see above).
 */
self.postMessage({ type: "FETCH_WASM" });
```

And that's it! Be sure to read up on the deployment options to see what it means to deploy without a bundler.

Importing non-browser JS

View full source code or view the compiled example online

The `#[wasm_bindgen]` attribute can be used on `extern "C" { .. }` blocks to import functionality from JS. This is how the `js-sys` and the `web-sys` crates are built, but you can also use it in your own crate!

For example if you're working with this JS file:

```
// defined-in-js.js
export function name() {
    return 'Rust';
}

export class MyClass {
    constructor() {
        this._number = 42;
    }

    get number() {
        return this._number;
    }

    set number(n) {
        return this._number = n;
    }

    render() {
        return `My number is: ${this.number}`;
    }
}
```

you can use it in Rust with:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen(module = "/defined-in-js.js")]
```

```
extern "C" {
    fn name() -> String;

    type MyClass;

    #[wasm_bindgen(constructor)]
    fn new() -> MyClass;

    #[wasm_bindgen(method, getter)]
    fn number(this: &MyClass) -> u32;
    #[wasm_bindgen(method, setter)]
    fn set_number(this: &MyClass, number: u32) -> MyClass;
    #[wasm_bindgen(method)]
    fn render(this: &MyClass) -> String;
}

// lifted from the `console_log` example
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
}

#[wasm_bindgen(start)]
fn run() {
     log(&format!("Hello from {}!", name())); // should output
"Hello from Rust!"

    let x = MyClass::new();
    assert_eq!(x.number(), 42);
    x.set_number(10);
    log(&x.render());
}
```

You can also [explore the full list of ways to configure imports]

Working with the `char` type

View full source code or view the compiled example online

The `#[wasm_bindgen]` macro will convert the rust `char` type to a single code-point js `string`, and this example shows how to work with this.

Opening this example should display a single counter with a random character for it's `key` and 0 for its `count`. You can click the `+` button to increase a counter's count. By clicking on the "add counter" button you should see a new counter added to the list with a different random character for it's `key`.

Under the hood javascript is choosing a random character from an Array of characters and passing that to the rust Counter struct's constructor so the character you are seeing on the page has made the full round trip from js to rust and back to js.

src/lib.rs

```
use wasm_bindgen::prelude::*;

// lifted from the `console_log` example
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
}

#[wasm_bindgen]
#[derive(Debug)]
pub struct Counter {
    key: char,
    count: i32,
}

#[wasm_bindgen]
impl Counter {
    pub fn new(key: char, count: i32) -> Counter {
        log(&format!("Counter::new({key}, {count})"));
        Counter { key, count }
    }

    pub fn key(&self) -> char {
        log("Counter.key()");
        self.key
    }

    pub fn count(&self) -> i32 {
        log("Counter.count");
        self.count
    }
```

```
    pub fn increment(&mut self) {
        log("Counter.increment");
        self.count += 1;
    }

    pub fn update_key(&mut self, key: char) {
        self.key = key;
    }
}
```

index.js

```
/* eslint-disable no-unused-vars */
import { chars } from './chars-list.js';
let imp = import('./pkg');
let mod;

let counters = [];
imp
  .then(wasm => {
      mod = wasm;
      addCounter();
      let b = document.getElementById('add-counter');
      if (!b) throw new Error('Unable to find #add-counter');
      b.addEventListener('click', ev => addCounter());
  })
  .catch(console.error);

function addCounter() {
    let ctr = mod.Counter.new(randomChar(), 0);
    counters.push(ctr);
    update();
}

function update() {
    let container = document.getElementById('container');
    if (!container) throw new Error('Unable to find #container
in dom');
    while (container.hasChildNodes()) {
        if (container.lastChild.id == 'add-counter') break;
        container.removeChild(container.lastChild);
    }
    for (var i = 0; i < counters.length; i++) {
        let counter = counters[i];
               container.appendChild(newCounter(counter.key(),
```

```
counter.count(), ev => {
            counter.increment();
            update();
        }));
    }
}

function randomChar() {
    console.log('randomChar');
    let idx = Math.floor(Math.random() * (chars.length - 1));
    console.log('index', idx);
    let ret = chars.splice(idx, 1)[0];
    console.log('char', ret);
    return ret;
}

function newCounter(key, value, cb) {
    let container = document.createElement('div');
    container.setAttribute('class', 'counter');
    let title = document.createElement('h1');
       title.appendChild(document.createTextNode('Counter  '  +
key));
    container.appendChild(title);
    container.appendChild(newField('Count', value));
    let plus = document.createElement('button');
    plus.setAttribute('type', 'button');
    plus.setAttribute('class', 'plus-button');
    plus.appendChild(document.createTextNode('+'));
    plus.addEventListener('click', cb);
    container.appendChild(plus);
    return container;
}

function newField(key, value) {
    let ret = document.createElement('div');
    ret.setAttribute('class', 'field');
```

```
    let name = document.createElement('span');
    name.setAttribute('class', 'name');
    name.appendChild(document.createTextNode(key));
    ret.appendChild(name);
    let val = document.createElement('span');
    val.setAttribute('class', 'value');
    val.appendChild(document.createTextNode(value));
    ret.appendChild(val);
    return ret;
}
```

js-sys: WebAssembly in WebAssembly

View full source code or view the compiled example online

Using the `js-sys` crate we can get pretty meta and instantiate `WebAssembly` modules from inside `WebAssembly` modules!

src/lib.rs

```
use js_sys::{Function, Object, Reflect, WebAssembly};
use wasm_bindgen::prelude::*;
use wasm_bindgen_futures::{spawn_local, JsFuture};

// lifted from the `console_log` example
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(a: &str);
}

macro_rules! console_log {
    ($($t:tt)*) => (log(&format_args!($($t)*).to_string()))
}

const WASM: &[u8] = include_bytes!("add.wasm");

async fn run_async() -> Result<(), JsValue> {
    console_log!("instantiating a new Wasm module directly");

                                        let         a         =
JsFuture::from(WebAssembly::instantiate_buffer(WASM,
&Object::new())).await?;
          let   b:   WebAssembly::Instance   =   Reflect::get(&a,
&"instance".into())?.dyn_into()?;

    let c = b.exports();

    let add = Reflect::get(c.as_ref(), &"add".into())?
        .dyn_into::<Function>()
        .expect("add export wasn't a function");

     let  three  =  add.call2(&JsValue::undefined(),  &1.into(),
```

```
&2.into())?;
    console_log!("1 + 2 = {:?}", three);
    let mem = Reflect::get(c.as_ref(), &"memory".into())?
        .dyn_into::<WebAssembly::Memory>()
                        .expect("memory   export   wasn't   a
`WebAssembly.Memory`");
      console_log!("created module has {} pages of memory",
mem.grow(0));
    console_log!("giving the module 4 more pages of memory");
    mem.grow(4);
      console_log!("now the module has {} pages of memory",
mem.grow(0));

    Ok(())
}

#[wasm_bindgen(start)]
fn run() {
    spawn_local(async {
        run_async().await.unwrap_throw();
    });
}
```

web-sys: DOM hello world

View full source code or view the compiled example online

Using `web-sys` we're able to interact with all the standard web platform methods, including those of the DOM! Here we take a look at a simple "Hello, world!" which manufactures a DOM element in Rust, customizes it, and then appends it to the page.

`Cargo.toml`

You can see here how we depend on `web-sys` and activate associated features to enable all the various APIs:

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "dom"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = { path = "../../" }

[dependencies.web-sys]
features  =  ['Document',  'Element',  'HtmlElement',  'Node',
'Window']
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

```
use wasm_bindgen::prelude::*;

// Called by our JS entry point to run the example
#[wasm_bindgen(start)]
fn run() -> Result<(), JsValue> {
      // Use `web_sys`'s global `window` function to get a
handle on the global
    // window object.
     let window = web_sys::window().expect("no global `window`
exists");
      let document = window.document().expect("should have a
document on window");
     let body = document.body().expect("document should have a
body");

    // Manufacture the element we're gonna append
    let val = document.create_element("p")?;
    val.set_text_content(Some("Hello from Rust!"));

    body.append_child(&val)?;

    Ok(())
}
```

web-sys: Closures

View full source code or view the compiled example online

The `Closure` type allows passing Rust closures to JavaScript. This example demonstrates different `Closure` APIs for various use cases.

Choosing a `Closure` API

- **`&dyn Fn`** / **`&mut dyn FnMut`** — For immediate/synchronous callbacks where JavaScript calls the closure right away and doesn't retain it. Lightweight with no JS wrapper overhead. These are unwind safe by default.
- **`ScopedClosure::borrow`** / **`ScopedClosure::borrow_mut`** — For known-lifetime callbacks where JavaScript may briefly retain the closure but you control when it becomes invalid. `borrow` is for immutable `Fn`, `borrow_mut` is for mutable `FnMut`.
- **`Closure::new`** — For indeterminate-lifetime closures like event handlers or timers. The closure must be `'static` and you must manage its lifetime (store it somewhere or call `forget()`).
- **`Closure::once`** / **`Closure::once_into_js`** — For one-shot callbacks that will only be called once.

All closure types are unwind safe — panics are caught and converted to JavaScript exceptions when built with `panic=unwind`. This requires the closure to implement `UnwindSafe`. If your closure captures non-unwind-safe types (like `Rc<RefCell<_>>`), use `AssertUnwindSafe` to wrap the closure, or use the `*_aborting` variants which don't require `UnwindSafe`.

See Passing Rust Closures to JS for detailed documentation.

src/lib.rs

```
use js_sys::{Array, Date};
use wasm_bindgen::prelude::*;
use web_sys::{Document, Element, HtmlElement, Window};

#[wasm_bindgen(start)]
fn run() -> Result<(), JsValue> {
       let window = web_sys::window().expect("should have a window in this context");
      let document = window.document().expect("window should have a document");

    // Demonstrate ScopedClosure for immediate callbacks.
    // This is the recommended way to pass closures to JS for synchronous use.
       // The closure can capture local references and is automatically cleaned up.
    demonstrate_scoped_closure();

    // Note: js_sys::Array::for_each currently still uses the old `&mut dyn FnMut`
    // pattern, which will be migrated to Closure in a future release.
    let array = Array::new();
    array.push(&"Hello".into());
    array.push(&1.into());
    let mut first_item = None;
    array.for_each(&mut |obj, idx, _arr| match idx {
        0 => {
            assert_eq!(obj, "Hello");
            first_item = obj.as_string();
        }
        1 => assert_eq!(obj, 1),
        _ => panic!("unknown index: {idx}"),
```

```
    });
    assert_eq!(first_item, Some("Hello".to_string()));

     // Below are some more advanced usages of the `Closure` type for closures
    // that need to live beyond our function call.

    setup_clock(&window, &document)?;
    setup_clicker(&document);

      // And now that our demo is ready to go let's switch things up so
      // everything is displayed and our loading prompt is hidden.
    document
        .get_element_by_id("loading")
        .expect("should have #loading on the page")
        .dyn_ref::<HtmlElement>()
        .expect("#loading should be an `HtmlElement`")
        .style()
        .set_property("display", "none")?;
    document
        .get_element_by_id("script")
        .expect("should have #script on the page")
        .dyn_ref::<HtmlElement>()
        .expect("#script should be an `HtmlElement`")
        .style()
        .set_property("display", "block")?;

    Ok(())
}

// Set up a clock on our page and update it each second to ensure it's got
// an accurate date.
//
```

```
// Note the usage of `Closure` here because the closure is "long lived",
// basically meaning it has to persist beyond the call to this one function.
// Also of note here is the `.as_ref().unchecked_ref()` chain, which is how
// you can extract `&Function`, what `web-sys` expects, from a `Closure`
// which only hands you `&JsValue` via `AsRef`.
fn setup_clock(window: &Window, document: &Document) -> Result<(), JsValue> {
    let current_time = document
        .get_element_by_id("current-time")
        .expect("should have #current-time on the page");
    update_time(&current_time);
            let a = Closure::<dyn Fn()>::new(move || update_time(&current_time));
    window

.set_interval_with_callback_and_timeout_and_arguments_0(a.as_ref().unchecked_ref(), 1000)?;
    fn update_time(current_time: &Element) {
        current_time.set_inner_html(&String::from(
                        Date::new_0().to_locale_string("en-GB", &JsValue::undefined()),
        ));
    }

       // The instance of `Closure` that we created will invalidate its
    // corresponding JS callback whenever it is dropped, so if we were to
     // normally return from `setup_clock` then our registered closure will
    // raise an exception when invoked.
    //
```

```
    // Normally we'd store the handle to later get dropped at an appropriate
    // time but for now we want it to be a global handler so we use the
     // `forget` method to drop it without invalidating the closure. Note that
    // this is leaking memory in Rust, so this should be done judiciously!
    a.forget();

    Ok(())
}

// Demonstrate ScopedClosure for immediate/synchronous callbacks.
//
// Use ScopedClosure::borrow (for FnMut) or ScopedClosure::borrow_immutable (for Fn) when
// JavaScript will call the closure immediately and won't retain it. Benefits:
// - Can capture non-'static references (like &mut local_var)
// - Automatic cleanup when ScopedClosure is dropped
// - Lifetime safety: ScopedClosure can't outlive the closure's captured data
// - Unwind safe (panics become JS exceptions)
fn demonstrate_scoped_closure() {
    #[wasm_bindgen(inline_js = r#"
        export function callThreeTimes(cb) {
            cb(1);
            cb(2);
            cb(3);
        }
    "#)]
    extern "C" {
         // This JS function calls the callback immediately, three times
```

```
        fn callThreeTimes(cb: &ScopedClosure<dyn FnMut(u32)>);
    }

    // Example: Using ScopedClosure::borrow to sum values
    // The closure captures &mut sum without requiring 'static
    let mut sum = 0u32;

    {
        let mut func = |value: u32| {
            sum += value;
        };
        let closure = ScopedClosure::borrow_mut(&mut func);
        // Pass the closure to JavaScript - it will be called
synchronously
        // and then invalidated when closure is dropped
        callThreeTimes(&closure);
    }

    assert_eq!(sum, 6);
}

// We also want to count the number of times that our green
square has been
// clicked. Our callback will update the `#num-clicks` div.
//
// This is pretty similar above, but showing how closures can
also implement
// `FnMut()`.
fn setup_clicker(document: &Document) {
    let num_clicks = document
        .get_element_by_id("num-clicks")
        .expect("should have #num-clicks on the page");
    let mut clicks = 0;
    let a = Closure::<dyn FnMut()>::new(move || {
        clicks += 1;
        num_clicks.set_inner_html(&clicks.to_string());
```

```
    });
    document
        .get_element_by_id("green-square")
        .expect("should have #green-square on the page")
        .dyn_ref::<HtmlElement>()
        .expect("#green-square be an `HtmlElement`")
        .set_onclick(Some(a.as_ref().unchecked_ref()));

     // See comments in `setup_clock` above for why we use
`a.forget()`.
    a.forget();
}
```

web-sys: `performance.now`

View full source code or view the compiled example online

Want to profile some Rust code in the browser? No problem! You can use the `performance.now()` API and friends to get timing information to see how long things take.

src/lib.rs

```
use std::time::{Duration, SystemTime, UNIX_EPOCH};

use wasm_bindgen::prelude::*;

// lifted from the `console_log` example
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(a: &str);
}

macro_rules! console_log {
    ($($t:tt)*) => (log(&format_args!($($t)*).to_string()))
}

#[wasm_bindgen(start)]
fn run() {
       let window = web_sys::window().expect("should have a
window in this context");
    let performance = window
        .performance()
        .expect("performance should be available");

         console_log!("the current time (in ms) is {}",
performance.now());

                                    let start =
perf_to_system(performance.timing().request_start());
                                     let end =
perf_to_system(performance.timing().response_end());

                console_log!("request started at {}",
humantime::format_rfc3339(start));
```

```
                console_log!("request    ended    at    {}",
humantime::format_rfc3339(end));
}

fn perf_to_system(amt: f64) -> SystemTime {
    let secs = (amt as u64) / 1_000;
    let nanos = (((amt as u64) % 1_000) as u32) * 1_000_000;
    UNIX_EPOCH + Duration::new(secs, nanos)
}
```

The `fetch` API

View full source code or view the compiled example online

This example uses the `fetch` API to make an HTTP request to the GitHub API and then parses the resulting JSON.

`Cargo.toml`

The `Cargo.toml` enables a number of features related to the `fetch` API and types used: `Headers`, `Request`, etc.

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "fetch"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
js-sys = { path = "../../crates/js-sys" }
wasm-bindgen = { path = "../../" }
wasm-bindgen-futures = { path = "../../crates/futures" }

[dependencies.web-sys]
features    =    ['Headers',    'Request',    'RequestInit',
'RequestMode', 'Response', 'Window']
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

```
use wasm_bindgen::prelude::*;
use wasm_bindgen_futures::JsFuture;
use web_sys::{Request, RequestInit, RequestMode, Response};

#[wasm_bindgen]
pub async fn run(repo: String) -> Result<JsValue, JsValue> {
    let opts = RequestInit::new();
    opts.set_method("GET");
    opts.set_mode(RequestMode::Cors);

                            let       url       =       format!
("https://api.github.com/repos/{repo}/branches/master");

         let   request   =   Request::new_with_str_and_init(&url,
&opts)?;

    request
        .headers()
        .set("Accept", "application/vnd.github.v3+json")?;

    let window = web_sys::window().unwrap();
                                 let        resp_value        =
JsFuture::from(window.fetch_with_request(&request)).await?;

    // `resp_value` is a `Response` object.
    assert!(resp_value.is_instance_of::<Response>());
    let resp: Response = resp_value.dyn_into().unwrap();

    // Convert this other `Promise` into a rust `Future`.
    let json = JsFuture::from(resp.json()?).await?;

    // Send the JSON response back to JS.
```

```
    Ok(json)
}
```

web-sys: Weather report

View full source code

This example makes an HTTP request to OpenWeather API, parses response in JSON and render UI from that JSON. It also shows the usage of `spawn_local` function for handling asynchronous tasks.

Please add your api key in *get_response()* before running this application.

src/lib.rs

```
extern crate chrono;
extern crate reqwest;

use chrono::prelude::DateTime;
use chrono::Utc;
use std::time::{Duration, UNIX_EPOCH};

use gloo::events::EventListener;
use json::JsonValue;
use wasm_bindgen::prelude::*;
use wasm_bindgen_futures::spawn_local;
use web_sys::Document;
use web_sys::Element;
use web_sys::HtmlInputElement;

#[wasm_bindgen(module = "/util.js")]
extern "C" {
    fn initialize(lat: f64, lon: f64);
}

#[wasm_bindgen(start)]
fn run() -> Result<(), JsValue> {
    let window = web_sys::window().expect("no global `window` exists");
     let document = window.document().expect("should have a document on window");
    let body = document.body().expect("document should have a body");

    // Manufacture the element we're gonna append
        let search_div = create_div(&document, "search", "ReportStyles-secondDiv col-md-12");
    let input_box = create_input_box(&document);
```

```
    search_div.append_child(&input_box)?;
    let submit_box = create_submit_box(&document);
    let temp_div = create_div(
        &document,
        "tempDetail",
        "ReportStyles-mainContainer col-md-12 maincontainer",
    );
    let first_div = document.create_element("div")?;
    let second_div = create_div(&document, "second_div", "col-
md-6");
        let  third_div  =  create_div(&document,  "third_div",
"ReportStyles-innerDiv");
        let  fourth_div  =  create_div(&document,  "cityName",
"ReportStyles-city");

    let table_div = document.create_element("table")?;
         table_div.set_class_name("ReportStyles-table  table-
bordered table-striped");

    let tbody_div = document.create_element("tbody")?;

    let ftr_div = document.create_element("tr")?;

    let ftd_div = document.create_element("td")?;
    ftd_div.set_class_name(" ReportStyles-firstTd");

    let img_div = document.create_element("div")?;
    img_div.set_id("temp");

    let std_div = document.create_element("td")?;
    std_div.set_class_name(" ReportStyles-secondTd");

    let weather_div = document.create_element("div")?;
    weather_div.set_id("weather");

    ftr_div.append_child(&ftd_div)?;
```

```
    ftd_div.append_child(&img_div)?;
    ftr_div.append_child(&std_div)?;
    std_div.append_child(&weather_div)?;

    let str_div = document.create_element("tr")?;
    let ptd_div = document.create_element("td")?;
    ptd_div.set_class_name(" ReportStyles-td");
    ptd_div.set_inner_html("Pressure");
    let sptd_div = document.create_element("td")?;
    sptd_div.set_id("pressure");
    str_div.append_child(&ptd_div)?;
    str_div.append_child(&sptd_div)?;

    let ttr_div = document.create_element("tr")?;
    let htd_div = document.create_element("td")?;
    htd_div.set_class_name(" ReportStyles-td");
    htd_div.set_inner_html("Humidity");
    let shtd_div = document.create_element("td")?;
    shtd_div.set_id("humidity");
    ttr_div.append_child(&htd_div)?;
    ttr_div.append_child(&shtd_div)?;

    let sunr_tr_div = document.create_element("tr")?;
    let sunr_td_div = document.create_element("td")?;
    sunr_td_div.set_class_name(" ReportStyles-td");
    sunr_td_div.set_inner_html("Sunrise[UTC]");
    let sunr_s_td_div = document.create_element("td")?;
    sunr_s_td_div.set_id("sunrise");
    sunr_tr_div.append_child(&sunr_td_div)?;
    sunr_tr_div.append_child(&sunr_s_td_div)?;

    let suns_tr_div = document.create_element("tr")?;
    let suns_td_div = document.create_element("td")?;
    suns_td_div.set_class_name(" ReportStyles-td");
    suns_td_div.set_inner_html("Sunset[UTC]");
    let suns_s_td_div = document.create_element("td")?;
```

```
    suns_s_td_div.set_id("sunset");
    suns_tr_div.append_child(&suns_td_div)?;
    suns_tr_div.append_child(&suns_s_td_div)?;

    let geo_tr_div = document.create_element("tr")?;
    let geo_htd_div = document.create_element("td")?;
    geo_htd_div.set_class_name(" ReportStyles-td");
    geo_htd_div.set_inner_html("Geo coords");
    let geo_shtd_div = document.create_element("td")?;
    geo_shtd_div.set_id("geocoords");
    geo_tr_div.append_child(&geo_htd_div)?;
    geo_tr_div.append_child(&geo_shtd_div)?;

    tbody_div.append_child(&ftr_div)?;
    tbody_div.append_child(&str_div)?;
    tbody_div.append_child(&ttr_div)?;
    tbody_div.append_child(&sunr_tr_div)?;
    tbody_div.append_child(&suns_tr_div)?;
    tbody_div.append_child(&geo_tr_div)?;

    table_div.append_child(&tbody_div)?;
    third_div.append_child(&fourth_div)?;
    third_div.append_child(&table_div)?;

    let map_div = document.create_element("div")?;
    map_div.set_class_name("col-md-6");
    let map_canvas_div = document.create_element("div")?;
    map_canvas_div.set_class_name(" ReportStyles-mapCanvas");
    map_canvas_div.set_id("map_canvas");
    map_div.append_child(&map_canvas_div)?;

    second_div.append_child(&third_div)?;
    first_div.append_child(&second_div)?;
    first_div.append_child(&map_div)?;

    temp_div.append_child(&first_div)?;
```

```
    search_div.append_child(&submit_box)?;
    body.append_child(&search_div)?;
    body.append_child(&temp_div)?;

     let on_click = EventListener::new(&submit_box, "click",
move |_event| {
        let input_value = document
            .get_element_by_id("name")
            .unwrap()
            .dyn_into::<HtmlInputElement>()
            .unwrap()
            .value();
        let temp_d = temp_div.clone();
        let city = fourth_div.clone();
        let image = img_div.clone();
        let weather = weather_div.clone();
        let pressure = sptd_div.clone();
        let humidity = shtd_div.clone();
        let sunrise = sunr_s_td_div.clone();
        let sunset = suns_s_td_div.clone();
        let geo = geo_shtd_div.clone();
                        let input_value: &'static _ =
Box::leak(Box::new(input_value));
        let response = get_response(input_value);
        spawn_local(async move {
            let parsed = response.await;
                                let lon = parsed["coord"]
["lon"].to_owned().as_f64().unwrap();
                                let lat = parsed["coord"]
["lat"].to_owned().as_f64().unwrap();
            initialize(lat, lon);
                                    let city_name: &str =
&parsed["name"].to_owned().to_string();
                    let country_name: &str = &parsed["sys"]
["country"].to_owned().to_string();
                            let place = [city_name, ",",
```

```
country_name].concat();
                            let  icon  =  &parsed["weather"][0]
["icon"].to_owned().to_string();
            let src = [
                "<img src='http://openweathermap.org/img/w/",
                icon,
                ".png'>",
                "  ",
            ]
            .concat();
                                let   temp   =   (parsed["main"]
["temp"].to_owned().as_f64().unwrap() - 273.15) as i64;

            let content = [src, temp.to_string()].concat();
                                let  p:  &str  =  &parsed["main"]
["pressure"].to_owned().to_string();
                                let  h:  &str  =  &parsed["main"]
["humidity"].to_owned().to_string();
                                let   sun_r   =   (parsed["sys"]
["sunrise"].to_owned().as_f64().unwrap()) as u64;
                                let   sun_s   =   (parsed["sys"]
["sunset"].to_owned().as_f64().unwrap()) as u64;

            temp_d
                .set_attribute("style", "display: block")
                .expect("failed to set attr style");
            city.set_inner_html(&place);
            image.set_inner_html(&content);
                  weather.set_inner_html(&parsed["weather"][0]
["main"].to_owned().to_string());
            pressure.set_inner_html(&([p, " hpa"].concat()));
            humidity.set_inner_html(&([h, "%"].concat()));
            sunrise.set_inner_html(&get_time(sun_r));
            sunset.set_inner_html(&get_time(sun_s));
             geo.set_inner_html(&(["[", &lon.to_string(), ",",
&lat.to_string(), "]"].concat()));
```

```
        });
    });

      // When a Closure is dropped it will invalidate the
associated JS closure.
     // Here we want JS callback to be alive for the entire
duration of the program.
    // So we used `forget` leak this instance of Closure.
     // It should be used sparingly to ensure the memory leak
doesn't affect the program too much.
    on_click.forget();
    Ok(())
}

fn create_div(document: &Document, id: &str, class: &str) ->
Element {
    let div = document.create_element("div").unwrap();
    div.set_id(id);
    div.set_class_name(class);
    div
}

fn create_submit_box(document: &Document) -> Element {
                       let      submit_box:      Element      =
document.create_element("input").unwrap();
    submit_box
        .set_attribute("type", "button")
        .expect("failed to set attr type to button");
    submit_box
        .set_attribute("value", "Search")
        .expect("failed to set attr value to Search");
    submit_box
        .set_attribute("name", "submit")
        .expect("failed to set attr name to submit");
    submit_box.set_id("submit");
     submit_box.set_class_name(" ReportStyles-bootstrapButton
```

```
btn btn-info");
    submit_box
}

fn create_input_box(document: &Document) -> Element {
    let input_box = document.create_element("input").unwrap();
    input_box
        .set_attribute("name", "name")
        .expect("failed to set attr name to name");
    input_box.set_attribute("value", "Delhi").expect(
        "
    failed to set attr value to Delhi",
    );
    input_box
        .set_attribute("type", "text")
        .expect("failed to set attr type to text");
    input_box
        .set_attribute("placeholder", "Type city name here")
         .expect("Failed to set attr placeholder to Type city
name here");
    input_box.set_id("name");
    input_box.set_class_name("ReportStyles-search");
    input_box
}

// Get response from weather api
async fn get_response(location: &str) -> JsonValue {
                                    let         url1        =
"http://api.openweathermap.org/data/2.5/weather?q=";
    let url2 = "&appid=<apiKey>";

    let url = [url1, location, url2].concat();

                                    let         resp        =
reqwest::get(&url).await.unwrap().text().await.unwrap();
```

```
    json::parse(&resp).unwrap()
}

// Convert millisecond into UTC date
fn get_time(millis: u64) -> String {
    let d = UNIX_EPOCH + Duration::from_secs(millis);
    // Create DateTime from SystemTime
    let datetime = DateTime::<Utc>::from(d);
     // Formats the combined date and time with the specified
format string.
    datetime.format("%H:%M:%S").to_string()
}
```

2D Canvas

View full source code or view the compiled example online

Drawing a smiley face with the 2D canvas API. This is a port of part of this MDN tutorial to `web-sys`.

A smiley face

`Cargo.toml`

The `Cargo.toml` enables features necessary to query the DOM and work with 2D canvas.

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "canvas"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
js-sys = { path = "../../crates/js-sys" }
wasm-bindgen = { path = "../../" }

[dependencies.web-sys]
features = ['CanvasRenderingContext2d', 'Document', 'Element',
'HtmlCanvasElement', 'Window']
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

Gets the `<canvas>` element, creates a 2D rendering context, and draws the smiley face.

```
use std::f64;
use wasm_bindgen::prelude::*;

#[wasm_bindgen(start)]
fn start() {
                                    let         document         =
web_sys::window().unwrap().document().unwrap();
                                      let         canvas         =
document.get_element_by_id("canvas").unwrap();
    let canvas: web_sys::HtmlCanvasElement = canvas
        .dyn_into::<web_sys::HtmlCanvasElement>()
        .map_err(|_| ())
        .unwrap();

    let context = canvas
        .get_context("2d")
        .unwrap()
        .unwrap()
        .dyn_into::<web_sys::CanvasRenderingContext2d>()
        .unwrap();

    context.begin_path();

    // Draw the outer circle.
    context
        .arc(75.0, 75.0, 50.0, 0.0, f64::consts::PI * 2.0)
        .unwrap();

    // Draw the mouth.
    context.move_to(110.0, 75.0);
                context.arc(75.0,    75.0,    35.0,    0.0,
```

```
f64::consts::PI).unwrap();

    // Draw the left eye.
    context.move_to(65.0, 65.0);
    context
        .arc(60.0, 65.0, 5.0, 0.0, f64::consts::PI * 2.0)
        .unwrap();

    // Draw the right eye.
    context.move_to(95.0, 65.0);
    context
        .arc(90.0, 65.0, 5.0, 0.0, f64::consts::PI * 2.0)
        .unwrap();

    context.stroke();
}
```

Julia Set

View full source code or view the compiled example online

While not showing off a lot of `web_sys` API surface area, this example shows a neat fractal that you can make!

index.js

A small bit of glue is added for this example

```
import('./pkg')
    .then(wasm => {
        const canvas = document.getElementById('drawing');
        const ctx = canvas.getContext('2d');

        const realInput = document.getElementById('real');
                                  const    imaginaryInput    =
document.getElementById('imaginary');
        const renderBtn = document.getElementById('render');

        renderBtn.addEventListener('click', () => {
            const real = parseFloat(realInput.value) || 0;
            const imaginary = parseFloat(imaginaryInput.value)
|| 0;
            wasm.draw(ctx, 600, 600, real, imaginary);
        });

        wasm.draw(ctx, 600, 600, -0.15, 0.65);
    })
    .catch(console.error);
```

src/lib.rs

The bulk of the logic is in the generation of the fractal

```
use std::ops::Add;
use wasm_bindgen::prelude::*;
use wasm_bindgen::Clamped;
use web_sys::{CanvasRenderingContext2d, ImageData};

#[wasm_bindgen]
pub fn draw(
    ctx: &CanvasRenderingContext2d,
    width: u32,
    height: u32,
    real: f64,
    imaginary: f64,
) -> Result<(), JsValue> {
    // The real workhorse of this algorithm, generating pixel
data
    let c = Complex { real, imaginary };
    let data = get_julia_set(width, height, c);
                                    let         data         =
ImageData::new_with_u8_clamped_array_and_sh(Clamped(&data),
width, height)?;
    ctx.put_image_data(&data, 0.0, 0.0)
}

fn  get_julia_set(width:  u32,  height:  u32,  c:  Complex)  ->
Vec<u8> {
    let mut data = Vec::new();

    let param_i = 1.5;
    let param_r = 1.5;
    let scale = 0.005;

    for x in 0..width {
```

```
        for y in 0..height {
            let z = Complex {
                real: y as f64 * scale - param_r,
                imaginary: x as f64 * scale - param_i,
            };
            let iter_index = get_iter_index(z, c);
            data.push((iter_index / 4) as u8);
            data.push((iter_index / 2) as u8);
            data.push(iter_index as u8);
            data.push(255);
        }
    }

    data
}

fn get_iter_index(z: Complex, c: Complex) -> u32 {
    let mut iter_index: u32 = 0;
    let mut z = z;
    while iter_index < 900 {
        if z.norm() > 2.0 {
            break;
        }
        z = z.square() + c;
        iter_index += 1;
    }
    iter_index
}

#[derive(Clone, Copy, Debug)]
struct Complex {
    real: f64,
    imaginary: f64,
}

impl Complex {
```

```
    fn square(self) -> Complex {
        let real = (self.real * self.real) - (self.imaginary *
self.imaginary);
        let imaginary = 2.0 * self.real * self.imaginary;
        Complex { real, imaginary }
    }

    fn norm(&self) -> f64 {
                (self.real  *  self.real)  +  (self.imaginary  *
self.imaginary)
    }
}

impl Add<Complex> for Complex {
    type Output = Complex;

    fn add(self, rhs: Complex) -> Complex {
        Complex {
            real: self.real + rhs.real,
            imaginary: self.imaginary + rhs.imaginary,
        }
    }
}
```

WebAudio

View full source code or view the compiled example online

This example creates an FM oscillator using the WebAudio API and `web-sys`.

`Cargo.toml`

The `Cargo.toml` enables the types needed to use the relevant bits of the WebAudio API.

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "webaudio"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = { path = "../../" }

[dependencies.web-sys]
features = [
  'AudioContext',
  'AudioDestinationNode',
  'AudioNode',
  'AudioParam',
  'GainNode',
  'OscillatorNode',
  'OscillatorType',
]
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

The Rust code implements the FM oscillator.

```
use wasm_bindgen::prelude::*;
use web_sys::{AudioContext, OscillatorType};

/// Converts a midi note to frequency
///
/// A midi note is an integer, generally in the range of 21 to
108
pub fn midi_to_freq(note: u8) -> f32 {
    27.5 * 2f32.powf((note as f32 - 21.0) / 12.0)
}

#[wasm_bindgen]
pub struct FmOsc {
    ctx: AudioContext,
    /// The primary oscillator.  This will be the fundamental
frequency
    primary: web_sys::OscillatorNode,

    /// Overall gain (volume) control
    gain: web_sys::GainNode,

    /// Amount of frequency modulation
    fm_gain: web_sys::GainNode,

       /// The oscillator that will modulate the primary
oscillator's frequency
    fm_osc: web_sys::OscillatorNode,

    /// The ratio between the primary frequency and the fm_osc
frequency.
    ///
    /// Generally fractional values like 1/2 or 1/4 sound best
```

```
    fm_freq_ratio: f32,

    fm_gain_ratio: f32,
}

impl Drop for FmOsc {
    fn drop(&mut self) {
        let _ = self.ctx.close();
    }
}

#[wasm_bindgen]
impl FmOsc {
    #[wasm_bindgen(constructor)]
    pub fn new() -> Result<FmOsc, JsValue> {
        let ctx = web_sys::AudioContext::new()?;

        // Create our web audio objects.
        let primary = ctx.create_oscillator()?;
        let fm_osc = ctx.create_oscillator()?;
        let gain = ctx.create_gain()?;
        let fm_gain = ctx.create_gain()?;

        // Some initial settings:
        primary.set_type(OscillatorType::Sine);
        primary.frequency().set_value(440.0); // A4 note
        gain.gain().set_value(0.0); // starts muted
        fm_gain.gain().set_value(0.0); // no initial frequency
modulation
        fm_osc.set_type(OscillatorType::Sine);
        fm_osc.frequency().set_value(0.0);

        // Connect the nodes up!

         // The primary oscillator is routed through the gain
node, so that
```

```
        // it can control the overall output volume.
        primary.connect_with_audio_node(&gain)?;

          // Then connect the gain node to the AudioContext
destination (aka
        // your speakers).
        gain.connect_with_audio_node(&ctx.destination())?;

          // The FM oscillator is connected to its own gain
node, so it can
        // control the amount of modulation.
        fm_osc.connect_with_audio_node(&fm_gain)?;

             // Connect the FM oscillator to the frequency
parameter of the main
         // oscillator, so that the FM node can modulate its
frequency.

fm_gain.connect_with_audio_param(&primary.frequency())?;

        // Start the oscillators!
        primary.start()?;
        fm_osc.start()?;

        Ok(FmOsc {
            ctx,
            primary,
            gain,
            fm_gain,
            fm_osc,
            fm_freq_ratio: 0.0,
            fm_gain_ratio: 0.0,
        })
    }

     /// Sets the gain for this oscillator, between 0.0 and
```

```
1.0.
    #[wasm_bindgen]
    pub fn set_gain(&self, mut gain: f32) {
        gain = gain.clamp(0.0, 1.0);
        self.gain.gain().set_value(gain);
    }

    #[wasm_bindgen]
    pub fn set_primary_frequency(&self, freq: f32) {
        self.primary.frequency().set_value(freq);

         // The frequency of the FM oscillator depends on the
frequency of the
         // primary oscillator, so we update the frequency of
both in this method.
        self.fm_osc.frequency().set_value(self.fm_freq_ratio *
freq);
            self.fm_gain.gain().set_value(self.fm_gain_ratio  *
freq);
    }

    #[wasm_bindgen]
    pub fn set_note(&self, note: u8) {
        let freq = midi_to_freq(note);
        self.set_primary_frequency(freq);
    }

     /// This should be between 0 and 1, though higher values
are accepted.
    #[wasm_bindgen]
    pub fn set_fm_amount(&mut self, amt: f32) {
        self.fm_gain_ratio = amt;

        self.fm_gain
            .gain()
                                .set_value(self.fm_gain_ratio  *
```

```
self.primary.frequency().value());
    }

    /// This should be between 0 and 1, though higher values
are accepted.
    #[wasm_bindgen]
    pub fn set_fm_frequency(&mut self, amt: f32) {
        self.fm_freq_ratio = amt;
        self.fm_osc
            .frequency()
                                .set_value(self.fm_freq_ratio  *
self.primary.frequency().value());
    }
}
```

index.js

A small bit of JavaScript glues the rust module to input widgets and translates events into calls into Wasm code.

```
import('./pkg')
  .then(rust_module => {
    let fm = null;

    const play_button = document.getElementById("play");
    play_button.addEventListener("click", event => {
      if (fm === null) {
        fm = new rust_module.FmOsc();
        fm.set_note(50);
        fm.set_fm_frequency(0);
        fm.set_fm_amount(0);
        fm.set_gain(0.8);
      } else {
        fm.free();
        fm = null;
      }
    });

                            const       primary_slider       =
document.getElementById("primary_input");
    primary_slider.addEventListener("input", event => {
      if (fm) {
        fm.set_note(parseInt(event.target.value));
      }
    });

    const fm_freq = document.getElementById("fm_freq");
    fm_freq.addEventListener("input", event => {
      if (fm) {
        fm.set_fm_frequency(parseFloat(event.target.value));
      }
```

```
  });

  const fm_amount = document.getElementById("fm_amount");
  fm_amount.addEventListener("input", event => {
    if (fm) {
      fm.set_fm_amount(parseFloat(event.target.value));
    }
  });
})
.catch(console.error);
```

WebGL Example

View full source code or view the compiled example online

This example draws a triangle to the screen using the WebGL API.

`Cargo.toml`

The `Cargo.toml` enables features necessary to obtain and use a WebGL rendering context.

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "webgl"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
js-sys = { path = "../../crates/js-sys" }
wasm-bindgen = { path = "../../" }

[dependencies.web-sys]
features = [
  'Document',
  'Element',
  'HtmlCanvasElement',
  'WebGlBuffer',
  'WebGlVertexArrayObject',
  'WebGl2RenderingContext',
  'WebGlProgram',
  'WebGlShader',
  'Window',
]
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

This source file handles all of the necessary logic to obtain a rendering context, compile shaders, fill a buffer with vertex coordinates, and draw a triangle to the screen.

```
use wasm_bindgen::prelude::*;
use      web_sys::{WebGl2RenderingContext,      WebGlProgram,
WebGlShader};

#[wasm_bindgen(start)]
fn start() -> Result<(), JsValue> {
                                    let        document        =
web_sys::window().unwrap().document().unwrap();
                                     let        canvas        =
document.get_element_by_id("canvas").unwrap();
    let canvas: web_sys::HtmlCanvasElement = canvas.dyn_into::
<web_sys::HtmlCanvasElement>()?;

    let context = canvas
        .get_context("webgl2")?
        .unwrap()
        .dyn_into::<WebGl2RenderingContext>()?;

    let vert_shader = compile_shader(
        &context,
        WebGl2RenderingContext::VERTEX_SHADER,
        r##"#version 300 es

        in vec4 position;

        void main() {

            gl_Position = position;
        }
        "##,
```

```
    )?;

    let frag_shader = compile_shader(
        &context,
        WebGl2RenderingContext::FRAGMENT_SHADER,
        r##"#version 300 es

        precision highp float;
        out vec4 outColor;

        void main() {
            outColor = vec4(1, 1, 1, 1);
        }
        "##,
    )?;
        let program = link_program(&context, &vert_shader,
&frag_shader)?;
    context.use_program(Some(&program));

    let vertices: [f32; 9] = [-0.7, -0.7, 0.0, 0.7, -0.7, 0.0,
0.0, 0.7, 0.0];

                    let    position_attribute_location    =
context.get_attrib_location(&program, "position");
      let buffer = context.create_buffer().ok_or("Failed to
create buffer")?;
     context.bind_buffer(WebGl2RenderingContext::ARRAY_BUFFER,
Some(&buffer));

     // Note that `Float32Array::view` is somewhat dangerous
(hence the
      // `unsafe`!). This is creating a raw view into our
module's
     // `WebAssembly.Memory` buffer, but if we allocate more
pages for ourself
     // (aka do a memory allocation in Rust) it'll cause the
```

```
buffer to change,
    // causing the `Float32Array` to be invalid.
    //
     // As a result, after `Float32Array::view` we have to be
very careful not to
    // do any memory allocations before it's dropped.
    unsafe {
                            let   positions_array_buf_view   =
js_sys::Float32Array::view(&vertices);

        context.buffer_data_with_array_buffer_view(
            WebGl2RenderingContext::ARRAY_BUFFER,
            &positions_array_buf_view,
            WebGl2RenderingContext::STATIC_DRAW,
        );
    }

    let vao = context
        .create_vertex_array()
        .ok_or("Could not create vertex array object")?;
    context.bind_vertex_array(Some(&vao));

    context.vertex_attrib_pointer_with_i32(
        position_attribute_location as u32,
        3,
        WebGl2RenderingContext::FLOAT,
        false,
        0,
        0,
    );

context.enable_vertex_attrib_array(position_attribute_location
as u32);

    context.bind_vertex_array(Some(&vao));
```

```
    let vert_count = (vertices.len() / 3) as i32;
    draw(&context, vert_count);

    Ok(())
}

fn draw(context: &WebGl2RenderingContext, vert_count: i32) {
    context.clear_color(0.0, 0.0, 0.0, 1.0);
    context.clear(WebGl2RenderingContext::COLOR_BUFFER_BIT);

     context.draw_arrays(WebGl2RenderingContext::TRIANGLES, 0,
vert_count);
}

pub fn compile_shader(
    context: &WebGl2RenderingContext,
    shader_type: u32,
    source: &str,
) -> Result<WebGlShader, String> {
    let shader = context
        .create_shader(shader_type)
         .ok_or_else(|| String::from("Unable to create shader
object"))?;
    context.shader_source(&shader, source);
    context.compile_shader(&shader);

    if context
                                .get_shader_parameter(&shader,
WebGl2RenderingContext::COMPILE_STATUS)
        .as_bool()
        .unwrap_or(false)
    {
        Ok(shader)
    } else {
        Err(context
            .get_shader_info_log(&shader)
```

```
                .unwrap_or_else(|| String::from("Unknown error
creating shader")))
    }
}

pub fn link_program(
    context: &WebGl2RenderingContext,
    vert_shader: &WebGlShader,
    frag_shader: &WebGlShader,
) -> Result<WebGlProgram, String> {
    let program = context
        .create_program()
         .ok_or_else(|| String::from("Unable to create shader
object"))?;

    context.attach_shader(&program, vert_shader);
    context.attach_shader(&program, frag_shader);
    context.link_program(&program);

    if context
                               .get_program_parameter(&program,
WebGl2RenderingContext::LINK_STATUS)
        .as_bool()
        .unwrap_or(false)
    {
        Ok(program)
    } else {
        Err(context
            .get_program_info_log(&program)
                .unwrap_or_else(|| String::from("Unknown error
creating program object")))
    }
}
```

WebSockets Example

View full source code or view the compiled example online

This example connects to an echo server on `wss://echo.websocket.org`, sends a `ping` message, and receives the response.

`Cargo.toml`

The `Cargo.toml` enables features necessary to create a `WebSocket` object and to access events such as `MessageEvent` or `ErrorEvent`.

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "websockets"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
js-sys = { path = "../../crates/js-sys" }
wasm-bindgen = { path = "../../" }

[dependencies.web-sys]
features = [
  "BinaryType",
  "Blob",
  "ErrorEvent",
  "FileReader",
  "MessageEvent",
  "ProgressEvent",
  "WebSocket",
]
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

This code shows the basic steps required to work with a `WebSocket`. At first it opens the connection, then subscribes to events `onmessage`, `onerror`, `onopen`. After the socket is opened it sends a `ping` message, receives an echoed response and prints it to the browser console.

```
use wasm_bindgen::prelude::*;
use web_sys::{ErrorEvent, MessageEvent, WebSocket};

macro_rules! console_log {
    ($($t:tt)*) => (log(&format_args!($($t)*).to_string()))
}

#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
}

#[wasm_bindgen(start)]
fn start_websocket() -> Result<(), JsValue> {
    // Connect to an echo server
    let ws = WebSocket::new("wss://echo.websocket.org")?;
     // For small binary messages, like CBOR, Arraybuffer is
more efficient than Blob handling
    ws.set_binary_type(web_sys::BinaryType::Arraybuffer);
    // create callback
    let cloned_ws = ws.clone();
    let onmessage_callback = Closure::<dyn FnMut(_)>::new(move
|e: MessageEvent| {
        // Handle difference Text/Binary,...
                    if  let  Ok(abuf)  =  e.data().dyn_into::
<js_sys::ArrayBuffer>() {
            console_log!("message event, received arraybuffer:
{:?}", abuf);
```

```
            let array = js_sys::Uint8Array::new(&abuf);
            let len = array.byte_length() as usize;
            console_log!("Arraybuffer received {}bytes: {:?}",
len, array.to_vec());
             // here you can for example use Serde Deserialize
decode the message
              // for demo purposes we switch back to Blob-type
and send off another binary message

cloned_ws.set_binary_type(web_sys::BinaryType::Blob);
             match cloned_ws.send_with_u8_array(&[5, 6, 7, 8])
{
                        Ok(_) => console_log!("binary message
successfully sent"),
                      Err(err) => console_log!("error sending
message: {:?}", err),
            }
              } else if let Ok(blob) = e.data().dyn_into::
<web_sys::Blob>() {
            console_log!("message event, received blob: {:?}",
blob);
             // better alternative to juggling with FileReader
is to use https://crates.io/crates/gloo-file
            let fr = web_sys::FileReader::new().unwrap();
            let fr_c = fr.clone();
            // create onLoadEnd callback
                        let onloadend_cb = Closure::<dyn
FnMut(_)>::new(move |_e: web_sys::ProgressEvent| {
                                                let array =
js_sys::Uint8Array::new(&fr_c.result().unwrap());
                let len = array.byte_length() as usize;
                  console_log!("Blob received {}bytes: {:?}",
len, array.to_vec());
                 // here you can for example use the received
image/png data
            });
```

```
fr.set_onloadend(Some(onloadend_cb.as_ref().unchecked_ref()));
               fr.read_as_array_buffer(&blob).expect("blob not
readable");
            onloadend_cb.forget();
              } else if let Ok(txt) = e.data().dyn_into::
<js_sys::JsString>() {
            console_log!("message event, received Text: {:?}",
txt);
        } else {
                console_log!("message event, received Unknown:
{:?}", e.data());
        }
    });
    // set message event handler on WebSocket

ws.set_onmessage(Some(onmessage_callback.as_ref().unchecked_re
f()));
    // forget the callback to keep it alive
    onmessage_callback.forget();

     let onerror_callback = Closure::<dyn FnMut(_)>::new(move
|e: ErrorEvent| {
        console_log!("error event: {:?}", e);
    });

ws.set_onerror(Some(onerror_callback.as_ref().unchecked_ref())
);
    onerror_callback.forget();

    let cloned_ws = ws.clone();
     let onopen_callback = Closure::<dyn FnMut()>::new(move ||
{
        console_log!("socket opened");
        match cloned_ws.send_with_str("ping") {
                  Ok(_) => console_log!("message successfully
```

```
sent"),
                Err(err) => console_log!("error sending message:
{:?}", err),
            }
            // send off binary message
            match cloned_ws.send_with_u8_array(&[0, 1, 2, 3]) {
                Ok(_) => console_log!("binary message successfully
sent"),
                Err(err) => console_log!("error sending message:
{:?}", err),
            }
        });

ws.set_onopen(Some(onopen_callback.as_ref().unchecked_ref()));
        onopen_callback.forget();

        Ok(())
}
```

WebRTC DataChannel Example

View full source code or view the compiled example online

This example creates 2 peer connections and 2 data channels in single browser tab. Send ping/pong between `peer1.dc` and `peer2.dc`.

`Cargo.toml`

The `Cargo.toml` enables features necessary to use WebRTC DataChannel and its negotiation.

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "webrtc_datachannel"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
js-sys = { path = "../../crates/js-sys" }
wasm-bindgen = { path = "../../" }
wasm-bindgen-futures = { path = "../../crates/futures" }

[dependencies.web-sys]
features = [
  "MessageEvent",
  "RtcPeerConnection",
  "RtcSignalingState",
  "RtcSdpType",
  "RtcSessionDescriptionInit",
  "RtcPeerConnectionIceEvent",
  "RtcIceCandidate",
  "RtcDataChannel",
  "RtcDataChannelEvent",
]
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

The Rust code connects WebRTC data channel.

```
use js_sys::Reflect;
use wasm_bindgen::prelude::*;
use wasm_bindgen_futures::JsFuture;
use web_sys::{
       MessageEvent,  RtcDataChannelEvent,  RtcPeerConnection,
RtcPeerConnectionIceEvent, RtcSdpType,
    RtcSessionDescriptionInit,
};

macro_rules! console_log {
    ($($t:tt)*) => (log(&format_args!($($t)*).to_string()))
}
macro_rules! console_warn {
    ($($t:tt)*) => (warn(&format_args!($($t)*).to_string()))
}

#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
    #[wasm_bindgen(js_namespace = console)]
    fn warn(s: &str);
}

#[wasm_bindgen(start)]
async fn start() -> Result<(), JsValue> {
    /*
     * Set up PeerConnections
     * pc1 <=> pc2
     *
     */
    let pc1 = RtcPeerConnection::new()?;
```

```
            console_log!("pc1 created: state {:?}",
pc1.signaling_state());
    let pc2 = RtcPeerConnection::new()?;
            console_log!("pc2 created: state {:?}",
pc2.signaling_state());

    /*
     * Create DataChannel on pc1 to negotiate
     * Message will be shown here after connection established
     *
     */
    let dc1 = pc1.create_data_channel("my-data-channel");
    console_log!("dc1 created: label {:?}", dc1.label());

    let dc1_clone = dc1.clone();
    let onmessage_callback = Closure::<dyn FnMut(_)>::new(move
|ev: MessageEvent| {
        if let Some(message) = ev.data().as_string() {
            console_warn!("{:?}", message);
                        dc1_clone.send_with_str("Pong  from
pc1.dc!").unwrap();
        }
    });

dc1.set_onmessage(Some(onmessage_callback.as_ref().unchecked_r
ef()));
    onmessage_callback.forget();

    /*
     * If negotiation has done, this closure will be called
     *
     */
            let ondatachannel_callback = Closure::<dyn
FnMut(_)>::new(move |ev: RtcDataChannelEvent| {
        let dc2 = ev.channel();
        console_log!("pc2.ondatachannel!: {:?}", dc2.label());
```

```
                    let  onmessage_callback  =  Closure::<dyn
FnMut(_)>::new(move |ev: MessageEvent| {
            if let Some(message) = ev.data().as_string() {
                console_warn!("{:?}", message);
            }
        });

dc2.set_onmessage(Some(onmessage_callback.as_ref().unchecked_r
ef()));
        onmessage_callback.forget();

        let dc2_clone = dc2.clone();
        let onopen_callback = Closure::<dyn FnMut()>::new(move
|| {
                            dc2_clone.send_with_str("Ping  from
pc2.dc!").unwrap();
        });

dc2.set_onopen(Some(onopen_callback.as_ref().unchecked_ref()))
;
        onopen_callback.forget();
    });

pc2.set_ondatachannel(Some(ondatachannel_callback.as_ref().unc
hecked_ref()));
    ondatachannel_callback.forget();

    /*
     * Handle ICE candidate each other
     *
     */
    let pc2_clone = pc2.clone();
    let onicecandidate_callback1 =
                      Closure::<dyn  FnMut(_)>::new(move  |ev:
RtcPeerConnectionIceEvent| {
```

```
            if let Some(candidate) = ev.candidate() {
                    console_log!("pc1.onicecandidate: {:#?}",
candidate.candidate());
                                                    let   _   =
pc2_clone.add_ice_candidate_with_opt_rtc_ice_candidate(Some(&c
andidate));
            }
        });

pc1.set_onicecandidate(Some(onicecandidate_callback1.as_ref().
unchecked_ref()));
    onicecandidate_callback1.forget();

    let pc1_clone = pc1.clone();
    let onicecandidate_callback2 =
                    Closure::<dyn   FnMut(_)>::new(move   |ev:
RtcPeerConnectionIceEvent| {
            if let Some(candidate) = ev.candidate() {
                    console_log!("pc2.onicecandidate: {:#?}",
candidate.candidate());
                                                    let   _   =
pc1_clone.add_ice_candidate_with_opt_rtc_ice_candidate(Some(&c
andidate));
            }
        });

pc2.set_onicecandidate(Some(onicecandidate_callback2.as_ref().
unchecked_ref()));
    onicecandidate_callback2.forget();

    /*
     * Send OFFER from pc1 to pc2
     *
     */
    let offer = JsFuture::from(pc1.create_offer()).await?;
                let    offer_sdp    =    Reflect::get(&offer,
```

```
&JsValue::from_str("sdp"))?
        .as_string()
        .unwrap();
    console_log!("pc1: offer {:?}", offer_sdp);

                                    let        offer_obj        =
RtcSessionDescriptionInit::new(RtcSdpType::Offer);
    offer_obj.set_sdp(&offer_sdp);
    let sld_promise = pc1.set_local_description(&offer_obj);
    JsFuture::from(sld_promise).await?;
    console_log!("pc1: state {:?}", pc1.signaling_state());

    /*
     * Receive OFFER from pc1
     * Create and send ANSWER from pc2 to pc1
     *
     */
                                    let        offer_obj        =
RtcSessionDescriptionInit::new(RtcSdpType::Offer);
    offer_obj.set_sdp(&offer_sdp);
    let srd_promise = pc2.set_remote_description(&offer_obj);
    JsFuture::from(srd_promise).await?;
    console_log!("pc2: state {:?}", pc2.signaling_state());

    let answer = JsFuture::from(pc2.create_answer()).await?;
                let    answer_sdp    =    Reflect::get(&answer,
&JsValue::from_str("sdp"))?
        .as_string()
        .unwrap();
    console_log!("pc2: answer {:?}", answer_sdp);

                                    let        answer_obj        =
RtcSessionDescriptionInit::new(RtcSdpType::Answer);
    answer_obj.set_sdp(&answer_sdp);
    let sld_promise = pc2.set_local_description(&answer_obj);
    JsFuture::from(sld_promise).await?;
```

```
    console_log!("pc2: state {:?}", pc2.signaling_state());

    /*
     * Receive ANSWER from pc2
     *
     */
                                let        answer_obj        =
RtcSessionDescriptionInit::new(RtcSdpType::Answer);
    answer_obj.set_sdp(&answer_sdp);
    let srd_promise = pc1.set_remote_description(&answer_obj);
    JsFuture::from(srd_promise).await?;
    console_log!("pc1: state {:?}", pc1.signaling_state());

    Ok(())
}
```

`web-sys`: A `requestAnimationFrame` Loop

View full source code or view the compiled example online

This is an example of a `requestAnimationFrame` loop using the `web-sys` crate! It renders a count of how many times a `requestAnimationFrame` callback has been invoked and then it breaks out of the `requestAnimationFrame` loop after 300 iterations.

`Cargo.toml`

You can see here how we depend on `web-sys` and activate associated features to enable all the various APIs:

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "request-animation-frame"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = { path = "../../" }

[dependencies.web-sys]
features = ['Document', 'Element', 'HtmlElement', 'Node',
'Window']
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

```
use std::cell::RefCell;
use std::rc::Rc;
use wasm_bindgen::prelude::*;

fn window() -> web_sys::Window {
    web_sys::window().expect("no global `window` exists")
}

fn request_animation_frame(f: &Closure<dyn FnMut()>) {
    window()
        .request_animation_frame(f.as_ref().unchecked_ref())
        .expect("should register `requestAnimationFrame` OK");
}

fn document() -> web_sys::Document {
    window()
        .document()
        .expect("should have a document on window")
}

fn body() -> web_sys::HtmlElement {
    document().body().expect("document should have a body")
}

// This function is automatically invoked after the Wasm module is instantiated.
#[wasm_bindgen(start)]
fn run() -> Result<(), JsValue> {
    // Here we want to call `requestAnimationFrame` in a loop, but only a fixed
      // number of times. After it's done we want all our resources cleaned up. To
      // achieve this we're using an `Rc`. The `Rc` will
```

```
eventually store the
    // closure we want to execute on each frame, but to start
out it contains
    // `None`.
    //
      // After the `Rc` is made we'll actually create the
closure, and the closure
     // will reference one of the `Rc` instances. The other
`Rc` reference is
    // used to store the closure, request the first frame, and
then is dropped
    // by this function.
    //
       // Inside the closure we've got a persistent `Rc`
reference, which we use
    // for all future iterations of the loop
    let f = Rc::new(RefCell::new(None));
    let g = f.clone();

    let mut i = 0;
    *g.borrow_mut() = Some(Closure::new(move || {
        if i > 300 {
            body().set_text_content(Some("All done!"));

            // Drop our handle to this closure so that it will
get cleaned
            // up once we return.
            let _ = f.borrow_mut().take();
            return;
        }

        // Set the body's text content to how many times this
        // requestAnimationFrame callback has fired.
        i += 1;
           let text = format!("requestAnimationFrame has been
called {i} times.");
```

```
        body().set_text_content(Some(&text));

         // Schedule ourself for another requestAnimationFrame
callback.
        request_animation_frame(f.borrow().as_ref().unwrap());
    }));

    request_animation_frame(g.borrow().as_ref().unwrap());
    Ok(())
}
```

Paint Example

View full source code or view the compiled example online

A simple painting program.

`Cargo.toml`

The `Cargo.toml` enables features necessary to work with the DOM, events and 2D canvas.

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "wasm-bindgen-paint"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
js-sys = { path = "../../crates/js-sys" }
wasm-bindgen = { path = "../../" }

[dependencies.web-sys]
features = [
  'CanvasRenderingContext2d',
  'CssStyleDeclaration',
  'Document',
  'Element',
  'EventTarget',
  'HtmlCanvasElement',
  'HtmlElement',
  'MouseEvent',
  'Node',
  'Window',
]
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

Creates the `<canvas>` element, applies a CSS style to it, adds it to the document, get a 2D rendering context and adds listeners for mouse events.

```
use std::cell::Cell;
use std::rc::Rc;
use wasm_bindgen::prelude::*;

#[wasm_bindgen(start)]
fn start() -> Result<(), JsValue> {
                                    let        document        =
web_sys::window().unwrap().document().unwrap();
    let canvas = document
        .create_element("canvas")?
        .dyn_into::<web_sys::HtmlCanvasElement>()?;
    document.body().unwrap().append_child(&canvas)?;
    canvas.set_width(640);
    canvas.set_height(480);
    canvas.style().set_property("border", "solid")?;
    let context = canvas
        .get_context("2d")?
        .unwrap()
        .dyn_into::<web_sys::CanvasRenderingContext2d>()?;
    let context = Rc::new(context);
    let pressed = Rc::new(Cell::new(false));
    {
        let context = context.clone();
        let pressed = pressed.clone();
            let  closure  =  Closure::<dyn  FnMut(_)>::new(move
|event: web_sys::MouseEvent| {
            context.begin_path();
                    context.move_to(event.offset_x()  as  f64,
event.offset_y() as f64);
            pressed.set(true);
        });
```

```
          canvas.add_event_listener_with_callback("mousedown",
closure.as_ref().unchecked_ref())?;
        closure.forget();
    }
    {
        let context = context.clone();
        let pressed = pressed.clone();
             let closure = Closure::<dyn FnMut(_)>::new(move
|event: web_sys::MouseEvent| {
            if pressed.get() {
                    context.line_to(event.offset_x() as f64,
event.offset_y() as f64);
                context.stroke();
                context.begin_path();
                     context.move_to(event.offset_x() as f64,
event.offset_y() as f64);
            }
        });
          canvas.add_event_listener_with_callback("mousemove",
closure.as_ref().unchecked_ref())?;
        closure.forget();
    }
    {
             let closure = Closure::<dyn FnMut(_)>::new(move
|event: web_sys::MouseEvent| {
            pressed.set(false);
                    context.line_to(event.offset_x() as f64,
event.offset_y() as f64);
            context.stroke();
        });
            canvas.add_event_listener_with_callback("mouseup",
closure.as_ref().unchecked_ref())?;
        closure.forget();
    }
```

```
    Ok(())
}
```

Wasm in Web Worker

View full source code

A simple example of parallel execution by spawning a web worker with `web_sys`, loading Wasm code in the web worker and interacting between the main thread and the worker.

Building & compatibility

At the time of this writing, only Chrome supports modules in web workers, e.g. Firefox does not. To have compatibility across browsers, the whole example is set up without relying on ES modules as target. Therefore we have to build with `--target no-modules`. The full command can be found in `build.sh`.

`Cargo.toml`

The `Cargo.toml` enables features necessary to work with the DOM, log output to the JS console, creating a worker and reacting to message events.

```
[package]
authors = ["The wasm-bindgen Developers"]
edition = "2021"
name = "wasm-in-web-worker"
publish = false
version = "0.0.0"

[lib]
crate-type = ["cdylib"]

[dependencies]
console_error_panic_hook = { version = "0.1.6", optional =
true }
wasm-bindgen = { path = "../../" }

[dependencies.web-sys]
features = [
  'console',
  'Document',
  'HtmlElement',
  'HtmlInputElement',
  'MessageEvent',
  'Window',
  'Worker',
]
path = "../../crates/web-sys"

[lints]
workspace = true
```

src/lib.rs

Creates a struct `NumberEval` with methods to act as stateful object in the worker and function `startup` to be launched in the main thread. Also includes internal helper functions `setup_input_oninput_callback` to attach a `wasm_bindgen::Closure` as callback to the `oninput` event of the input field and `get_on_msg_callback` to create a `wasm_bindgen::Closure` which is triggered when the worker returns a message.

```
use std::cell::RefCell;
use std::rc::Rc;
use wasm_bindgen::prelude::*;
use    web_sys::{console,    HtmlElement,    HtmlInputElement,
MessageEvent, Worker};

/// A number evaluation struct
///
/// This struct will be the main object which responds to
messages passed to the
/// worker. It stores the last number which it was passed to
have a state. The
/// statefulness is not required in this example but should
show how
/// larger, more complex scenarios with statefulness can be
set up.
#[wasm_bindgen]
pub struct NumberEval {
    number: i32,
}

#[wasm_bindgen]
impl NumberEval {
    /// Create new instance.
    pub fn new() -> NumberEval {
        NumberEval { number: 0 }
```

```
    }

      /// Check if a number is even and store it as last
processed number.
    ///
    /// # Arguments
    ///
      /// * `number` - The number to be checked for being
even/odd.
    pub fn is_even(&mut self, number: i32) -> bool {
        self.number = number;
        self.number % 2 == 0
    }

     /// Get last number that was checked - this method is
added to work with
    /// statefulness.
    pub fn get_last_number(&self) -> i32 {
        self.number
    }
}

/// Run entry point for the main thread.
#[wasm_bindgen]
pub fn startup() {
     // Here, we create our worker. In a larger app, multiple
callbacks should be
       // able to interact with the code in the worker.
Therefore, we wrap it in
       // `Rc<RefCell>` following the interior mutability
pattern. Here, it would
     // not be needed but we include the wrapping anyway as
example.
                              let       worker_handle       =
Rc::new(RefCell::new(Worker::new("./worker.js").unwrap()));
        console::log_1(&"Created a new worker from within
```

```
Wasm".into());

     // Pass the worker to the function which sets up the
`oninput` callback.
    setup_input_oninput_callback(worker_handle);
}

fn                          setup_input_oninput_callback(worker:
Rc<RefCell<web_sys::Worker>>) {
                                    let        document        =
web_sys::window().unwrap().document().unwrap();

     // If our `onmessage` callback should stay valid after
exiting from the
     // `oninput` closure scope, we need to either forget it
(so it is not
     // destroyed) or store it somewhere. To avoid leaking
memory every time we
     // want to receive a response from the worker, we move a
handle into the
     // `oninput` closure to which we will always attach the
last `onmessage`
     // callback. The initial value will not be used and we
silence the warning.
    #[allow(unused_assignments)]
                let    mut    persistent_callback_handle    =
get_on_msg_callback();

    #[allow(unused_assignments)]
    let callback = Closure::new(move || {
        console::log_1(&"oninput callback triggered".into());
                                          let     document     =
web_sys::window().unwrap().document().unwrap();

        let input_field = document
            .get_element_by_id("inputNumber")
```

```
            .expect("#inputNumber should exist");
        let input_field = input_field
            .dyn_ref::<HtmlInputElement>()
                        .expect("#inputNumber  should  be  a
HtmlInputElement");

        // If the value in the field can be parsed to a `i32`,
send it to the
        // worker. Otherwise clear the result field.
        match input_field.value().parse::<i32>() {
            Ok(number) => {
                     // Access worker behind shared handle,
following the interior
                // mutability pattern.
                let worker_handle = &*worker.borrow();
                                                  let   _   =
worker_handle.post_message(&number.into());
                                  persistent_callback_handle  =
get_on_msg_callback();

                     // Since the worker returns the message
asynchronously, we
                 // attach a callback to be triggered when the
worker returns.
                worker_handle

.set_onmessage(Some(persistent_callback_handle.as_ref().unchec
ked_ref()));
            }
            Err(_) => {
                document
                    .get_element_by_id("resultField")
                    .expect("#resultField should exist")
                    .dyn_ref::<HtmlElement>()
                            .expect("#resultField should be a
HtmlInputElement")
```

```
                    .set_inner_text("");
            }
        }
    });

     // Attach the closure as `oninput` callback to the input
field.
    document
        .get_element_by_id("inputNumber")
        .expect("#inputNumber should exist")
        .dyn_ref::<HtmlInputElement>()
        .expect("#inputNumber should be a HtmlInputElement")
        .set_oninput(Some(callback.as_ref().unchecked_ref()));

    // Leaks memory.
    callback.forget();
}

/// Create a closure to act on the message returned by the
worker
fn get_on_msg_callback() -> Closure<dyn FnMut(MessageEvent)> {
    Closure::new(move |event: MessageEvent| {
                console::log_2(&"Received  response:  ".into(),
&event.data());

        let result = match event.data().as_bool().unwrap() {
            true => "even",
            false => "odd",
        };

                                          let     document     =
web_sys::window().unwrap().document().unwrap();
        document
            .get_element_by_id("resultField")
            .expect("#resultField should exist")
            .dyn_ref::<HtmlElement>()
```

```
                            .expect("#resultField  should  be  a
HtmlInputElement")
            .set_inner_text(result);
    })
}
```

`index.html`

Includes the input element `#inputNumber` to type a number into and a HTML element `#resultField` were the result of the evaluation even/odd is written to. Since we require to build with `--target no-modules` to be able to load Wasm code in the worker across browsers, the `index.html` also includes loading both `wasm_in_web_worker.js` and `index.js`.

```
<html>

<head>
            <meta   content="text/html;charset=utf-8"   http-
equiv="Content-Type" />
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <div id="wrapper">
        <h1>Main Thread/Wasm Web Worker Interaction</h1>

        <input type="text" id="inputNumber">

        <div id="resultField"></div>
    </div>

    <!-- Make `wasm_bindgen` available for `index.js` -->
    <script src='./wasm_in_web_worker.js'></script>
     <!-- Note that there is no `type="module"` in the script
tag -->
    <script src="./index.js"></script>

</body>

</html>
```

index.js

Loads our Wasm file asynchronously and calls the entry point `startup` of the main thread which will create a worker.

```
// We only need `startup` here which is the main entry point
// In theory, we could also use all other functions/struct types from Rust which we have bound with
// `#[wasm_bindgen]`
const {startup} = wasm_bindgen;

async function run_wasm() {
    // Load the Wasm file by awaiting the Promise returned by `wasm_bindgen`
    // `wasm_bindgen` was imported in `index.html`
    await wasm_bindgen();

    console.log('index.js loaded');

    // Run main Wasm entry point
     // This will create a worker from within our Rust code compiled to Wasm
    startup();
}

run_wasm();
```

`worker.js`

Loads our Wasm file by first importing `wasm_bindgen` via `importScripts('./pkg/wasm_in_web_worker.js')` and then awaiting the Promise returned by `wasm_bindgen(...)`. Creates a new object to do the background calculation and bind a method of the object to the `onmessage` callback of the worker.

```
// The worker has its own scope and no direct access to functions/objects of the
// global scope. We import the generated JS file to make `wasm_bindgen`
// available which we need to initialize our Wasm code.
importScripts('./wasm_in_web_worker.js');

console.log('Initializing worker')

// In the worker, we have a different struct that we want to use as in
// `index.js`.
const {NumberEval} = wasm_bindgen;

async function init_wasm_in_worker() {
    // Load the Wasm file by awaiting the Promise returned by `wasm_bindgen`.
    await wasm_bindgen('./wasm_in_web_worker_bg.wasm');

    // Create a new object of the `NumberEval` struct.
    var num_eval = NumberEval.new();

    // Set callback to handle messages passed to the worker.
    self.onmessage = async event => {
            // By using methods of a struct as reaction to messages passed to the
        // worker, we can preserve our state between messages.
```

```
        var worker_result = num_eval.is_even(event.data);

          // Send response back to be handled by callback in
main thread.
        self.postMessage(worker_result);
    };
};

init_wasm_in_worker();
```

Parallel Raytracing

View full source code or view the compiled example online

This is an example of using threads with WebAssembly, Rust, and `wasm-bindgen`, culminating in a parallel raytracer demo. There's a number of moving pieces to this demo and it's unfortunately not the easiest thing to wrangle, but it's hoped that this'll give you a bit of a taste of what it's like to use threads and Wasm with Rust on the web.

Building the demo

One of the major gotchas with threaded WebAssembly is that Rust does not ship a precompiled target (e.g. standard library) which has threading support enabled. This means that you'll need to recompile the standard library with the appropriate rustc flags, namely `-C target-feature=+atomics`. Note that this requires a nightly Rust toolchain.

To do this you can use the `RUSTFLAGS` environment variable that Cargo reads:

```
export RUSTFLAGS='-C target-feature=+atomics'
```

To recompile the standard library it's recommended to use Cargo's `-Zbuild-std` feature:

```
cargo build --target wasm32-unknown-unknown -Z build-std=panic_abort,std
```

Note that you can also configure this via `.cargo/config.toml`:

```
[unstable]
build-std = ['std', 'panic_abort']

[build]
target = "wasm32-unknown-unknown"
rustflags = '-Ctarget-feature=+atomics'
```

After this `cargo build` should produce a WebAssembly file with threading enabled, and the standard library will be appropriately compiled

as well.

The final step in this is to run `wasm-bindgen` as usual, and `wasm-bindgen` needs no extra configuration to work with threads. You can continue to run it through `wasm-pack`, for example.

Running the demo

Currently it's required to use the `--target no-modules` or `--target web` flag with `wasm-bindgen` to run threaded code. This is because the WebAssembly file imports memory instead of exporting it, so we need to hook initialization of the wasm module at this time to provide the appropriate memory object. This demo uses `--target no-modules`, because Firefox does not support modules in workers.

With `--target no-modules` you'll be able to use `importScripts` inside of each web worker to import the shim JS generated by `wasm-bindgen` as well as calling the `wasm_bindgen` initialization function with the shared memory instance from the main thread. The expected usage is that WebAssembly on the main thread will post its memory object to all other threads to get instantiated with.

Caveats

Unfortunately at this time running Wasm on the web with threads has a number of caveats, although some are specific to just `wasm-bindgen`. These are some pieces to consider and watch out for, although we're always looking for improvements to be made so if you have an idea please file an issue!

- The main thread in a browser cannot block. This means that if you run WebAssembly code on the main thread you can *never* block, meaning you can't do so much as acquire a mutex. This is an extremely difficult limitation to work with on the web, although one workaround is to run Wasm exclusively in web workers and run JS on the main thread. It is possible to run the same wasm across all threads, but you need to be extremely vigilant about synchronization with the main thread.

- Setting up a threaded environment is a bit wonky and doesn't feel smooth today. For example `--target bundler` is unsupported and very specific shims are required on both the main thread and worker threads. These are possible to work with but are somewhat brittle since there's no standard way to spin up web workers as Wasm threads.
- There is no standard notion of a "thread". For example the standard library has no viable route to implement the `std::thread` module. As a consequence there is no concept of thread exit and TLS destructors will never run. We do expose a helper, `__wbindgen_thread_destroy`, that deallocates the thread stack and TLS. If you invoke it, it *must* be the last function you invoke from the Wasm module for a given thread.
- Any thread launched after the first one *might attempt to block* implicitly in its initialization routine. This is a constraint introduced by the way we set up the space for thread stacks and TLS. This means that if you attempt to run a Wasm module in the main thread *after* you are already running it in a worker, it might fail.
- Web Workers executing WebAssembly code cannot receive events from JS. A Web Worker has to fully return back to the browser (and ideally should do so occasionally) to receive JS messages and such. This means that common paradigms like a rayon thread pool do not apply straightforward-ly to the web. The intention of the web is that all long-term blocking happens in the browser itself, not in each thread, but many crates in the ecosystem leveraging threading are not necessarily engineered this way.

These caveats are all largely inherited from the web platform itself, and they're important to consider when designing an application for threading. It's highly unlikely that you can pull a crate off the shelf and "just use it" due to these limitations. You'll need to be sure to carefully plan ahead and ensure that gotchas such as these don't cause issues in the future. As mentioned before though we're always trying to actively develop this support so if folks have ideas about how to improve, or if web standards change, we'll try to update this documentation!

Browser Requirements

This demo should work in the latest Firefox and Chrome versions at this time, and other browsers are likely to follow suit. Note that threads and `SharedArrayBuffer` require HTTP headers to be set to work correctly. For more information see the documentation on MDN under "Security requirements" as well as Firefox's rollout blog post. This means that during local development you'll need to configure your web server appropriately or enable a workaround in your browser.

Wasm audio worklet

View full source code or view the compiled example online

This is an example of using threads inside specific worklets with WebAssembly, Rust, and `wasm-bindgen`, culminating in an oscillator demo. This demo should complement the parallel-raytrace example by demonstrating an alternative approach using ES modules with on-the-fly module creation.

Building the demo

One of the major gotchas with threaded WebAssembly is that Rust does not ship a precompiled target (e.g. standard library) which has threading support enabled. This means that you'll need to recompile the standard library with the appropriate rustc flags, namely `-C target-feature=+atomics`. Note that this requires a nightly Rust toolchain. See the more detailed instructions of the parallel-raytrace example.

Caveats

This example shares most of its caveats with the parallel-raytrace example. However, it tries to encapsulate worklet creation in a Rust module, so the application developer does not need to maintain custom JS code.

Browser Requirements

This demo should work in the latest Chrome, Firefox and Safari versions at this time. Note that this example requires HTTP headers to be set like in parallel-raytrace.

TODO MVC using wasm-bingen and web-sys

View full source code or view the compiled example online

wasm-bindgen and web-sys coded TODO MVC

The code was rewritten from the ES6 version.

The core differences are:

- Having an Element wrapper that takes care of dyn and into refs in web-sys,
- A Scheduler that allows Controller and View to communicate to each other by emulating something similar to the JS event loop.

Size

The size of the project hasn't undergone much work to make it optimised yet.

- ~96kb release build
- ~76kb optimised with binaryen
- ~28kb brotli compressed

Reference

This section contains reference material for using `wasm-bindgen`. It is not intended to be read start to finish. Instead, it aims to quickly answer questions like:

- Is type X supported as a parameter in a Rust function exported to JavaScript?
- What was that CLI flag to disable ECMAScript modules output, and instead attach the JavaScript bindings directly to `window`?

Working with wasm-bindgen Generics

This guide provides an overview of wasm-bindgen's generic type system, which brings TypeScript-like type safety to Rust-JavaScript interop.

For a complete reference of all generic types and traits, see js-sys.

Table of Contents

- Overview
- Example
- Defining Generic Import Types
    - Generic Type Constraints
- The ErasableGeneric Trait
    - JsGeneric Trait
- Upcasting Types
    - Automatic Upcast Generation
    - Upcast Rules
- Exporting Functions with Generic Types

Overview

Wasm-bindgen generics use **type erasure** to provide rich typing information in Rust while generating efficient JavaScript bindings. Generic parameters exist only in Rust code—they are completely erased in the generated JavaScript.

Currently, all of `js-sys` now supports experimental type erased generics, with `web-sys` still pending.

> When passing a typed value (e.g., `Array<Number>`) to a function expecting a wider type (e.g., `&Array<JsValue>`), use the `upcast()` method: `my_array.upcast()`. This is a zero-cost compile-time conversion.

Example

Consider importing a JS function that returns a Promise for an array of strings. This can be implemented using the typed `Promise<T>` and typed `Array<T>` in js-sys:

```
#[wasm_bindgen]
extern "C" {
    fn getPromiseArray() -> Promise<Array<JsString>>;
}
```

Now, when awaiting the future from the Promise, the types are directly inferred without any further type hints necessary:

```
let promise_array = getPromiseArray();

// inferred as Array<JsString>
let array = JsFuture::from(promise_array).await.unwrap();
for js_string in array {
    // inferred as a JsString already
    let string: String = js_string.into();
}
```

Defining Generic Import Types

Generic types can be defined for all imported JavaScript types.

For example, to define a linked list you might type a `ListNode` type with the following definition:

```
#[wasm_bindgen]
extern "C" {
    pub type ListNode<T>;

    #[wasm_bindgen(constructor)]
    pub fn new<T>(value: T) -> ListNode<T>;

    #[wasm_bindgen(method, getter)]
    pub fn value<T>(this: &ListNode<T>) -> T;

    #[wasm_bindgen(method, getter)]
    pub fn next<T>(this: &ListNode<T>) -> Option<ListNode<T>>;

    #[wasm_bindgen(method, setter)]
           pub   fn   set_next<T>(this:   &ListNode<T>,   next:
Option<ListNode<T>>);
}
```

Usage with concrete types:

```
// Constructor infers ListNode type
let node1 = ListNode::new(JsString::from("first"));
let node2 = ListNode::new(JsString::from("second"));
node1.set_next(Some(node2));

let value = node1.value(); // Returns JsString
let     next_node     =     node1.next();     //     Returns
Option<ListNode<JsString>>
```

Trait bounds and lifetimes are also supported in generics definitions. The generic parameters and predicates are hoisted out of the extern "C" block to form the generics of the Rust wrapper around the JS bindgen function.

Generic Type Constraints

Imported types and imported functions support the full Rust type constraint system.

Traits can also be used to form classes of JS types, for example in js-sys we have:

- `Iterable`, `AsyncIterable`: Traits for objects implementing the iterator protocol via `[Symbol.iterator]()` or `[Symbol.asyncIterator]()`. Uses an associated `Item` type.
- `Promising`: Trait for types that are either `T` or `Promise<T>`. Uses an associated `Resolution` type.

These can be used to constrain function generics, for example a function taking an iterable or a promising value can be written:

```
use js_sys::{Iterable, Promising, Promise, Number};

#[wasm_bindgen]
extern "C" {
    // Takes anything iterable (Array, Set, Generator, etc.)
    fn processIterable<T: Iterable<Item = Number>>(items: T) -
> f64;

    // Takes either a Number or Promise<Number>
    fn getValue<T: Promising<Resolution = Number>>(value: T) -
> Promise<Number>;
}
```

The ErasableGeneric Trait

Generic parameters are erased during compilation, effectively turning `Promise<Array<JsString>>` into a singular `JsValue` for the Wasm Bindgen ABI binding generation, and similarly for values received in turn from JS.

Not all types erase into `JsValue` though, other types like Rust closures, or Rust `Option<Promise>` will erase into their corresponding Rust type with the JS erasure as `Option<JsValue>`.

To capture these semantics, generics are implemented on the `ErasableGeneric` trait. This trait has an associated `Repr` type which is the concrete erased type of the generic for binding generation.

Because all js-sys types implement `ErasableGeneric`, the type system can validate that `Promise<Array<JsString>>::Repr` is equal to `Promise<JsValue>::Repr` ensuring that the static concrete binding generation to works out soundly.

The default erasable generic uses `Repr = JsValue` as the most common case, while `Option<T>` uses `Option<T::Repr>` and `Result<T, E>` similarly uses `Result<T::Repr, E::Repr>`.

`ErasableGeneric<Repr = T>` may be used in trait bounds for generic code where necessary, but implementing it is an unsafe Wasm Bindgen internal without stability guarantees.

JsGeneric Trait

For the common case, it is recommended to use the `JsGeneric` trait bound when needed in generic functions - this is a shorthand for `ErasableGeneric<Repr = JsValue>` and is the primary generic repr used for JavaScript typing.

Types that implement `JsGeneric`:

- All js-sys types: `Object`, `Array`, `Function`, `Promise`, `Map`, `Set`, etc.

- JS primitives: `JsValue`, `Number`, `BigInt`, `Boolean`, `JsString`, `Symbol`
- JS special values: `Undefined`, `Null`
- Wrapper types: `JsOption<T>` (for any `T: JsGeneric`)
- All web-sys generated types
- Custom types imported via `#[wasm_bindgen]` automatically generate this trait on the `JsValue` repr.

Types that do NOT implement `JsGeneric`:

- Rust primitives: `u32`, `i32`, `f64`, `bool`, `char`, `String`, etc.

This means you cannot use Rust primitives directly as generic parameters where `JsGeneric` is required. For example, `Array<u32>` is not valid—use `Array<Number>` instead.

```
use wasm_bindgen::convert::JsGeneric;
use js_sys::{Array, Number};

fn process_js_values<T: JsGeneric>(items: &Array<T>) {
    // T can be Number, JsString, Object, etc. — but not u32
or String
}

// ✓ Works
let numbers: Array<Number> = Array::new_typed();
process_js_values(&numbers);

// ✗ Won't compile — u32 is not JsGeneric
// let numbers: Array<u32> = ...;
```

Upcasting Types

Generic JS types support type-safe upcasting through the `Upcast<Target>` trait, implemented using `FromUpcast<Source>` (just like `From` and `Into`). This enables widening from specific to general types via the `upcast()` and `upcast_into()` methods.

`upcast()` can be thought of as providing the TypeScript-like type safety on top of the erasable generics type system - it will not allow converting from a wider type to a narrower type and in the process providing type safety mechanisms at compile time through the Rust compiler.

This includes Function typing with covariance for return types, closure generics and all other container generic types, all implemented on the `FromUpcast<Source>` covariance primitive.

```
use js_sys::{Array, Number, Object};

// Number → JsValue
let num = Number::from(42);
let js_value: JsValue = num.upcast_into();

// Array<Number> → Array<JsValue>
let num_array: Array<Number> = Array::new_typed();
let js_array: Array<JsValue> = num_array.upcast_into();

// Array<Number> → Object (inheritance)
let obj: Object = num_array.upcast_into();
```

Automatic Upcast Generation

Upcast implementations are automatically generated for all imported JavaScript types based on their `extends` attribute:

```
use wasm_bindgen::prelude::*;
use js_sys::Object;

#[wasm_bindgen]
extern "C" {
```

```
    #[wasm_bindgen(extends = Object)]
    pub type MyCustomType;

    #[wasm_bindgen(extends = Object)]
    pub type MyCollection<T>;
}

// Upcast implementations are automatically generated:
let my_type = MyCustomType::new();
let obj: &Object = my_type.upcast(); // ✓ Object upcast by
ref (from extends)
let js_val: JsValue = my_type.clone().upcast_into(); // ✓
JsValue upcast by value (always generated)
```

Upcasts are always provided both for ref and value conversions.

The following Upcast implementations are automatically generated:

- **JsValue upcast**: All types → `JsValue`
- **Identity upcast**: Non-generic types → themselves
- **Structural upcast**: `Container<T>` → `Container<U>` when `T: Upcast<U>`
- **Inheritance upcast**: For each type in the `extends` attribute

To disable automatic Upcast generation (e.g., for types with custom implementations), use the `no_upcast` attribute:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(extends = Object, no_upcast)]
    pub type MyCustomType;
}
```

Upcast Rules

1. **JsValue upcast**: All types can upcast to `JsValue`
2. **Identity upcast**: Non-generic types can upcast to themselves

3. **Inheritance upcast**: Types upcast to their `extends` targets (e.g., Object)
4. **Structural upcast**: Generic types like `Array<T> → Array<U>` when `T → U`
5. **Nested upcast**: `Promise<Array<Number>> → Promise<Array<JsValue>>`

Exporting Functions with Generic Types

Exported functions support generic types with **concrete** type parameters only.

```
#[wasm_bindgen]
pub fn create_number_array() -> Array<Number> {
    let arr: Array<Number> = Array::new_typed();
    arr.push(&Number::from(1));
    arr
}

#[wasm_bindgen]
pub fn sum_numbers(arr: Array<Number>) -> f64 {
    arr.iter().map(|n| n.value_of()).sum()
}

// ✗ Generic function parameters not supported:
// pub fn create_array<T>(value: T) -> Array<T> { ... }
```

Note: TypeScript generation does not currently support generic types. Exports will appear as `Promise<any>`, `Array<any>`, etc. in generated `.d.ts` files.

Deploying Rust and WebAssembly

At this point in time deploying Rust and WebAssembly to the web or other locations unfortunately isn't a trivial task to do. This page hopes to serve as documentation for the various known options, and as always PRs are welcome to update this if it's out of date!

The methods of deployment and integration here are primarily tied to the `--target` flag.

Value	Summary
`bundler`	Suitable for loading in bundlers like Webpack
`web`	Directly loadable in a web browser
`nodejs`	Loadable via `require` as a Node.js CommonJS module
`deno`	Loadable using imports from Deno modules
`no-modules`	Like `web`, but older and doesn't use ES modules
`experimental-nodejs-module`	Loadable via `import` as a Node.js ESM module.
`module`	Uses the new source phase imports syntax to obtain the compiled WebAssembly module

Bundlers

`--target bundler`

The default output of `wasm-bindgen`, or the `bundler` target, assumes a model where the Wasm module itself is natively an ES module. This model, however, is not natively implemented in any JS implementation at this time. As a result, to consume the default output of `wasm-bindgen` you will need a bundler of some form.

> **Note**: the choice of this default output was done to reflect the trends of the JS ecosystem. While tools other than bundlers don't support Wasm files as native ES modules today they're all very much likely to in the future!

Currently the only known bundler known to be fully compatible with `wasm-bindgen` is webpack. Most examples use webpack, and you can check out the hello world example online to see the details of webpack configuration necessary.

Without a Bundler

`--target web` or `--target no-modules`

If you're not using a bundler but you're still running code in a web browser, `wasm-bindgen` still supports this! For this use case you'll want to use the `--target web` flag. You can check out a full example in the documentation, but the highlights of this output are:

- When compiling you'll pass `--target web` to `wasm-bindgen`
- The output can natively be included on a web page, and doesn't require any further postprocessing. The output is included as an ES module.
- The `--target web` mode is not able to use NPM dependencies.
- You'll want to review the browser requirements for `wasm-bindgen` because no polyfills will be available.

The CLI also supports an output mode called `--target no-modules` which is similar to the `web` target in that it requires manual initialization of the wasm and is intended to be included in web pages without any further postprocessing. See the without a bundler example for some more information about `--target no-modules`.

Node.js

`--target nodejs`

If you're deploying WebAssembly into Node.js (perhaps as an alternative to a native module), then you'll want to pass the `--target nodejs` flag to `wasm-bindgen`.

Like the "without a bundler" strategy, this method of deployment does not require any further postprocessing. The generated JS shims can be `require`'d just like any other Node module (even the `*_bg` Wasm file can be `require`'d as it has a JS shim generated as well).

Note that this method requires a version of Node.js with WebAssembly support, which is currently Node 8 and above.

Node.js Module

`--target experimental-nodejs-module`

If you're deploying WebAssembly into Node.js as a JavaScript module, then you'll want to pass the `--target experimental-nodejs-module` flag to `wasm-bindgen`.

Like the "node" strategy, this method of deployment does not require any further postprocessing. The generated JS shims can be `import`ed just like any other Node module.

Note that this method requires a version of Node.js with WebAssembly and module support, which is currently Node 12 and above.

Currently experimental. Target is expected to be changed before stabilization.

Deno

`--target deno`

To deploy WebAssembly to Deno, use the `--target deno` flag. To then import your module inside deno, use

```
// @deno-types="./out/crate_name.d.ts"
import { yourFunction } from "./out/crate_name.js";
```

Module

`--target module`

The `--target module` option generates a new cross-toolchain ES module target, that by default uses the new source phase imports syntax to obtain uninstantiated WebAssembly modules.

Source phase imports allow importing the compiled WebAssembly module (rather than its instance) using the Wasm ESM Integration.

Note that source phase imports are currently only supported in ESBuild and Node.js 24.

For example:

```
import source wasmModule from "./module.wasm";

// Example illustrative usage:
const instance = new WebAssembly.Instance(wasmModule, imports);
export function foo () {
  instance.doFoo();
}
```

This replaces the need for `compileStreaming` and platform-specific Wasm-loading paths which previously required separate target implementations.

NPM

If you'd like to deploy compiled WebAssembly to NPM, then the tool for the job is `wasm-pack`. More information on this coming soon!

Community Projects

This page showcases libraries, tools and platforms built for or with Wasm Bindgen.

Wasm Bindgen Utility Libraries & Tools

- console_error_panic_hook - Better panic messages in the browser console
- gloo - Modular toolkit providing ergonomic wrappers around browser APIs
- serde-wasm-bindgen - Native integration between serde and wasm-bindgen
- wasm-logger - Logger implementation for wasm-bindgen compatible with the log crate

Platforms using Wasm Bindgen

- Bevy - Data-driven game engine with wasm-bindgen support for browser games
- Cloudflare Workers Rust - Rust SDK for Cloudflare Workers built on wasm-bindgen
- Dioxus - Fullstack framework for web, desktop, and mobile apps with a single codebase
- Leptos - Full-stack framework with fine-grained reactivity and server-side rendering
- Yew - React-like framework for building multi-threaded frontend web apps with WebAssembly

Contributing

Have a project to add? We welcome contributions!

If your project or platform provides ecosystem tooling or support for wasm-bindgen, please open a pull request editing this file at `guide/src/reference/community-projects.md`, while maintaining with alphabetical ordering.

JS Snippets

Often when developing a crate you want to run on the web you'll want to include some JS code here and there. While `js-sys` and `web-sys` cover many needs they don't cover everything, so `wasm-bindgen` supports the ability to write JS code next to your Rust code and have it included in the final output artifact.

To include a local JS file, you'll use the `#[wasm_bindgen(module)]` macro:

```
#[wasm_bindgen(module = "/js/foo.js")]
extern "C" {
    fn add(a: u32, b: u32) -> u32;
}
```

This declaration indicates that all the functions contained in the `extern` block are imported from the file `/js/foo.js`, where the root is relative to the crate root (where `Cargo.toml` is located).

The `/js/foo.js` file will make its way to the final output when `wasm-bindgen` executes, so you can use the `module` annotation in a library without having to worry users of your library!

The JS file itself must be written with ES module syntax:

```
export function add(a, b) {
    return a + b;
}
```

A full design of this feature can be found in RFC 6 as well if you're interested!

Using `inline_js`

In addition to `module = "..."` if you're a macro author you also have the ability to use the `inline_js` attribute:

```
#[wasm_bindgen(inline_js = "export function add(a, b) { return
a + b; }")]
```

```
extern "C" {
    fn add(a: u32, b: u32) -> u32;
}
```

Using `inline_js` indicates that the JS module is specified inline in the attribute itself, and no files are loaded from the filesystem. They have the same limitations and caveats as when using `module`, but can sometimes be easier to generate for macros themselves. It's not recommended for hand-written code to make use of `inline_js` but instead to leverage `module` where possible.

Caveats

While quite useful local JS snippets currently suffer from a few caveats which are important to be aware of. Many of these are temporary though!

- Currently `import` statements are not supported in the JS file. This is a restriction we may lift in the future once we settle on a good way to support this. For now, though, js snippets must be standalone modules and can't import from anything else.
- Only `--target web` and the default bundler output mode are supported. To support `--target nodejs` we'd need to translate ES module syntax to CommonJS (this is planned to be done, just hasn't been done yet). Additionally to support `--target no-modules` we'd have to similarly translate from ES modules to something else.
- Paths in `module = "..."` must currently start with `/`, or be rooted at the crate root. It is intended to eventually support relative paths like `./` and `../`, but it's currently believed that this requires more support in the Rust `proc_macro` crate.

As above, more detail about caveats can be found in RFC 6.

Use of `static` to Access JS Objects

JavaScript modules will often export arbitrary static objects for use with their provided interfaces. These objects can be accessed from Rust by declaring a named `static` in the `extern` block with an `#[wasm_bindgen(thread_local_v2)]` attribute. `wasm-bindgen` will bind a `JsThreadLocal` for these objects, which can be cloned into a `JsValue`.

These values are cached in a thread-local and are meant to bind static values or objects only. For getters which can change their return value or throw see how to import getters.

For example, given the following JavaScript:

```
let COLORS = {
    red: 'rgb(255, 0, 0)',
    green: 'rgb(0, 255, 0)',
    blue: 'rgb(0, 0, 255)',
};
```

`static` can aid in the access of this object from Rust:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(thread_local_v2)]
    static COLORS: JsValue;
}

fn get_colors() -> JsValue {
    COLORS.with(JsValue::clone)
}
```

Since `COLORS` is effectively a JavaScript namespace, we can use the same mechanism to refer directly to namespaces exported from JavaScript modules, and even to exported classes:

```
let namespace = {
    // Members of namespace...
};
```

```
class SomeType {
    // Definition of SomeType...
};

export { SomeType, namespace };
```

The binding for this module:

```
#[wasm_bindgen(module = "/js/some-rollup.js")]
extern "C" {
    // Likewise with the namespace--this refers to the object
directly.
    #[wasm_bindgen(thread_local_v2, js_name = namespace)]
    static NAMESPACE: JsValue;

    // Refer to SomeType's class
    #[wasm_bindgen(thread_local_v2, js_name = SomeType)]
    static SOME_TYPE: JsValue;

    // Other bindings for SomeType
    type SomeType;
    #[wasm_bindgen(constructor)]
    fn new() -> SomeType;
}
```

Optional statics

If you expect the JavaScript value you're trying to access to not always be available you can use `Option<T>` to handle this:

```
extern "C" {
    type Crypto;
    #[wasm_bindgen(thread_local_v2, js_name = crypto)]
    static CRYPTO: Option<Crypto>;
}
```

If `crypto` is not declared or nullish (`null` or `undefined`) in JavaScript, it will simply return `None` in Rust. This will also account for namespaces: it will return `Some(T)` only if all parts are declared and not nullish.

Static strings

Strings can be imported to avoid going through `TextDecoder/Encoder` when requiring just a `JsString`. This can be useful when dealing with environments where `TextDecoder/Encoder` is not available, like in audio worklets.

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(thread_local_v2, static_string)]
    static STRING: JsString = "a string literal";
}
```

Passing Rust Closures to Imported JavaScript Functions

The `ScopedClosure` (with static lifetime alias `Closure`) and `dyn FnMut` types are the way to pass Rust closures to JavaScript. `Closure` and `ScopedClosure` are defined in the `wasm_bindgen` crate and exported in `wasm_bindgen::prelude`.

Closures are **unwind safe** by default: when built with `panic=unwind`, panics inside closures are caught and converted to JavaScript `PanicError` exceptions. See Catching Panics for details.

Choosing a `Closure` API

Use case	Import function signature	Accepts
Immediate/synchronous callbacks	`&dyn Fn` / `&mut dyn FnMut`	`&dyn Fn` / `&mut dyn FnMut` only
Known-lifetime callbacks	`&ScopedClosure<'lifetime, C>`	`&ScopedClosure<'a>`, `&ScopedClosure<'static>`
Indeterminate lifetime	`ScopedClosure<'static, C>`	`ScopedClosure<'static>` only

Constructor Patterns

Type	Constructor	Aborting Constructor	Assert Unwind Safe
`&dyn Fn` / `&mut dyn FnMut`	`&dyn Fn` (Fn) / `&mut dyn FnMut` (FnMut)	**None**	**None**
`&ScopedClosure<'a, C>`	`Closure::borrow` (Fn) / `borrow_mut` (FnMut)	`borrow_aborting` / `borrow_mut_aborting`	`borrow_assert_unwind_safe` / `borrow_mut_assert_unwind_safe`

Type	Constructor	Aborting Constructor	Assert Unwind Safe
`ScopedClosure<'static, C>`	`Closure::own` (`Closure::new`)	`own_aborting`	`own_assert_unwind_safe`
`ScopedClosure<'static, C>` (one-shot)	`Closure::once`	`Closure::once_aborting`	`once_assert_unwind_safe`

`Closure<C>` is a backwards compatible alias for `ScopedClosure<'static, C>`, while providing constructors for arbitrary lifetimes.

The default constructors require `UnwindSafe` when building with `panic=unwind`, and catch panics, converting them to JavaScript `PanicError` exceptions. The `_aborting` variants do NOT require `UnwindSafe` and do NOT catch panics—if the closure panics, the process will abort. See Catching Panics for details.

The `_assert_unwind_safe` variants catch panics but don't require `MaybeUnwindSafe`, enabling type inference with inline closures while still catching panics. Use these when you need inference and are confident the closure is unwind-safe.

Alternatively, you can wrap your closure with `std::panic::AssertUnwindSafe` and use the regular constructors (`new`, `new_mut`, `own`, `borrow`, `borrow_mut`). This is useful when you want to keep using the coercion-based constructors:

```
use std::panic::AssertUnwindSafe;
use wasm_bindgen::prelude::*;

let data = Rc::new(RefCell::new(0));
```

```
let closure = Closure::own(AssertUnwindSafe(move || {
    *data.borrow_mut() += 1;
}));
```

This constructor flexibility allows API consumers to decide on unwind safety behavior at the call site, rather than having it fixed by the function signature. A single function accepting `ScopedClosure <dyn FnMut(u32)>` can be called with closures created via with or without unwind safety, with varying or static lifetimes, and with optional once semantics.

`&dyn Fn` and `&mut dyn FnMut`

The `#[wasm_bindgen]` attribute supports passing closures as `&dyn Fn` or `&mut dyn FnMut` trait object references directly. These are best suited for simple synchronous callbacks that don't outlive the call.

```
#[wasm_bindgen]
extern "C" {
    fn takes_closure(f: &dyn Fn());
    fn takes_mut_closure(f: &mut dyn FnMut());
}
```

Unwind Safety

The `#[wasm_bindgen]` macro automatically injects a `MaybeUnwindSafe` bound on the closure argument for each `&dyn Fn` and `&mut dyn FnMut` parameter. The generated signature is equivalent to:

```
fn takes_mut_closure(f: &mut (impl FnMut() + MaybeUnwindSafe));
```

`MaybeUnwindSafe` is a no-op blanket impl when building without `panic=unwind`, so there is **no breaking change** for existing code. When building with `panic=unwind`, closures that capture types with interior mutability (e.g. `Cell<T>`, `RefCell<T>`) or mutable references (`&mut T`) will require the captured values to be wrapped in `std::panic::AssertUnwindSafe` before capture:

```
use std::panic::AssertUnwindSafe;

let mut count = 0i32;
// &mut i32 is not UnwindSafe — wrap the reference before the closure captures it
let mut count_ref = AssertUnwindSafe(&mut count);
takes_mut_closure(&mut move || {
    **count_ref += 1;
});
```

Closures that only capture plain `Copy` types or `UnwindSafe` values compile without any wrapping:

```
let message = String::from("hello"); // String: UnwindSafe
takes_closure(&move                                        ||
web_sys::console::log_1(&message.clone().into()));
```

For longer-lived closures or when you need explicit control over unwind safety behaviour, use `ScopedClosure` instead.

Known-Lifetime Callbacks with `ScopedClosure`

For longer lived closures, use `ScopedClosure`, which operates in two separate modes depending on whether it uses a static lifetime or a known lifetime.

When typing a JS function from Rust taking a closure argument there are two modes of operation:

1. `&ScopedClosure<'a, T>` **(pass by ref)**: The closure is borrowed out to JS, while retaining ownership in Rust. There is no JS GC finalizer integration. It is disposed by Rust when dropped. Works with any lifetime.
2. `ScopedClosure<'static, T>` **by value (pass by value)**: The closure is passed to JS ownership and integrated with JS GC finalizers. It is disposed entirely by JS. Only works with `'static` closures.

Note that `ScopedClosure<'static, T>` can also be passed by reference (`&ScopedClosure<'static, T>`) if you want Rust to retain ownership of a static closure.

Non-Static Lifetimes (Pass by Reference Only)

For creating a `ScopedClosure<'a, T>` from a non-static lifetime, use `Closure::borrow` (for immutable `Fn`) or `Closure::borrow_mut` (for mutable `FnMut`) when JavaScript may store the callback temporarily but you control when it becomes invalid. The closure is invalidated when the `ScopedClosure` is dropped.

Non-static closures can only be passed by reference since the underlying closure data may live on the stack.

These are unwind safe. For non-unwind-safe closures, use `Closure::borrow_aborting` and `Closure::borrow_mut_aborting` (aborts on panic), or use `Closure::borrow_assert_unwind_safe` to assert unwind safety while still catching panics.

```
use wasm_bindgen::prelude::*;
```

```
#[wasm_bindgen]
extern "C" {
    fn register_callback(cb: &ScopedClosure<dyn FnMut(u32)>);
    fn trigger_callbacks();
}

let mut result = 0;
{
    let mut func = |value: u32| {
        result += value;
    };
    let closure = Closure::borrow_mut(&mut func);
    register_callback(&closure);
    trigger_callbacks();  // Calls our closure
    // closure dropped here, invalidating the JS reference
}
```

The validity of the JavaScript function is tied to the lifetime of the `ScopedClosure` in Rust. **Once a `ScopedClosure` is dropped, it will deallocate its internal memory and invalidate the corresponding JavaScript function so that any further attempts to invoke it raise an exception.**

If JavaScript calls the closure after the `ScopedClosure` is dropped, it will throw: "closure invoked recursively or after being dropped".

Static Lifetimes with `Closure<T> = ScopedClosure<'static, T>`

For `'static` closures, use `Closure::own()` when JavaScript needs to retain the closure for an indeterminate period, such as for event listeners, timers, or callbacks that outlive the current function call.

Note: It is recommended in function signatures to use `ScopedClosure` and not `Closure`, as it can accept both short-lived and static closures `Closure`. This is because `Closure<T>` is an alias

for `ScopedClosure<'static, T>`. In a future release, `Closure` will be directly aliased to `ScopedClosure` instead._

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
      fn setInterval(closure: &Closure<dyn FnMut()>, millis:
u32) -> f64;
    fn clearInterval(token: f64);

    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
}

#[wasm_bindgen]
pub struct Interval {
    closure: Closure<dyn FnMut()>,
    token: f64,
}

impl Interval {
    pub fn new<F: 'static>(millis: u32, f: F) -> Interval
    where
        F: FnMut()
    {
        // Construct a new closure.
        let closure = Closure::new(f);

              // Pass the closure to JS, to run every n
milliseconds.
        let token = setInterval(&closure, millis);

        Interval { closure, token }
    }
}
```

```
// When the Interval is destroyed, clear its `setInterval` timer.
impl Drop for Interval {
    fn drop(&mut self) {
        clearInterval(self.token);
    }
}

// Keep logging "hello" every second until the resulting `Interval` is dropped.
#[wasm_bindgen]
pub fn hello() -> Interval {
    Interval::new(1_000, || log("hello"))
}
```

Transferring Ownership to JavaScript

You can pass a `ScopedClosure` by value to transfer ownership to JavaScript. This is useful for one-shot callbacks where you don't need to retain a handle in Rust:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    fn set_one_shot_callback(cb: ScopedClosure<dyn FnMut()>);
}

let cb = Closure::own(|| {
    // This closure must be 'static
});
set_one_shot_callback(cb);  // Ownership transferred to JS GC, no need to store or forget
```

Note that only `'static` closures (`ScopedClosure<'static, T>`) can be passed by value. Borrowed closures must be passed by reference.

One-Shot Static Closures with `ScopedClosure<'static, T>::once`

Use `Closure::once` (or `Closure::once` alias) for closures that should only be called once, such as Promise handlers. This allows using `FnOnce` closures that consume captured values.

```
use wasm_bindgen::prelude::*;

// Create a closure that consumes a String
let message = String::from("Hello!");
let closure: Closure<dyn FnMut()> = Closure::once(move || {
    // message is moved and consumed here
    web_sys::console::log_1(&message.into());
});
```

If you don't need to cancel the closure early, use `Closure::once_into_js` to convert directly to a `JsValue`. Note that if the JavaScript function is never called, the `FnOnce` and everything it closes over will leak.

```
use wasm_bindgen::prelude::*;

let callback = Closure::once_into_js(move || {
    // This runs when called from JS
});
// callback is a JsValue containing a JS function
```

Panic Handling

When built with `panic=unwind`, all `ScopedClosure` and `dyn Fn` variants catch panics and convert them to JavaScript `PanicError` exceptions. This requires the closure to satisfy Rust's `UnwindSafe` trait.

For more information on enabling panic catching, see Catching Panics.

UnwindSafe Requirement

The closure constructors for `ScopedClosure` all require that closures be `UnwindSafe`. They act as marker traits that indicates a type is safe to use across panic boundaries.

Common "not unwind safe" compiler errors are caused by capturing types with interior mutability:

- `Rc<Cell<_>>`, `Rc<RefCell<_>>`
- Other interior mutability types

The compiler error will indicate which captured type is problematic.

Fix 1: Use aborting variants

If you don't need panic catching, use the `*_aborting` variants (`own_aborting`, `once_aborting`, `Closure::borrow_aborting`, `Closure::borrow_mut_aborting`) which do not require `UnwindSafe`:

```
use std::cell::RefCell;
use std::rc::Rc;
use wasm_bindgen::prelude::*;

let data = Rc::new(RefCell::new(0));

// No UnwindSafe requirement — aborts on panic instead of catching
let closure = Closure::own_aborting(move || {
    *data.borrow_mut() += 1;
});
```

Fix 1b: Use assert_unwind_safe variants

For `ScopedClosure`, you can use `Closure::own_assert_unwind_safe` directly:

```
use std::cell::RefCell;
use std::rc::Rc;
use wasm_bindgen::prelude::*;

let data = Rc::new(RefCell::new(0));

// No UnwindSafe requirement — catches panics, caller asserts
safety
let        closure:        Closure<dyn        FnMut()>        =
Closure::own_assert_unwind_safe(move || {
    *data.borrow_mut() += 1;
});
```

Or use `Closure::wrap_assert_unwind_safe` with a boxed closure:

```
let data = Rc::new(RefCell::new(0));
let        closure:        Closure<dyn        FnMut()>        =
Closure::wrap_assert_unwind_safe(Box::new(move || {
    *data.borrow_mut() += 1;
}));
```

Fix 2: Assert unwind safety

If you need panic catching and are confident your closure is safe to use across panic boundaries, you can use `AssertUnwindSafe`:

```
use std::panic::AssertUnwindSafe;
use wasm_bindgen::prelude::*;

let closure = Closure::new(AssertUnwindSafe(move || {
    // you're asserting this is safe across panics
}));
```

Upcasting `ScopedClosure`

`ScopedClosure` supports full JS type variance within the erasable generic type system via `upcast()` and `upcast_into()`. This is possible since it eagerly moves the Rust function to a JS function on construction, in contrast to `dyn FnMut` whose `Repr` is its Rust fat pointer and not its `JsValue` representation.

This enables covariance and contravariance on argument and return types:

```
use wasm_bindgen::prelude::*;
use js_sys::Number;

// Return type covariance: i32 -> Number -> JsValue
let closure: Closure<dyn Fn() -> i32> = Closure::own(||
42i32);
let _wider: &Closure<dyn Fn() -> JsValue> =
closure.upcast_ref();

// Argument contravariance: JsValue -> Number -> i32
let closure: Closure<dyn Fn(JsValue)> = Closure::own(|_:
JsValue| {});
let _narrower: &Closure<dyn Fn(i32)> = closure.upcast_ref();
```

This works because the JavaScript function doesn't care about Rust's type distinctions—the conversion between `i32`, `Number`, and `JsValue` happens at the JS-Wasm boundary, not inside the closure itself.

Converting Closures to Typed Functions

The `js_sys::Function` type provides methods for type-safe conversion from `Closure`/`ScopedClosure` to typed `Function` with comprehensive covariance support:

Method	Input	Output	Use Case
`from_closure`	`ScopedClosure<'static, C>`	`Function<F>`	Owned closures (transfers ownership to JS)
`closure_ref`	`&ScopedClosure<C>`	`&Function<F>`	Borrowed closures

```
use js_sys::{Function, Number, JsString};
use wasm_bindgen::prelude::*;

// Owned static conversion - transfers ownership to JS
let closure: Closure<dyn FnMut() -> u32> = Closure::new(|| 42);
let func: Function<fn() -> Number> = Function::from_closure(closure);

// Borrowed ScopedClosure conversion
let mut val: u32 = 5;
let mut func = || { val += 1; val };
let closure = ScopedClosure::borrow_mut(&mut func);
let func_ref: &Function<fn() -> Number> = Function::closure_ref(&closure);
```

Passing Closures to Typed Callback APIs

Many JavaScript APIs like `DOMTokenList.forEach` or `Array.forEach` accept typed callback functions. You can pass Rust closures to these APIs using `Function::closure_ref` with `ScopedClosure::borrow_mut`:

```
use js_sys::{Array, Function, JsString, Number, Undefined};
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
      // Simulates APIs like DOMTokenList.forEach that take
typed callbacks
    fn invoke_for_each_callback(
                callback: &Function<fn(JsString, Number) ->
Undefined>,
        items: &Array<JsString>,
    );
}

let items: Array<JsString> = Array::new_typed();
items.push(&JsString::from("apple"));
items.push(&JsString::from("banana"));

let mut results = Vec::new();

// Rust closure borrowing local data
let mut func = |value: JsString, index: Number| {
    results.push((value.as_string().unwrap(), index.value_of()
as u32));
};

//     ScopedClosure     borrows     the     Rust     closure,
Function::closure_ref provides
// a typed &Function reference for the JS callback parameter
invoke_for_each_callback(
         Function::closure_ref(&ScopedClosure::borrow_mut(&mut
func)),
    &items
);
```

```
// After the call, results contains: [("apple", 0), ("banana", 1)]
```

This pattern works with any API expecting a typed `&Function<fn(...) -> ...>` callback, such as the generated `web-sys` bindings for `DOMTokenList::for_each`, `NodeList::for_each`, and similar iteration methods.

Future improvement: If `js-sys` is migrated into `wasm-bindgen` core in a future release, it will be possible to implement `Deref<Target = Function<F>>` for `ScopedClosure`, enabling automatic dereferencing and Rust-to-JS type conversions. This would simplify the above pattern to just `invoke_for_each_callback(&ScopedClosure::borrow_mut(&mut func), &items)`.

Receiving JavaScript Closures in Exported Rust Functions

You can use the `js-sys` crate to access JavaScript's `Function` type, and invoke that function via `Function.prototype.apply` and `Function.prototype.call`.

For example, we can wrap a `Vec<u32>` in a new type, export it to JavaScript, and invoke a JavaScript closure on each member of the `Vec`:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub struct VecU32 {
    xs: Vec<u32>,
}

#[wasm_bindgen]
impl VecU32 {
    pub fn each(&self, f: &js_sys::Function) {
        let this = JsValue::null();
        for &x in &self.xs {
            let x = JsValue::from(x);
            let _ = f.call1(&this, &x);
        }
    }
}
```

Since Rust has no function overloading, the `call#` method also requires a number representing the amount of arguments passed to the JavaScript closure.

Working with a JS `Promise` and a Rust `Future`

Many APIs on the web work with a `Promise`, such as an `async` function in JS. Naturally you'll probably want to interoperate with them from Rust! To do that you can use `js_sys::Promise` together with Rust `async` functions.

Awaiting a `Promise` directly

`js_sys::Promise` implements `IntoFuture`, so you can `.await` it directly in any `async` function. This is the recommended approach:

```
use js_sys::Promise;
use wasm_bindgen::prelude::*;

async fn get_from_js() -> Result<JsValue, JsValue> {
    let promise = Promise::resolve(&42.into());
    let result = promise.await?;
    Ok(result)
}
```

This works out of the box when `wasm-bindgen-futures` is in your dependency tree, which activates the `futures` feature on `js-sys`. You can also enable it explicitly without `wasm-bindgen-futures`:

```
[dependencies]
js-sys = { version = "0.3", features = ["futures"] }
```

For typed promises the awaited value matches the type parameter directly:

```
use js_sys::{Promise, Number};

async fn get_number() -> Result<Number, JsValue> {
    let promise: Promise<Number> = fetch_number();
    let number = promise.await?;  // number: Number
    Ok(number)
}
```

Using `JsFuture` explicitly

If you need a named `Future` type — for example to store it in a struct or pass it around — use `JsFuture` from `js_sys::futures` or `wasm_bindgen_futures`:

```
use js_sys::futures::JsFuture;
use wasm_bindgen::JsValue;

async fn get_from_js() -> Result<JsValue, JsValue> {
    let promise = js_sys::Promise::resolve(&42.into());
    let result = JsFuture::from(promise).await?;
    Ok(result)
}
```

`JsFuture` is also re-exported from `wasm_bindgen_futures` for backwards compatibility:

```
use wasm_bindgen_futures::JsFuture;
```

A successful promise becomes `Ok` and a rejected promise becomes `Err`, corresponding to JS `.then` and `.catch`.

Importing JS async functions

You can import a JS async function directly with an `extern "C"` block, and the promise will be converted to a future automatically. The return type can be `JsValue`, no return at all, or `Result` and `Option` types to primitives or types supporting JsCast conversions:

```
#[wasm_bindgen]
extern "C" {
    async fn async_func_1_ret_number() -> JsValue;
    async fn async_func_2();
    async fn async_func_3_ret_string() -> JsString;
    async fn async_func_4_ret_array() -> Uint8Array;
}

async fn get_from_js() -> f64 {
    async_func_1_ret_number().await.as_f64().unwrap_or(0.0)
}
```

The `async` attribute can be combined with `catch` to manage errors from the JS promise:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(catch)]
    async fn async_func_3() -> Result<JsValue, JsValue>;
    #[wasm_bindgen(catch)]
    async fn async_func_4() -> Result<(), JsValue>;
}
```

Exporting Rust `async fn` to JS

Use an `async` function with `#[wasm_bindgen]` to export a function that returns a `Promise` to JavaScript:

```
#[wasm_bindgen]
pub async fn foo() {
    // ...
}
```

When invoked from JS the `foo` function here will return a `Promise`, so you can use it as:

```
import { foo } from "my-module";

async function shim() {
    const result = await foo();
    // ...
}
```

Return values of `async fn`

When using an `async fn` in Rust and exporting it to JS there are some restrictions on the return type. The return value of an exported Rust function will eventually become `Result<JsValue, JsValue>` where `Ok` turns into a successfully resolved promise and `Err` is equivalent to throwing an exception.

The following types are supported as return types from an `async fn`:

- `()` — turns into a successful `undefined` in JS
- `T: Into<JsValue>` — turns into a successful JS value
- `Result<(), E: Into<JsValue>>` — `Ok(())` resolves to `undefined`; `Err` rejects with `E` converted to a JS value
- `Result<T: Into<JsValue>, E: Into<JsValue>>` — both payloads are converted into a `JsValue`

Note that many types implement being converted into a `JsValue`, such as all imported types via `#[wasm_bindgen]` (aka those in `js-sys` or `web-sys`), primitives like `u32`, and all exported `#[wasm_bindgen]` types. In general, you should be able to write code without having too many explicit conversions, and the macro should take care of the rest!

`spawn_local` and `future_to_promise`

Use `spawn_local` to run a `Future<Output = ()>` on the current thread without returning a `Promise` to JavaScript:

```
use js_sys::futures::spawn_local;

spawn_local(async {
    // drive a future to completion on the JS microtask queue
});
```

Use `future_to_promise` to convert a Rust `Future` into a JavaScript `Promise` explicitly:

```
use js_sys::futures::future_to_promise;
use wasm_bindgen::JsValue;

let promise = future_to_promise(async {
    Ok(JsValue::from(42))
});
```

Both functions are also re-exported from `wasm_bindgen_futures` unchanged.

Using Generic Promise Types

Promises support erasable generic type parameters. With `IntoFuture`, the resolved type flows through directly without a cast:

```
use js_sys::{Promise, Number, Array};

#[wasm_bindgen]
extern "C" {
    fn fetchNumbers() -> Promise<Array<Number>>;
}

async fn process_numbers() -> Result<f64, JsValue> {
    // Await the typed promise — no cast needed
    let numbers: Array<Number> = fetchNumbers().await?;

    let mut sum = 0.0;
    for i in 0..numbers.length() {
        sum += numbers.get(i).value_of();
    }
    Ok(sum)
}
```

Compatibility note

`wasm-bindgen-futures` is now a thin re-export shim. The implementation lives in `js-sys` under the `futures` feature, which `wasm-bindgen-futures` activates automatically when it is a dependency. All existing import paths (`wasm_bindgen_futures::JsFuture`, `wasm_bindgen_futures::spawn_local`, etc.) continue to work without any changes.

Learn more:

- `js_sys::futures` API documentation
- `wasm_bindgen_futures` on crates.io

Iterating over JavaScript Values

Methods That Return `js_sys::Iterator`

Some JavaScript collections have methods for iterating over their values or keys:

- `Map::values`
- `Set::keys`
- etc...

These methods return `js_sys::Iterator`, which is the Rust representation of a JavaScript object that has a `next` method that either returns the next item in the iteration, notes that iteration has completed, or throws an error. That is, `js_sys::Iterator` represents an object that implements the duck-typed JavaScript iteration protocol.

`js_sys::Iterator` and `js_sys::AsyncIterator` also support erasable generic type parameters like `Iterator<T>` and `AsyncIterator<T>`, allowing you to document the expected type of values yielded by the iterator. The generic type is erased at the ABI boundary, but provides type safety and documentation in your Rust code.

`js_sys::Iterator` can be converted into a Rust iterator either by reference (into `js_sys::Iter<'a>`) or by value (into `js_sys::IntoIter`). The Rust iterator will yield items of type `Result<JsValue>`. If it yields an `Ok(...)`, then the JS iterator protocol returned an element. If it yields an `Err(...)`, then the JS iterator protocol threw an exception.

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn count_strings_in_set(set: &js_sys::Set) -> u32 {
    let mut count = 0;

    // Call `keys` to get an iterator over the set's elements.
Because this is
     // in a `for ... in ...` loop, Rust will automatically
call its
```

```
    // `IntoIterator` trait implementation to convert it into a Rust iterator.
    for x in set.keys() {
           // We know the built-in iterator for set elements won't throw
        // exceptions, so just unwrap the element. If this was an untrusted
         // iterator, we might want to explicitly handle the case where it throws
        // an exception instead of returning a `{ value, done }` object.
        let x = x.unwrap();

        // If `x` is a string, increment our count of strings in the set!
        if x.is_string() {
            count += 1;
        }
    }

    count
}
```

Iterating Over Any JavaScript Object that Implements the Iterator Protocol

You could manually test for whether an object implements JS's duck-typed iterator protocol, and if so, convert it into a `js_sys::Iterator` that you can finally iterate over. You don't need to do this by-hand, however, since we bundled this up as [the `js_sys::try_iter` function!]

For example, we can write a function that collects the numbers from any JS iterable and returns them as an `Array`:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn collect_numbers(some_iterable: &JsValue) ->
Result<js_sys::Array, JsValue> {
    let nums = js_sys::Array::new();

                                let iterator =
js_sys::try_iter(some_iterable)?.ok_or_else(|| {
        "need to pass iterable JS values!"
    })?;

    for x in iterator {
          // If the iterator's `next` method throws an error,
propagate it
        // up to the caller.
        let x = x?;

        // If `x` is a number, add it to our array of numbers!
        if x.as_f64().is_some() {
            nums.push(&x);
        }
    }
```

```
    Ok(nums)
}
```

Serializing and Deserializing Arbitrary Data Into and From `JsValue` with Serde

It's possible to pass arbitrary data from Rust to JavaScript by serializing it with Serde. This can be done through the `serde-wasm-bindgen` crate.

Add dependencies

To use `serde-wasm-bindgen`, you first have to add it as a dependency in your `Cargo.toml`. You also need the `serde` crate, with the `derive` feature enabled, to allow your types to be serialized and deserialized with Serde.

```
[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde-wasm-bindgen = "0.4"
```

Derive the `Serialize` and `Deserialize` Traits

Add `#[derive(Serialize, Deserialize)]` to your type. All of your type's members must also be supported by Serde, i.e. their types must also implement the `Serialize` and `Deserialize` traits.

For example, let's say we'd like to pass this `struct` to JavaScript; doing so is not possible in `wasm-bindgen` normally due to the use of `HashMap`s, arrays, and nested `Vec`s. None of those types are supported for sending across the wasm ABI naively, but all of them implement Serde's `Serialize` and `Deserialize`.

Note that we do not need to use the `#[wasm_bindgen]` macro.

```
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize)]
pub struct Example {
    pub field1: HashMap<u32, String>,
    pub field2: Vec<Vec<f32>>,
    pub field3: [f32; 4],
}
```

Send it to JavaScript with `serde_wasm_bindgen::to_value`

Here's a function that will pass an `Example` to JavaScript by serializing it to `JsValue`:

```
#[wasm_bindgen]
pub fn send_example_to_js() -> JsValue {
    let mut field1 = HashMap::new();
    field1.insert(0, String::from("ex"));
    let example = Example {
        field1,
        field2: vec![vec![1., 2.], vec![3., 4.]],
        field3: [1., 2., 3., 4.]
    };

    serde_wasm_bindgen::to_value(&example).unwrap()
}
```

Receive it from JavaScript with `serde_wasm_bindgen::from_value`

Here's a function that will receive a `JsValue` parameter from JavaScript and then deserialize an `Example` from it:

```
#[wasm_bindgen]
pub fn receive_example_from_js(val: JsValue) {
                          let      example:      Example      =
serde_wasm_bindgen::from_value(val).unwrap();
    ...
}
```

JavaScript Usage

In the `JsValue` that JavaScript gets, `field1` will be a `Map`, `field2` will be a JavaScript `Array` whose members are `Array`s of numbers, and `field3` will be an `Array` of numbers.

```
import { send_example_to_js, receive_example_from_js } from "example";

// Get the example object from wasm.
let example = send_example_to_js();

// Add another "Vec" element to the end of the "Vec<Vec<f32>>"
example.field2.push([5, 6]);

// Send the example object back to wasm.
receive_example_from_js(example);
```

An alternative approach - using JSON

`serde-wasm-bindgen` works by directly manipulating JavaScript values. This requires a lot of calls back and forth between Rust and JavaScript, which can sometimes be slow. An alternative way of doing this is to serialize values to JSON, and then parse them on the other end. Browsers' JSON implementations are usually quite fast, and so this approach can outstrip `serde-wasm-bindgen`'s performance in some cases. But this approach supports only types that can be serialized as JSON, leaving out some important types that `serde-wasm-bindgen` supports such as `Map`, `Set`, and array buffers.

That's not to say that using JSON is always faster, though - the JSON approach can be anywhere from 2x to 0.2x the speed of `serde-wasm-bindgen`, depending on the JS runtime and the values being passed. It also leads to larger code size than `serde-wasm-bindgen`. So, make sure to profile each for your own use cases.

This approach is implemented in `gloo_utils::format::JsValueSerdeExt`:

```
# Cargo.toml
[dependencies]
gloo-utils = { version = "0.1", features = ["serde"] }
```

```
use gloo_utils::format::JsValueSerdeExt;

#[wasm_bindgen]
pub fn send_example_to_js() -> JsValue {
    let mut field1 = HashMap::new();
    field1.insert(0, String::from("ex"));
    let example = Example {
        field1,
        field2: vec![vec![1., 2.], vec![3., 4.]],
        field3: [1., 2., 3., 4.]
    };
```

```
    JsValue::from_serde(&example).unwrap()
}

#[wasm_bindgen]
pub fn receive_example_from_js(val: JsValue) {
    let example: Example = val.into_serde().unwrap();
    ...
}
```

History

In previous versions of `wasm-bindgen`, `gloo-utils`'s JSON-based Serde support (`JsValue::from_serde` and `JsValue::into_serde`) was built into `wasm-bindgen` itself. However, this required a dependency on `serde_json`, which had a problem: with certain features of `serde_json` and other crates enabled, `serde_json` would end up with a circular dependency on `wasm-bindgen`, which is illegal in Rust and caused people's code to fail to compile. So, these methods were extracted out into `gloo-utils` with an extension trait and the originals were deprecated.

Accessing Properties of Untyped JavaScript Values

To read and write arbitrary properties from any untyped JavaScript value regardless if it is an `instanceof` some JavaScript class or not, use the `js_sys::Reflect` APIs. These APIs are bindings to the JavaScript builtin `Reflect` object and its methods.

You might also benefit from using duck-typed interfaces instead of working with untyped values.

Reading Properties with `js_sys::Reflect::get`

API documentation for `js_sys::Reflect::get`.

A function that returns the value of a property.

Rust Usage

```
let value = js_sys::Reflect::get(&target, &property_key)?;
```

JavaScript Equivalent

```
let value = target[property_key];
```

Writing Properties with `js_sys::Reflect::set`

API documentation for `js_sys::Reflect::set`.

A function that assigns a value to a property. Returns a boolean that is true if the update was successful.

Rust Usage

```
js_sys::Reflect::set(&target, &property_key, &value)?;
```

JavaScript Equivalent

```
target[property_key] = value;
```

Determining if a Property Exists with `js_sys::Reflect::has`

API documentation for `js_sys::Reflect::has`.

The JavaScript `in` operator as function. Returns a boolean indicating whether an own or inherited property exists on the target.

Rust Usage

```
if js_sys::Reflect::has(&target, &property_key)? {
    // ...
} else {
    // ...
}
```

JavaScript Equivalent

```
if (property_key in target) {
    // ...
} else {
    // ...
}
```

But wait — there's more!

See the `js_sys::Reflect` API documentation for the full listing of JavaScript value reflection and introspection capabilities.

Working with Duck-Typed Interfaces

Liberal use of the `structural` attribute on imported methods, getters, and setters allows you to define duck-typed interfaces. A duck-typed interface is one where many different JavaScript objects that don't share the same base class in their prototype chain and therefore are not `instanceof` the same base can be used the same way.

Defining a Duck-Typed Interface in Rust

```
use wasm_bindgen::prelude::*;

/// Here is a duck-typed interface for any JavaScript object
that has a `quack`
/// method.
///
/// Note that any attempts to check if an object is a `Quacks`
with
/// `JsCast::is_instance_of` (i.e. the `instanceof` operator)
will fail because
/// there is no JS class named `Quacks`.
#[wasm_bindgen]
extern "C" {
    pub type Quacks;

    #[wasm_bindgen(structural, method)]
    pub fn quack(this: &Quacks) -> String;
}

/// Next, we can export a function that takes any object that
quacks:
#[wasm_bindgen]
pub fn make_em_quack_to_this(duck: &Quacks) {
    let _s = duck.quack();
    // ...
}
```

JavaScript Usage

```
import { make_em_quack_to_this } from "./rust_duck_typed_interfaces";

// All of these objects implement the `Quacks` interface!

const alex = {
  quack: () => "you're not wrong..."
};

const ashley = {
  quack: () => "<corgi.gif>"
};

const nick = {
  quack: () => "rappers I monkey-flip em with the funky rhythm I be kickin"
};

// Get all our ducks in a row and call into wasm!

make_em_quack_to_this(alex);
make_em_quack_to_this(ashley);
make_em_quack_to_this(nick);
```

The `wasm-bindgen` Command Line Interface

The `wasm-bindgen` command line tool has a number of options available to it to tweak the JavaScript that is generated. The most up-to-date set of flags can always be listed via `wasm-bindgen --help`.

Installation

```
cargo install -f wasm-bindgen-cli
```

Usage

```
wasm-bindgen [options] ./target/wasm32-unknown-unknown/release/crate.wasm
```

Options

`--out-dir DIR`

The target directory to emit the JavaScript bindings, TypeScript definitions, processed `.wasm` binary, etc...

`--target`

This flag indicates what flavor of output what `wasm-bindgen` should generate. For example it could generate code to be loaded in a bundler like Webpack, a native web page, or Node.js. For a full list of options to pass this flag, see the section on deployment

`--no-modules-global VAR`

When `--target no-modules` is used this flag can indicate what the name of the global to assign generated bindings to.

For more information about this see the section on deployment

`--typescript`

Output a TypeScript declaration file for the generated JavaScript bindings. This is on by default.

`--no-typescript`

By default, a `*.d.ts` TypeScript declaration file is generated for the generated JavaScript bindings, but this flag will disable that.

`--omit-imports`

When the `module` attribute is used with the `wasm-bindgen` macro, the code generator will emit corresponding `import` or `require` statements in the header section of the generated javascript. This flag causes those import statements to be omitted. This is necessary for some use cases, such as generating javascript which is intended to be used with Electron (with node integration disabled), where the imports are instead handled through a separate preload script.

`--debug`

Generates a bit more JS and Wasm in "debug mode" to help catch programmer errors, but this output isn't intended to be shipped to production.

`--no-demangle`

When post-processing the `.wasm` binary, do not demangle Rust symbols in the "names" custom section.

`--keep-lld-exports`

When post-processing the `.wasm` binary, do not remove exports that are synthesized by Rust's linker, LLD.

`--keep-debug`

When post-processing the `.wasm` binary, do not strip DWARF debug info custom sections. See debug information for more.

`--browser`

When generating bundler-compatible code (see the section on deployment) this indicates that the bundled code is always intended to go into a browser so a few checks for Node.js can be elided.

`--omit-default-module-path`

Don't add WebAssembly fallback imports in generated JavaScript.

`--split-linked-modules`

Controls whether wasm-bindgen will split linked modules out into their own files. Enabling this is recommended, because it allows lazy-loading the linked modules and setting a stricter Content Security Policy.

wasm-bindgen uses the `new URL('…', import.meta.url)` syntax to resolve the links to such split out files. This breaks with most bundlers, since the bundler doesn't know to include the linked module in its output. That's why this option is disabled by default. Webpack 5 is an exception, which has special treatment for that syntax.

For other bundlers, you'll need to take extra steps to get it to work, likely by using a plugin. Alternatively, you can leave the syntax as is and instead manually configure the bundler to copy all files in `snippets/` to the output directory, preserving their paths relative to whichever bundled file ends up containing the JS shim.

On the no-modules target, `link_to!` won't work if used outside of a document, e.g. inside a worker. This is because it's impossible to figure out what the URL of the linked module is without a reference point like `import.meta.url`.

`--experimental-reset-state-function`

Generates and publicly exports a `__wbg_reset_state()` function that allows reinitializing the entire library state for environments that wish to reuse and reset the same JavaScript execution context without reloading the entire library.

This feature is currently only supported for `--target module`, `--target web`, and `--target nodejs`.

When this flag is enabled, the generated code will also associate all objects with execution instance identity that validates and throws for stale references, carefully associating Wasm bindgen pointer and finalization references with an instance id that is changed whenever this function is called.

Example usage:

```
import { __wbg_reset_state, inc } from './wbg-lib.js';

console.log(inc()); // logs 1
console.log(inc()); // logs 2

// fully reset the Wasm VM state
__wbg_reset_state();

console.log(inc()); // logs 1
```

Note: This feature adds overhead to the generated code and should only be enabled when needed for environment-specific requirements.

See also Handling Aborts for the `schedule_reinit()` API which automatically generates the reinit machinery without this flag.

Optimizing for Size with `wasm-bindgen`

The Rust and WebAssembly Working Group's Game of Life tutorial has an excellent section on shrinking Wasm code size, but there's a few `wasm-bindgen`-specific items to mention as well!

First and foremost, `wasm-bindgen` is designed to be lightweight and a "pay only for what you use" mentality. If you suspect that `wasm-bindgen` is bloating your program that is a bug and we'd like to know about it! Please feel free to file an issue, even if it's a question!

What to profile

With `wasm-bindgen` there's a few different files to be measuring the size of. The first of which is the output of the compiler itself, typically at `target/wasm32-unknown-unknown/release/foo.wasm`. **This file is not optimized for size and you should not measure it.** The output of the compiler when linking with `wasm-bindgen` is by design larger than it needs to be, the `wasm-bindgen` CLI tool will automatically strip all unneeded functionality out of the binary.

This leaves us with two primary generated files to measure the size of:

- **Generated wasm** - after running the `wasm-bindgen` CLI tool you'll get a file in `--out-dir` that looks like `foo_bg.wasm`. This file is the final fully-finished artifact from `wasm-bindgen`, and it reflects the size of the app you'll be publishing. All the optimizations mentioned in the code size tutorial will help reduce the size of this binary, so feel free to go crazy!
- **Generated JS** - the other file after running `wasm-bindgen` is a `foo.js` file which is what's actually imported by other JS code. This file is already generated to be as small as possible (not including unneeded functionality). The JS, however, is not uglified or minified,

but rather still human readable and debuggable. It's expected that you'll run an uglifier or bundler of the JS output to minimize it further in your application. If you spot a way we could reduce the output JS size further (or make it more amenable to bundler minification), please let us know!

Example

As an example, the `wasm-bindgen` repository contains an example about generating small Wasm binaries and shows off how to generate a small wasm file for adding two numbers.

Debug Information

Currently, debug information in the form of DWARF, is stripped away from the output module. To keep it, use `--keep-debug` with the CLI.

However, currently there are no known environments that support DWARF information with Wasm. You can follow the Debug C/C++ WebAssembly guide to get DWARF support in Chrome. This doesn't just demangle symbols in your stacktraces, but also allows for live debugging in the dev-tools or in external editors have a debugger bridge to Chrome.

The `wasm-bindgen-test-runner` currently generates DWARF debug information for tests by default.

Supported Rust Targets

Note: This section is about Rust target triples, not targets like node/web workers/browsers. More information on that coming soon!

The `wasm-bindgen` project is designed to target the `wasm32-unknown-unknown` target in Rust. This target is a "bare bones" target for Rust which emits WebAssembly as output. The standard library is largely inert as modules like `std::fs` and `std::net` will simply return errors.

Non-wasm targets

Note that `wasm-bindgen` also aims to compile on all targets. This means that it should be safe, if you like, to use `#[wasm_bindgen]` even when compiling for Windows (for example). For example:

```
#[wasm_bindgen]
pub fn add(a: u32, b: u32) -> u32 {
    a + b
}

#[cfg(not(target_arch = "wasm32"))]
fn main() {
    println!("1 + 2 = {}", add(1, 2));
}
```

This program will compile and work on all platforms, not just `wasm32-unknown-unknown`. Note that imported functions with `#[wasm_bindgen]` will unconditionally panic on non-wasm targets. For example:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
}

fn main() {
    log("hello!");
}
```

This program will unconditionally panic on all platforms other than `wasm32-unknown-unknown`.

For better compile times, however, you likely want to only use `#[wasm_bindgen]` on the `wasm32-unknown-unknown` target. You can have a target-specific dependency like so:

```
[target.'cfg(target_arch = "wasm32")'.dependencies]
wasm-bindgen = "0.2"
```

And in your code you can use:

```
#[cfg(target_arch = "wasm32")]
#[wasm_bindgen]
pub fn only_on_the_wasm_target() {
    // ...
}
```

Other Web Targets

The `wasm-bindgen` target does not support the `wasm32-unknown-emscripten` nor the `asmjs-unknown-emscripten` targets. There are currently no plans to support these targets either. All annotations work like other platforms on the targets, retaining exported functions and causing all imports to panic.

Supported Browsers

The output of `wasm-bindgen` includes a JS file, and as a result it's good to know what browsers that file is expected to be used in! By default the output uses ES modules with Wasm imports which isn't implemented in browsers today, but when using a bundler (like Webpack) or `--target web` you should be able to produce output suitable for all browsers.

Firefox, Chrome, Safari, and Edge browsers are all supported by `wasm-bindgen`. If you find a problem in one of these browsers please report it as we'd like to fix the bug! If you find a bug in another browser we would also like to be aware of it!

Caveats

- **IE 11 -** `wasm-bindgen` by default requires support for `WebAssembly`, but no version of IE currently supports `WebAssembly`. You can support IE by compiling Wasm files to JS using `wasm2js`. Note that at this time no bundler will do this by default, but we'd love to document plugins which do this if you are aware of one!

If you find other incompatibilities please report them to us! We'd love to either keep this list up-to-date or fix the underlying bugs :)

Support for Weak References

By default wasm-bindgen does use the TC39 weak references proposal if support is detected. At the time of this writing all major browsers do support it.

Without weak references your JS integration may be susceptible to memory leaks in Rust, for example:

- You could forget to call `.free()` on a JS object, leaving the Rust memory allocated.
- Rust closures converted to JS values (the `Closure` type) may not be executed and cleaned up.
- Rust closures have `Closure::{into_js_value,forget}` methods which explicitly do not free the underlying memory.

These issues are all solved with the weak references proposal in JS. `FinalizationRegistry` will ensure that all memory is cleaned up, regardless of whether it's explicitly deallocated or not. Note that explicit deallocation is always a possibility and supported, but if it's not called then memory will still be automatically deallocated if `FinalizationRegistry` support is detected.

Support for Reference Types

WebAssembly recently has gained support for a new value type called `externref`. Proposed in the [WebAssembly reference types repo] this feature of WebAssembly is hoped to enable more efficient communication between the host (JS) and the Wasm module. This feature removes the need for much of the JS glue generated by `wasm-bindgen` because it can natively call APIs with JS values.

For example, this Rust function:

```
#[wasm_bindgen]
pub fn takes_js_value(a: &JsValue) {
    // ...
}
```

generates this JS glue *without* reference types support:

```
const heap = new Array(32).fill(undefined);

heap.push(undefined, null, true, false);

let stack_pointer = 32;

function addBorrowedObject(obj) {
      if (stack_pointer == 1) throw new Error('out of js
stack');
    heap[--stack_pointer] = obj;
    return stack_pointer;
}

export function takes_js_value(a) {
    try {
        wasm.takes_js_value(addBorrowedObject(a));
    } finally {
        heap[stack_pointer++] = undefined;
```

```
    }
}
```

We can see here how under the hood the JS is managing a table of JS values which are passed to the Wasm binary, so Wasm actually only works in indices. If we compile with `-Ctarget-feature=+reference-types` (by default since Rust v1.82), however, the generated JS looks like:

```
export function takes_js_value(a) {
    wasm.takes_js_value(a);
}
```

And that's it! The WebAssembly binary takes the JS value directly and manages it internally.

Currently this feature is supported in Firefox 79+ and Chrome. Support in other browsers is likely coming soon! In Node.js this feature is behind the `--experimental-wasm-anyref` flag, although the support does not currently align with the upstream specification as of 14.6.0.

Catching Panics

By default, when a Rust function exported to JavaScript panics, Rust will abort and any allocated resources will be leaked. If you build with with `-Cpanic=unwind` and the `std` feature, Rust panics will be caught at the JavaScript boundary and converted into JavaScript exceptions, allowing proper cleanup and error handling.

When enabled, panics in exported Rust functions are caught and thrown as `PanicError` exceptions in JavaScript. For async functions, the returned promise is rejected with the `PanicError`.

Requirements

- **`panic=unwind`** - The panic strategy must be set to `unwind` (not `abort`)
- **`-Zbuild-std`** - Required to rebuild the standard library with the `unwind` panic strategy
- **Rust nightly compiler** - `-Zbuild-std` is only available on nightly
- **`std` feature** - `std` support is required to use `std::panic::catch_unwind`. to catch panics.

Building

Build your project with the required flags:

```
RUSTFLAGS="-Cpanic=unwind" cargo +nightly build --target wasm32-unknown-unknown -Zbuild-std=std,panic_unwind
```

Or set these in `.cargo/config.toml`:

```
[unstable]
build-std = ["std", "panic_unwind"]

[build]
target = "wasm32-unknown-unknown"
rustflags = ["-C", "panic=unwind"]

[profile.release]
panic = "unwind"
```

Then build with:

```
cargo +nightly build
```

How It Works

Synchronous Functions

When a synchronous exported function panics, the panic is caught and a `PanicError` is thrown to JavaScript:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn divide(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("division by zero");
    }
    a / b
}
```

```
import { divide } from './my_wasm_module.js';

try {
    divide(10, 0);
} catch (e) {
    console.log(e.name);    // "PanicError"
    console.log(e.message); // "division by zero"
}
```

Async Functions

For async functions, panics cause the returned promise to be rejected with a `PanicError`:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub async fn fetch_data(url: String) -> Result<JsValue, JsValue> {
    if url.is_empty() {
        panic!("URL cannot be empty");
```

```
    }
    // ... fetch implementation
    Ok(JsValue::NULL)
}
```

```
import { fetch_data } from './my_wasm_module.js';

try {
    await fetch_data("");
} catch (e) {
    console.log(e.name);    // "PanicError"
    console.log(e.message); // "URL cannot be empty"
}
```

Closures

When built with `panic=unwind`, all closure types catch panics by default and convert them to JavaScript `PanicError` exceptions.

Direct closures (`&dyn Fn` / `&mut dyn FnMut`)

Direct closure arguments (`&dyn Fn` and `&mut dyn FnMut`) are unwind safe by default. The `#[wasm_bindgen]` macro auto-injects a `MaybeUnwindSafe` bound, so the compiler will require callers to wrap non-unwind-safe captured values (e.g. `Cell<T>`, `&mut T`) in `std::panic::AssertUnwindSafe`. See Passing Rust Closures to JavaScript for details and examples.

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    fn forEach(cb: &mut dyn FnMut(u32));
}

forEach(&mut |x| {
    if x == 0 {
        panic!("zero not allowed!");
    }
});
```

`ScopedClosure` constructors

All default `ScopedClosure` constructors (`new`/`own`, `wrap`, `once`, `once_into_js`, `borrow`, `borrow_mut`) catch panics and require `UnwindSafe`. Each has two alternative suffixed variants:

- `*_assert_unwind_safe` — catches panics but skips the `UnwindSafe` check

- `*_aborting` — does not catch panics (aborts instead) and does not require `UnwindSafe`

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
extern "C" {
    fn setCallback(f: &Closure<dyn FnMut()>);
}

let closure = Closure::new(|| {
    panic!("closure panicked!");
});
setCallback(&closure);
```

```
try {
    triggerCallback();
} catch (e) {
    console.log(e.name);    // "PanicError"
    console.log(e.message); // "closure panicked!"
}
```

See Passing Rust Closures to JavaScript for more details on closure APIs and the `UnwindSafe` requirement.

The PanicError Class

When a panic occurs, a `PanicError` JavaScript exception is created with:

- `name` property set to `"PanicError"`
- `message` property containing the panic message (if the panic was created with a string message)

If the panic payload is not a `String` or `&str` (e.g., `panic_any(42)`), the message will be `"No panic message available"`.

Limitations

Nightly Only

This feature requires a nightly Rust compiler and will not work on stable Rust.

UnwindSafe Requirement

Exported function arguments and closure captures must satisfy Rust's `UnwindSafe` trait. For `&dyn Fn` and `&mut dyn FnMut` import arguments the macro enforces this via an auto-injected `MaybeUnwindSafe` bound. For captured values that are not `UnwindSafe` (such as `&mut T`, `Cell<T>`, or `RefCell<T>`), wrap them in `std::panic::AssertUnwindSafe` before the closure captures them:

```
let cell = std::cell::Cell::new(0u32);
let cell_ref = std::panic::AssertUnwindSafe(&cell);
takes_mut_closure(&mut move || { cell_ref.set(cell_ref.get() + 1); });
```

Mutable Slice Arguments

Functions with `&mut [T]` slice arguments cannot be used because mutable slices are not `UnwindSafe`. Consider using owned types like `Vec<T>` instead.

Hard Aborts

Some errors — `unreachable` instructions, stack overflow, or out-of-memory — are non-recoverable and cannot be caught by `catch_unwind`. When a hard abort occurs the Wasm instance is permanently poisoned and subsequent export calls will throw `"Module terminated"`.

wasm-bindgen provides abort handlers and a reinit mechanism for responding to these events and optionally recovering. See Handling Aborts for details on `set_on_abort`, `schedule_reinit()`, `set_on_reinit`, and host-initiated termination.

See Also

- `catch` attribute - For catching JavaScript exceptions in Rust
- `Result<T, E>` type - For explicit error handling between Rust and JavaScript

Handling Aborts

When built with `panic=unwind`, wasm-bindgen can catch Rust panics at the JavaScript boundary (see Catching Panics). However, there will still always be non-recoverable *hard aborts* — `unreachable` instructions, stack overflow, or out-of-memory — that cannot be caught by `catch_unwind`. When a hard abort occurs the Wasm instance is permanently poisoned: any subsequent export call will throw `"Module terminated"`.

The `wasm_bindgen::handler` module provides hooks for responding to these events and optionally recovering by reinitializing the module.

> **Note:** Hard abort detection and the abort handler API (`set_on_abort`) currently require `panic=unwind`. Support for `panic=abort` may be added in a future release. `schedule_reinit()` works with both `panic=unwind` and `panic=abort`.

Termination States

When using `panic=unwind`, the `(export "__instance_terminated" (global i32))` flag points to a boolean in linear memory:

Value	Meaning	Behavior on next export call
`0`	Live	Normal execution
`1`	Terminated	Abort hook fires (if not already called), then throws `"Module terminated"`

All aborts will set this `__instance_terminated` flag.

When the terminated state is set, reentrancy guards call the abort hook and ensure that the WebAssembly instance cannot again execute, immediately throwing if any new call is attempted instead.

Abort Handling

`handler::set_on_abort` registers a callback that fires when the instance has been aborted.

`set_on_abort` returns the previously registered handler (`None` if none was set), mirroring the `std::panic::set_hook` convention.

```
use std::sync::atomic::{AtomicBool, Ordering};
use wasm_bindgen::prelude::*;

static ABORTED: AtomicBool = AtomicBool::new(false);

fn on_abort() {
    ABORTED.store(true, Ordering::SeqCst);
}

#[wasm_bindgen(start)]
pub fn start() {
    wasm_bindgen::handler::set_on_abort(on_abort);
}
```

The abort handler is also invoked if the host terminates the instance by writing directly to the `__instance_terminated` flag from JavaScript. This means the handler fires on the next export call, giving it a chance to respond (including calling `schedule_reinit()` to trigger automatic recovery).

Reinitialization

`handler::schedule_reinit()` can be used to reinitialize the WebAssembly instance, while keeping the JS wrapper bindings in place, performing a transparent reinitialization of the library when state loss is acceptable. Works with both `panic=unwind` and `panic=abort` builds.

`schedule_reinit` may be called at any time. With `panic=unwind` it can also be called from within the abort hook to create a self-recovering abort handler.

When called, `schedule_reinit()` signals the module for reinitialization. The next call to any export detects this, creates a fresh `WebAssembly.Instance` from the same module, while keeping the same JS wrapper bindings in place but updated to the new instance.

When called during normal execution, the current call completes normally and the reset happens on the next export call — no abort hook fires.

When called from within the abort handler, the abort hook has already fired and the guard proceeds to reinitialize instead of throwing:

```
use wasm_bindgen::prelude::*;

fn on_abort() {
    // Schedule the module for reinitialization.
    wasm_bindgen::handler::schedule_reinit();
}

#[wasm_bindgen(start)]
pub fn start() {
    // Register the abort hook
    wasm_bindgen::handler::set_on_abort(on_abort);
}
```

With this setup, a hard abort (or host-initiated termination) transparently reinitializes the module on the next export call. The caller sees a fresh instance with all Rust statics reset to their initial values.

See Also

- Catching Panics — catching recoverable Rust panics as JavaScript exceptions

Supported Rust Types and their JavaScript Representations

This section provides an overview of all the types that `wasm-bindgen` can send and receive across the WebAssembly ABI boundary, and how they translate into JavaScript.

Many core JavaScript types support generic type parameters using type erasure, allowing you to work with typed collections like `Array<T>`, `Promise<T>`, `Map<K, V>`, and more.

Imported `extern Whatever;` JavaScript Types

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	Yes	No	Yes	Yes	Yes	Instances of the extern `Whatever` JavaScript class / prototype constructor

Imported JavaScript types support generic type parameters for working with typed JavaScript collections and APIs.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
#[derive(Copy, Clone, Debug)]
pub enum NumberEnum {
    Foo = 0,
    Bar = 1,
    Qux = 2,
}

#[wasm_bindgen]
#[derive(Copy, Clone, Debug)]
pub enum StringEnum {
    Foo = "foo",
    Bar = "bar",
    Qux = "qux",
}

#[wasm_bindgen]
pub struct Struct {
    pub number: NumberEnum,
    pub string: StringEnum,
}

#[wasm_bindgen]
extern "C" {
    pub type SomeJsType;
}

#[wasm_bindgen]
pub fn imported_type_by_value(x: SomeJsType) {
    /* ... */
}

#[wasm_bindgen]
pub fn imported_type_by_shared_ref(x: &SomeJsType) {
    /* ... */
}

#[wasm_bindgen]
pub fn return_imported_type() -> SomeJsType {
    unimplemented!()
}

#[wasm_bindgen]
pub fn take_option_imported_type(x: Option<SomeJsType>) {
    /* ... */
}
```

```
#[wasm_bindgen]
pub fn return_option_imported_type() -> Option<SomeJsType> {
    unimplemented!()
}
```

Example JavaScript Usage

```
import {
  imported_type_by_value,
  imported_type_by_shared_ref,
  return_imported_type,
  take_option_imported_type,
  return_option_imported_type,
} from './guide_supported_types_examples';

imported_type_by_value(new SomeJsType());
imported_type_by_shared_ref(new SomeJsType());

let x = return_imported_type();
console.log(x instanceof SomeJsType); // true

take_option_imported_type(null);
take_option_imported_type(undefined);
take_option_imported_type(new SomeJsType());

let y = return_option_imported_type();
if (y == null) {
  // ...
} else {
  console.log(y instanceof SomeJsType); // true
}
```

Exported `struct Whatever` Rust Types

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	Yes	Yes	Yes	Yes	Yes	Instances of a `wasm-bindgen`-generated JavaScript `class Whatever { ... }`

Note: Public fields implementing `Copy` have automatically generated getters/setters. To generate getters/setters for non-`Copy` public fields, use `#[wasm_bindgen(getter_with_clone)]` for the struct or implement getters/setters manually.

Exported functions can use generic JavaScript types with concrete type parameters (like `Promise<Number>`), but cannot have their own generic type parameters.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub struct ExportedNamedStruct {
    // pub value: String, // This won't work. See working example below.
    pub inner: u32,
}

#[wasm_bindgen(getter_with_clone)]
pub struct ExportedNamedStructNonCopy {
    pub non_copy_value: String,
    pub copy_value: u32,
}

#[wasm_bindgen]
pub fn named_struct_by_value(x: ExportedNamedStruct) {}

#[wasm_bindgen]
pub fn named_struct_by_shared_ref(x: &ExportedNamedStruct) {}

#[wasm_bindgen]
pub fn named_struct_by_exclusive_ref(x: &mut ExportedNamedStruct) {}

#[wasm_bindgen]
pub fn return_named_struct(inner: u32) -> ExportedNamedStruct {
    ExportedNamedStruct { inner }
}

#[wasm_bindgen]
pub fn named_struct_by_optional_value(x: Option<ExportedNamedStruct>) {}

#[wasm_bindgen]
pub fn return_optional_named_struct(inner: u32) -> Option<ExportedNamedStruct> {
    Some(ExportedNamedStruct { inner })
}

#[wasm_bindgen]
pub struct ExportedTupleStruct(pub u32, pub u32);

#[wasm_bindgen]
pub fn return_tuple_struct(x: u32, y: u32) -> ExportedTupleStruct {
    ExportedTupleStruct(x, y)
}
```

Example JavaScript Usage

```
import {
  ExportedNamedStruct,
  named_struct_by_value,
  named_struct_by_shared_ref,
  named_struct_by_exclusive_ref,
  return_named_struct,
  named_struct_by_optional_value,
  return_optional_named_struct,

  ExportedTupleStruct,
  return_tuple_struct
} from './guide_supported_types_examples';

let namedStruct = return_named_struct(42);
console.log(namedStruct instanceof ExportedNamedStruct); // true
console.log(namedStruct.inner); // 42

named_struct_by_shared_ref(namedStruct);
named_struct_by_exclusive_ref(namedStruct);
named_struct_by_value(namedStruct);

let optionalNamedStruct = return_optional_named_struct(42);
named_struct_by_optional_value(optionalNamedStruct);

let tupleStruct = return_tuple_struct(10, 20);
console.log(tupleStruct instanceof ExportedTupleStruct); // true
console.log(tupleStruct[0], tupleStruct[1]); // 10, 20
```

`JsValue`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	Yes	No	Yes	No	No	Any JavaScript value

`JsValue` is also the default representation for many erasable generic types at the ABI boundary.

`&[JsValue]` slices

You can pass `&[JsValue]` slices from Rust to JavaScript. The slice is passed as a JavaScript `Array`.

```
#[wasm_bindgen]
extern "C" {
    fn process_values(values: &[JsValue]);
}

pub fn call_js() {
    let values: Vec<JsValue> = vec![1.into(), "hello".into(), true.into()];
    process_values(&values);
}
```

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_js_value_by_value(x: JsValue) {}

#[wasm_bindgen]
pub fn take_js_value_by_shared_ref(x: &JsValue) {}

#[wasm_bindgen]
pub fn return_js_value() -> JsValue {
    JsValue::NULL
}
```

Example JavaScript Usage

```
import {
  take_js_value_by_value,
  take_js_value_by_shared_ref,
  return_js_value,
} from './guide_supported_types_examples';

take_js_value_by_value(42);
take_js_value_by_shared_ref('hello');

let v = return_js_value();
```

`js-sys`

The `js-sys` crate provides bindings to all JavaScript global APIs guaranteed to exist in every JavaScript environment by the ECMAScript standard.

Wasm-bindgen supports generic typing using type erasure for imported JavaScript types. Generic parameters exist only in Rust code—they are completely erased in the generated JavaScript bindings.

For a conceptual overview and usage guide, see Working with Generics.

> When passing a typed value (e.g., `Array<Number>`) to a function expecting a wider type (e.g., `&Array<JsValue>`), use the `upcast()` method: `my_array.upcast()`. See Upcasting Types for details.

All generic types listed implement `JsGeneric` and default to `JsValue` when type parameters are unspecified.

Type	Description
`Array<T>`	Typed JavaScript array
`ArrayTuple<T1, ..., T8>`	Array with per-item typing (up to 8)
`Function<fn(A) -> R>`	Typed function with return and arguments
`Promise<T>`	Promise resolving to `T`
`Map<K, V>`	Typed map
`Set<T>`	Typed set
`Iterator<T>` / `AsyncIterator<T>`	Typed iterators
`Generator<T>` / `AsyncGenerator<T>`	Typed generators
`Object<T>`	Object with typed values

Type	Description
`WeakMap<K, V>` / `WeakSet<T>` / `WeakRef<T>`	Weak collections
`JsOption<T>`	Nullable JS value (`T`, `null`, or `undefined`)
`Number`	JavaScript number primitive
`JsString`	JavaScript string primitive
`Boolean`	JavaScript boolean primitive
`BigInt`	JavaScript BigInt primitive
`Symbol`	JavaScript Symbol primitive
`Undefined`	JavaScript `undefined` value
`Null`	JavaScript `null` value

Array<T>

Typed JavaScript `Array` builtin.

```
use js_sys::{Array, Number, JsString};

let numbers: Array<Number> = Array::new_typed();
numbers.push(&Number::from(42));

let first: Number = numbers.get(0);

for num in numbers.iter() {
    // num is Number
}
```

ArrayTuple<T1, ..., T8>

JavaScript `Array` with per-item typing, supporting up to 8 items.

```
use js_sys::{ArrayTuple, Number, JsString, Boolean};

let tuple: ArrayTuple<Number, JsString, Boolean> = ArrayTuple::new();
tuple.set0(&Number::from(1));
tuple.set1(&JsString::from("hello"));
tuple.set2(&Boolean::from(true));

let n: Number = tuple.get0();
let s: JsString = tuple.get1();
let b: Boolean = tuple.get2();
```

Function<fn(A) -> R>

Typed JavaScript function with return type and up to 8 arguments.

```
use js_sys::{Function, Number, JsString};

#[wasm_bindgen]
extern "C" {
    fn getFormatter() -> Function<fn(Number) -> JsString>;
}

let formatter = getFormatter();
let result: JsString = formatter.call1(&JsValue::UNDEFINED,
&Number::from(42))?;
```

Type aliases

Alias	Description
`TypedFunction`	Strict arity checking (arguments default to `None`)
`AnyFunction`	Lenient behavior (arguments default to `JsValue`)
`VoidFunction`	`Function<fn(...) -> Undefined>` with strict arity

Calling with tuples

```
let f: Function<fn(JsString, Boolean) -> Number> =
get_function();

// call accepts a tuple of references
let result = f.call(&context, ())?;                    // no
args
let result = f.call(&context, (&my_string,))?;             //
one arg
let result = f.call(&context, (&my_string, &my_bool))?;   //
```

```
two args

// bindn returns a new function with arguments pre-bound
let bound: Function<fn(Boolean) -> Number> = f.bindn(&context,
(&my_string,));

// Argument upcasting: pass subtypes where supertypes are
expected
let f: Function<fn(JsValue, JsValue) -> Number> =
get_function();
let result = f.call(&context, (&my_number, &my_string))?; //
Number, JsString upcast to JsValue
```

Converting from Closures

Use `Function::from_closure()` to convert owned `Closure` types to typed `Function`, and `Function::closure_ref()` for borrowed closures:

```
use js_sys::{Function, Number, JsString};
use wasm_bindgen::prelude::*;

// Owned closure to Function (transfers ownership to JS)
let closure: Closure<dyn FnMut(Number) -> JsString> =
Closure::new(|n: Number| {
    JsString::from(format!("Value: {}", n.value_of()))
});
let func: Function<fn(Number) -> JsString> =
Function::from_closure(closure);
```

For borrowed closures, use `closure_ref`:

```
let mut count = 0u32;
let mut increment = || {
    count += 1;
    Number::from(count)
};
let closure = ScopedClosure::borrow_mut(&mut increment);
```

```
let func: &Function<fn() -> Number> = Function::closure_ref(&closure);
```

Primitive type coercion

Rust primitives automatically coerce to JS types in closure return positions:

```
// u32 coerces to Number
let closure: Closure<dyn Fn() -> u32> = Closure::new(|| 42);
let func: Function<fn() -> Number> = Function::from_closure(closure);

// String coerces to JsString
let closure: Closure<dyn Fn() -> String> = Closure::new(|| "hello".to_string());
let func: Function<fn() -> JsString> = Function::from_closure(closure);
```

This works for all numeric primitives (`u8`, `i32`, `f64`, etc.) and string types (`String`, `&str`, `char`).

See Passing Rust Closures to JS for more details on closure types and lifecycle management.

Promise<T>

Typed JavaScript `Promise` that resolves to `T`.

`Promise<T>` implements `IntoFuture`, so it can be `.await`ed directly in any `async` function. The resolved value has type `T` without any cast:

```
use js_sys::{Promise, Number};

#[wasm_bindgen]
extern "C" {
    fn fetchCount() -> Promise<Number>;
}

async fn example() -> Result<Number, JsValue> {
    let count: Number = fetchCount().await?;
    Ok(count)
}
```

This requires the `futures` feature on `js-sys`, which is activated automatically when `wasm-bindgen-futures` is a dependency. Enable it directly if needed:

```
[dependencies]
js-sys = { version = "0.3", features = ["futures"] }
```

If you need a named `JsFuture` value, use `JsFuture::from`:

```
use js_sys::futures::JsFuture;

let future: JsFuture<Number> = JsFuture::from(fetchCount());
```

The Promising trait

The `Promising` trait represents types that are either `T` or `Promise<T>`. Use it to accept both immediate values and promises in function signatures:

```
use js_sys::{Promising, Promise, Number};

#[wasm_bindgen]
```

```
extern "C" {
    // Accepts either Number or Promise<Number>
    fn getValue<T: Promising<Resolution = Number>>(value: T) -
> Promise<Number>;
}
```

Map<K, V>

Typed JavaScript `Map` builtin.

```
use js_sys::{Map, JsString, Number};

let map: Map<JsString, Number> = Map::new_typed();
map.set(&JsString::from("key"), &Number::from(100));

let value: Number = map.get(&JsString::from("key"));
```

Set<T>

Typed JavaScript `Set` builtin.

```
use js_sys::{Set, JsString};

let set: Set<JsString> = Set::new_typed();
set.add(&JsString::from("value"));

let has: bool = set.has(&JsString::from("value"));
```

Iterator<T> and AsyncIterator<T>

Typed iterators for synchronous and asynchronous iteration.

```
use js_sys::{Iterator, Number};

#[wasm_bindgen]
extern "C" {
    fn getNumberIterator() -> Iterator<Number>;
}

let iter = getNumberIterator();
while let Some(num) = iter.next() {
    // num is Number
}
```

The Iterable and AsyncIterable traits

The `Iterable` and `AsyncIterable` traits represent objects implementing the iterator protocol via `[Symbol.iterator]()` or `[Symbol.asyncIterator]()`. Use them to accept any iterable type in function signatures:

```
use js_sys::{Iterable, AsyncIterable, Number};

#[wasm_bindgen]
extern "C" {
    // Accepts Array, Set, Generator, or any other iterable
    fn processIterable<T: Iterable<Item = Number>>(items: T);

    // Accepts AsyncGenerator or any other async iterable
     fn processAsyncIterable<T: AsyncIterable<Item = Number>>
(items: T);
}
```

Generator<T> and AsyncGenerator<T>

Typed generator builtins.

```
use js_sys::{Generator, Number};

#[wasm_bindgen]
extern "C" {
    fn createGenerator() -> Generator<Number>;
}

let gen = createGenerator();
let result = gen.next();
```

Object<T>

Typed object records where values are of type `T`.

```
use js_sys::{Object, Number};

let obj: Object<Number> = Object::new_typed();
```

WeakMap<K, V>, WeakSet<T>, WeakRef<T>

Typed weak collection builtins.

```
use js_sys::{WeakMap, WeakSet, WeakRef, Object, Number};

let weak_map: WeakMap<Object, Number> = WeakMap::new_typed();
let weak_set: WeakSet<Object> = WeakSet::new_typed();
let weak_ref: WeakRef<Object> = WeakRef::new(&my_object);
```

JsOption<T>

Represents a JS value that may be `T`, `null`, or `undefined`.

```
use wasm_bindgen::JsOption;
use js_sys::Number;

#[wasm_bindgen]
extern "C" {
    fn maybeGetValue() -> JsOption<Number>;
}

let value = maybeGetValue();

// Check emptiness
if value.is_empty() {
    // null or undefined
}

// Convert to Option
match value.into_option() {
    Some(num) => { /* use num */ }
    None => { /* handle null */ }
}

// Unwrap methods
let num = value.unwrap();
let num = value.expect("should exist");
let num = value.unwrap_or_default();
let num = value.unwrap_or_else(|| Number::from(0));

// Create values
let with_value = JsOption::wrap(Number::from(42));
let empty: JsOption<Number> = JsOption::new();
let from_opt = JsOption::from_option(Some(Number::from(42)));
```

`JsOption<T>` vs `Option<T>`:

Type	Behavior
`Option<T>`	Undefined or Null check at ABI boundary, immediately converts to `Some(T)` or `None`
`JsOption<T>`	Defers undefined or null check, works in `JsGeneric` positions

`Box<[T]>` and `Vec<T>`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	No	No	Yes	Yes	Yes	A JavaScript `Array` object

You can pass boxed slices and `Vec`s of several different types to and from JS:

- `JsValue`s.
- Imported JavaScript types.
- Exported Rust types.
- `String`s.

You can also pass boxed slices of numbers to JS, except that they're converted to typed arrays (`Uint8Array`, `Int32Array`, etc.) instead of regular arrays.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_boxed_js_value_slice_by_value(x: Box<[JsValue]>) {}

#[wasm_bindgen]
pub fn return_boxed_js_value_slice() -> Box<[JsValue]> {
    vec![JsValue::NULL, JsValue::UNDEFINED].into_boxed_slice()
}

#[wasm_bindgen]
pub fn take_option_boxed_js_value_slice(x: Option<Box<[JsValue]>>) {}

#[wasm_bindgen]
pub fn return_option_boxed_js_value_slice() -> Option<Box<[JsValue]>> {
    None
}
```

Example JavaScript Usage

```
import {
  take_boxed_js_value_slice_by_value,
  return_boxed_js_value_slice,
  take_option_boxed_js_value_slice,
  return_option_boxed_js_value_slice,
} from './guide_supported_types_examples';

take_boxed_js_value_slice_by_value([null, true, 2, {}, []]);

let values = return_boxed_js_value_slice();
console.log(values instanceof Array); // true

take_option_boxed_js_value_slice(null);
take_option_boxed_js_value_slice(undefined);
take_option_boxed_js_value_slice([1, 2, 3]);

let maybeValues = return_option_boxed_js_value_slice();
if (maybeValues == null) {
  // ...
} else {
  console.log(maybeValues instanceof Array); // true
}
```

`*const T` and `*mut T`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	No	No	Yes	Yes	Yes	A JavaScript number value

Example Rust Usage

```
use std::ptr;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_pointer_by_value(x: *mut u8) {}

#[wasm_bindgen]
pub fn return_pointer() -> *mut u8 {
    ptr::null_mut()
}
```

Example JavaScript Usage

```
import {
  take_pointer_by_value,
  return_pointer,
} from './guide_supported_types_examples';
import { memory } from './guide_supported_types_examples_bg';

let ptr = return_pointer();
let buf = new Uint8Array(memory.buffer);
let value = buf[ptr];
console.log(`The byte at the ${ptr} address is ${value}`);

take_pointer_by_value(ptr);
```

`NonNull<T>`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	No	No	Yes	Yes	Yes	A JavaScript number value

Example Rust Usage

```
use std::ptr;
use std::ptr::NonNull;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub unsafe fn take_pointer_by_value(x: Option<NonNull<u8>>) {
    Box::from_raw(x.unwrap().as_ptr());
}

#[wasm_bindgen]
pub fn return_pointer() -> Option<NonNull<u8>> {
    Some(NonNull::from(Box::leak(Box::new(42))))
}
```

Example JavaScript Usage

```
import {
  take_pointer_by_value,
  return_pointer,
} from './guide_supported_types_examples';
import { memory } from './guide_supported_types_examples_bg';

let ptr = return_pointer();
let buf = new Uint8Array(memory.buffer);
let value = buf[ptr];
console.log(`The byte at the ${ptr} address is ${value}`);

take_pointer_by_value(ptr);
```

Numbers: `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `u128`, `i128`, `isize`, `usize`, `f32`, and `f64`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	No	No	Yes	Yes	Yes	A JavaScript number or bigint value

JavaScript `Number`s are 64-bit floating point value under the hood and cannot accurately represent all of Rust's numeric types. `wasm-bindgen` will automatically use either `BigInt` or `Number` to accurately represent Rust's numeric types in JavaScript:

- `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `isize`, `usize`, `f32`, and `f64` will be represented as `Number` in JavaScript.
- `u64`, `i64`, `u128`, and `i128` will be represented as `BigInt` in JavaScript.

Note: Wasm is currently a 32-bit architecture, so `isize` and `usize` are 32-bit integers and "fit" into a JavaScript `Number`.

Note: `u128` and `i128` require `wasm-bindgen` version 0.2.96 or later.

Converting from JavaScript to Rust

`wasm-bindgen` will automatically handle the conversion of JavaScript numbers to Rust numeric types. The conversion rules are as follows:

`Number` to `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `isize`, and `usize`

If the JavaScript number is `Infinity`, `-Infinity`, or `NaN`, then the Rust value will be 0. Otherwise, the JavaScript number will rounded towards zero (see `Math.trunc` or `f64::trunc`). If the rounded number is too large or too small for the target integer type, it will wrap around.

For example, if the target type is `i8`, Rust will see the following values for the following inputs:

JS input number	Rust value (`i8`)
42	42
-42	-42
1.999	1
-1.999	-1
127	127
128	-128
255	-1
256	0
-0	0
`±Infinity`	0
`NaN`	0

This is the same behavior as assigning the JavaScript `Number` to a typed array of the appropriate integer type in JavaScript, i.e. `new Uint8Array([value])[0]`.

Except for the handling of `Infinity` and `-Infinity`, this is the same behavior as casting `f64` to the appropriate integer type in Rust, i.e. `value_f64 as u32`.

`BigInt` to `u64`, `i64`, `u128`, and `i128`

If the JavaScript `BigInt` is too large or too small for the target integer type, it will wrap around.

This is the same behavior as assigning the JavaScript `BigInt` to a typed array for 64-bit integer types in JavaScript, i.e. `new Int64Array([value])[0]`.

`Number` to `f32`

The JavaScript `Number` is converted to a Rust `f32` using the same rules as casting `f64` to `f32` in Rust, i.e. `value_f64 as f32`.

This is the same behavior as `Math.fround` or assigning the JavaScript `Number` to a `Float32Array` in JavaScript, i.e. `new Float32Array([value])[0]`.

`Number` to `f64`

Since JavaScript numbers are 64-bit floating point values, converting a JavaScript `Number` to a Rust `f64` is a no-op.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_number_by_value(x: u32) {}

#[wasm_bindgen]
pub fn return_number() -> f64 {
    42.0
}

#[wasm_bindgen]
pub fn take_option_number(x: Option<u8>) {}

#[wasm_bindgen]
pub fn return_option_number() -> Option<i16> {
    Some(-300)
}
```

Example JavaScript Usage

```
import {
  take_number_by_value,
  return_number,
  take_option_number,
  return_option_number,
} from './guide_supported_types_examples';

take_number_by_value(42);

let x = return_number();
console.log(typeof x); // "number"

take_option_number(null);
take_option_number(undefined);
take_option_number(13);

let y = return_option_number();
if (y == null) {
  // ...
} else {
  console.log(typeof y); // "number"
}
```

`bool`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	No	No	Yes	Yes	Yes	A JavaScript boolean value

Note: Only JavaScript `Boolean` values (`true` or `false`) are supported when calling into Rust. If you want to pass truthy or falsy values to Rust, convert them to a boolean using `Boolean(value)` first.

If you are using TypeScript, you don't have to worry about this, as TypeScript will emit a compiler error if you try to pass a non-`boolean` value.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_bool_by_value(x: bool) {}

#[wasm_bindgen]
pub fn return_bool() -> bool {
    true
}

#[wasm_bindgen]
pub fn take_option_bool(x: Option<bool>) {}

#[wasm_bindgen]
pub fn return_option_bool() -> Option<bool> {
    Some(false)
}
```

Example JavaScript Usage

```
import {
  take_char_by_value,
  return_char,
  take_option_bool,
  return_option_bool,
} from './guide_supported_types_examples';

take_bool_by_value(true);

let b = return_bool();
console.log(typeof b); // "boolean"

take_option_bool(null);
take_option_bool(undefined);
take_option_bool(true);

let c = return_option_bool();
if (c == null) {
  // ...
} else {
  console.log(typeof c); // "boolean"
}
```

`char`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	No	No	Yes	Yes	Yes	A JavaScript string value

Since JavaScript doesn't have a character type, `char` is represented as a JavaScript string with one Unicode code point.

> **Note**: JavaScript strings uses UTF-16 encoding. This means that a single `char` may be represented by a string of length 1 or 2 in JavaScript, depending on the Unicode code point. See `String.fromCodePoint` for more information.

When passed into Rust, the `char` value of a JavaScript string is determined using `codePointAt(0)`. If the JavaScript string is empty or starts with an unpaired surrogate, a runtime error will be thrown.

> **Note**: For more information about unpaired surrogates, see the documentation for `str`.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_char_by_value(x: char) {}

#[wasm_bindgen]
pub fn return_char() -> char {
    '🚀'
}
```

Example JavaScript Usage

```
import {
  take_char_by_value,
  return_char,
} from './guide_supported_types_examples';

take_char_by_value('a');

let c = return_char();
console.log(typeof c); // "string"
```

`str`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
No	Yes	No	No	No	No	JavaScript string value

Copies the string's contents back and forth between the JavaScript garbage-collected heap and the Wasm linear memory with `TextDecoder` and `TextEncoder`. If you don't want to perform this copy, and would rather work with handles to JavaScript string values, use the `js_sys::JsString` type.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_str_by_shared_ref(x: &str) {}
```

Example JavaScript Usage

```
import {
  take_str_by_shared_ref,
} from './guide_supported_types_examples';

take_str_by_shared_ref('hello');
```

UTF-16 vs UTF-8

Strings in JavaScript are encoded as UTF-16, but with one major exception: they can contain unpaired surrogates. For some Unicode characters UTF-16 uses two 16-bit values. These are called "surrogate pairs" because they always come in pairs. In JavaScript, it is possible for these surrogate pairs to be missing the other half, creating an "unpaired surrogate".

When passing a string from JavaScript to Rust, it uses the `TextEncoder` API to convert from UTF-16 to UTF-8. This is normally perfectly fine... unless there are unpaired surrogates. In that case it will replace the unpaired surrogates with U+FFFD (`&#xFFFD;`, the replacement character). That means the string in Rust is now different from the string in JavaScript!

If you want to guarantee that the Rust string is the same as the JavaScript string, you should instead use `js_sys::JsString` (which keeps the string in JavaScript and doesn't copy it into Rust).

If you want to access the raw value of a JS string, you can use `JsString::iter`, which returns an `Iterator<Item = u16>`. This perfectly preserves everything (including unpaired surrogates), but it does not do any encoding (so you have to do that yourself!).

If you simply want to ignore strings which contain unpaired surrogates, you can use `JsString::is_valid_utf16` to test whether the string contains unpaired surrogates or not.

`String`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	No	No	Yes	Yes	Yes	JavaScript string value

Copies the string's contents back and forth between the JavaScript garbage-collected heap and the Wasm linear memory with `TextDecoder` and `TextEncoder`

> **Note**: Be sure to check out the documentation for `str` to learn about some caveats when working with strings between JS and Rust.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_string_by_value(x: String) {}

#[wasm_bindgen]
pub fn return_string() -> String {
    "hello".into()
}

#[wasm_bindgen]
pub fn take_option_string(x: Option<String>) {}

#[wasm_bindgen]
pub fn return_option_string() -> Option<String> {
    None
}
```

Example JavaScript Usage

```
import {
  take_string_by_value,
  return_string,
  take_option_string,
  return_option_string,
} from './guide_supported_types_examples';

take_string_by_value('hello');

let s = return_string();
console.log(typeof s); // "string"

take_option_string(null);
take_option_string(undefined);
take_option_string('hello');

let t = return_option_string();
if (t == null) {
  // ...
} else {
  console.log(typeof s); // "string"
}
```

Number Slices: `[u8]`, `[i8]`, `[u16]`, `[i16]`, `[u32]`, `[i32]`, `[u64]`, `[i64]`, `[f32]`, `[f64]`, `[MaybeUninit<u8>]`, `[MaybeUninit<i8>]`, `[MaybeUninit<u16>]`, `[MaybeUninit<i16>]`, `[MaybeUninit<u32>]`, `[MaybeUninit<i32>]`, `[MaybeUninit<u64>]`, `[MaybeUninit<i64>]`, `[MaybeUninit<f32>]`, and `[MaybeUninit<f64>]`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<&T>` parameter	`Option<T>` return value	JavaScript representation
No	Yes	Yes	No	No	No	A JavaScript `TypedArray` view of the Wasm memory for the boxed slice of the appropriate type (`Int32Array`, `Uint8Array`, etc)

Note: Numeric `MaybeUninit<T>` can always be assumed to be initialized upon transmission from Rust to JS and vice-versa. However, uninitialized values coming from Rust might contain unspecified values.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_number_slice_by_shared_ref(x: &[f64]) {}

#[wasm_bindgen]
pub fn take_number_slice_by_exclusive_ref(x: &mut [u8]) {}
```

Example JavaScript Usage

```
import {
  take_number_slice_by_shared_ref,
  take_number_slice_by_exclusive_ref,
} from './guide_supported_types_examples';

take_number_slice_by_shared_ref(new Float64Array(100));
take_number_slice_by_exclusive_ref(new Uint8Array(100));
```

Boxed Number Slices: `Box<[u8]>`, `Box<[i8]>`, `Box<[u16]>`, `Box<[i16]>`, `Box<[u32]>`, `Box<[i32]>`, `Box<[u64]>`, `Box<[i64]>`, `Box<[f32]>`, `Box<[f64]>`, `Box<[MaybeUninit<u8>]>`, `Box<[MaybeUninit<i8>]>`, `Box<[MaybeUninit<u16>]>`, `Box<[MaybeUninit<i16>]>`, `Box<[MaybeUninit<u32>]>`, `Box<[MaybeUninit<i32>]>`, `Box<[MaybeUninit<u64>]>`, `Box<[MaybeUninit<i64>]>`, `Box<[MaybeUninit<f32>]>`, and `Box<[MaybeUninit<f64>]>`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
Yes	No	No	Yes	Yes	Yes	A JavaScript `TypedArray` of the appropriate type (`Int32Array`, `Uint8Array`, etc...)

Note: The contents of the slice are copied into a JavaScript `TypedArray` from the Wasm linear memory when returning a boxed slice to JavaScript, and vice versa when receiving a JavaScript `TypedArray` as a boxed slice in Rust.

Note: Numeric `MaybeUninit<T>` can always be assumed to be initialized upon transmission from Rust to JS and vice-versa. However, uninitialized values coming from Rust might contain unspecified values.

Example Rust Usage

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn take_boxed_number_slice_by_value(x: Box<[f64]>) {}

#[wasm_bindgen]
pub fn return_boxed_number_slice() -> Box<[u32]> {
    (0..42).collect::<Vec<u32>>().into_boxed_slice()
}

#[wasm_bindgen]
pub fn take_option_boxed_number_slice(x: Option<Box<[u8]>>) {}

#[wasm_bindgen]
pub fn return_option_boxed_number_slice() -> Option<Box<[i32]>> {
    None
}
```

Example JavaScript Usage

```
import {
  take_boxed_number_slice_by_value,
  return_boxed_number_slice,
  take_option_boxed_number_slice,
  return_option_boxed_number_slice,
} from './guide_supported_types_examples';

take_boxed_number_slice_by_value(new Uint8Array(100));

let x = return_boxed_number_slice();
console.log(x instanceof Uint32Array); // true

take_option_boxed_number_slice(null);
take_option_boxed_number_slice(undefined);
take_option_boxed_number_slice(new Int16Array(256));

let y = return_option_boxed_number_slice();
if (y == null) {
  // ...
} else {
  console.log(x instanceof Int32Array); // true
}
```

`Result<T, E>`

`T` parameter	`&T` parameter	`&mut T` parameter	`T` return value	`Option<T>` parameter	`Option<T>` return value	JavaScript representation
No	No	No	Yes	No	No	Same as `T`, or an exception

The `Result` type can be returned from functions exported to JS as well as closures in Rust. The `Ok` type must be able to be converted to JS, and the `Err` type must implement `Into<JsValue>`. Whenever `Ok(val)` is encountered it's converted to JS and handed off, and whenever `Err(error)` is encountered an exception is thrown in JS with `error`.

You can use `Result` to enable handling of JS exceptions with `?` in Rust, naturally propagating it upwards to the Wasm boundary. Furthermore you can also return custom types in Rust so long as they're all convertible to `JsValue`.

Note that if you import a JS function with `Result` you need `#[wasm_bindgen(catch)]` to be annotated on the import (unlike exported functions, which require no extra annotation). This may not be necessary in the future though and it may work "as is"!.

`#[wasm_bindgen]` Attributes

The `#[wasm_bindgen]` macro supports a good amount of configuration for controlling precisely how exports are exported, how imports are imported, and what the generated JavaScript glue ends up looking like. This section is an exhaustive reference of the possibilities!

`#[wasm_bindgen]` on JavaScript Imports

This section enumerates the attributes available for customizing bindings for JavaScript functions and classes imported into Rust within an `extern "C" { ... }` block.

`catch`

The `catch` attribute allows catching a JavaScript exception. This can be attached to any imported function or method, and the function must return a `Result` where the `Err` payload is a `JsValue`:

```
#[wasm_bindgen]
extern "C" {
    // `catch` on a standalone function.
    #[wasm_bindgen(catch)]
    fn foo() -> Result<(), JsValue>;

    // `catch` on a method.
    type Zoidberg;
    #[wasm_bindgen(catch, method)]
        fn  woop_woop_woop(this:  &Zoidberg)  ->  Result<u32,
JsValue>;
}
```

If calling the imported function throws an exception, then `Err` will be returned with the exception that was raised. Otherwise, `Ok` is returned with the result of the function.

> By default `wasm-bindgen` will take no action when Wasm calls a JS function which ends up throwing an exception. The Wasm spec right now doesn't support stack unwinding and as a result Rust code **will not execute destructors**. This can unfortunately cause memory leaks in Rust right now.
>
> This limitation is entirely avoided when building with `-Cpanic=unwind` and the `std` feature enabled. Unexpected JS exceptions that would otherwise cause issues will result in a proper unwind, with the JS exception propagated to the caller and destructors running correctly.

constructor

The `constructor` attribute is used to indicate that the function being bound should actually translate to calling the `new` operator in JavaScript. The final argument must be a type that's imported from JavaScript, and it's what will get used in the generated glue:

```
#[wasm_bindgen]
extern "C" {
    type Shoes;

    #[wasm_bindgen(constructor)]
    fn new() -> Shoes;
}
```

This will attach a `new` static method to the `Shoes` type, and in JavaScript when this method is called, it will be equivalent to `new Shoes()`.

```
// Become a cobbler; construct `new Shoes()`
let shoes = Shoes::new();
```

extends = Class

The `extends` attribute can be used to say that an imported type extends (in the JS class hierarchy sense) another type. This will generate `AsRef`, `From`, `Deref`, and `Upcast` impls for converting a type into another given that we statically know the inheritance hierarchy:

```
#[wasm_bindgen]
extern "C" {
    type Foo;

    #[wasm_bindgen(extends = Foo)]
    type Bar;
}

let x: &Bar = ...;
let y: &Foo = x.as_ref(); // zero cost cast via AsRef
let z: Foo = x.clone().upcast(); // zero cost cast via Upcast
```

The trait implementations generated for the above block are:

```
impl From<Bar> for Foo { ... }
impl AsRef<Foo> for Bar { ... }
impl Deref for Bar { type Target = Foo; ... }
impl Upcast<Foo> for Bar { }
```

The first `extends` target is used as the `Deref` target. To disable `Deref` generation, use the `no_deref` attribute. To disable `Upcast` generation, use the `no_upcast` attribute.

The `extends = ...` attribute can be specified multiple times for longer inheritance chains, and `AsRef`, `Upcast`, etc. impls will be generated for each of the types.

```
#[wasm_bindgen]
extern "C" {
    type Foo;
```

```
    #[wasm_bindgen(extends = Foo)]
    type Bar;

    #[wasm_bindgen(extends = Foo, extends = Bar)]
    type Baz;
}

let x: &Baz = ...;
let y1: &Bar = x.as_ref();
let y2: &Foo = y1.as_ref();

// Or using upcast:
let baz: Baz = ...;
let bar: Bar = baz.clone().upcast_into();
let foo: &Foo = baz.upcast();
```

The `upcast()` method is also used when working with generic types. For example, when passing an `Array<Number>` to a function expecting `&Array<JsValue>`, use `my_array.upcast()`. See Working with Generics for more details.

`getter` and `setter`

These two attributes can be combined with `method` to indicate that this is a getter or setter method. A `getter`-tagged function by default accesses the JavaScript property with the same name as the getter function. A `setter`'s function name is currently required to start with `set_` and the property it accesses is the suffix after `set\_`.

Consider the following JavaScript class that has a getter and setter for the `white_russians` property:

```
class TheDude {
  get white_russians() {
    ...
  }
  set white_russians(val) {
    ...
  }
}
```

We would import this with the following `#[wasm_bindgen]` attributes:

```
#[wasm_bindgen]
extern "C" {
    type TheDude;

    #[wasm_bindgen(method, getter)]
    fn white_russians(this: &TheDude) -> u32;

    #[wasm_bindgen(method, setter)]
    fn set_white_russians(this: &TheDude, val: u32);
}
```

Here we're importing the `TheDude` type and defining the ability to access each object's `white_russians` property. The first function here is a getter and will be available in Rust as `the_dude.white_russians()`, and the latter is the setter which is accessible as

`the_dude.set_white_russians(2)`. Note that both functions have a `this` argument as they're tagged with `method`.

Finally, you can also pass an argument to the `getter` and `setter` properties to configure what property is accessed. When the property is explicitly specified then there is no restriction on the method name. For example the below is equivalent to the above:

```
#[wasm_bindgen]
extern "C" {
    type TheDude;

    #[wasm_bindgen(method, getter = white_russians)]
    fn my_custom_getter_name(this: &TheDude) -> u32;

    #[wasm_bindgen(method, setter = white_russians)]
    fn my_custom_setter_name(this: &TheDude, val: u32);
}
```

Heads up! `getter` and `setter` functions are found on the constructor's prototype chain once at load time, cached, and then the cached accessor is invoked on each access. If you need to dynamically walk the prototype chain on every access, add the `structural` attribute!

```
// This is the default function Rust will invoke on
`the_dude.white_russians()`:
const white_russians = Object.getOwnPropertyDescriptor(
  TheDude.prototype,
  "white_russians"
).get;

// This is what you get by adding `structural`:
const white_russians = function(the_dude) {
  return the_dude.white_russians;
};
```

`final`

The `final` attribute is the converse of the `structural` attribute. It configures how `wasm-bindgen` will generate JS imports to call the imported function. Notably a function imported by `final` never changes after it was imported, whereas a function imported by default (or with `structural`) is subject to runtime lookup rules such as walking the prototype chain of an object. Note that `final` is not suitable for accessing data descriptor properties of JS objects; to accomplish this, use the `structural` attribute.

The `final` attribute is intended to be purely related to performance. It ideally has no user-visible effect, and `structural` imports (the default) should be able to transparently switch to `final` eventually.

The eventual performance aspect is that with the component model proposal then `wasm-bindgen` will need to generate far fewer JS function shims to import than it does today. For example, consider this import today:

```
#[wasm_bindgen]
extern "C" {
    type Foo;
    #[wasm_bindgen(method)]
    fn bar(this: &Foo, argument: &str) -> JsValue;
}
```

Without the `final` attribute the generated JS looks like this:

```
// without `final`
export function __wbg_bar_a81456386e6b526f(arg0, arg1, arg2) {
    let varg1 = getStringFromWasm(arg1, arg2);
    return addHeapObject(getObject(arg0).bar(varg1));
}
```

We can see here that this JS function shim is required, but it's all relatively self-contained. It does, however, execute the `bar` method in a duck-type-y fashion in the sense that it never validates `getObject(arg0)` is of type `Foo` to actually call the `Foo.prototype.bar` method.

If we instead, however, write this:

```
#[wasm_bindgen]
extern "C" {
    type Foo;
    #[wasm_bindgen(method, final)] // note the change here
    fn bar(this: &Foo, argument: &str) -> JsValue;
}
```

it generates this JS glue (roughly):

```
const __wbg_bar_target = Foo.prototype.bar;

export function __wbg_bar_a81456386e6b526f(arg0, arg1, arg2) {
    let varg1 = getStringFromWasm(arg1, arg2);
                                                    return
addHeapObject(__wbg_bar_target.call(getObject(arg0), varg1));
}
```

The difference here is pretty subtle, but we can see how the function being called is hoisted out of the generated shim and is bound to always be `Foo.prototype.bar`. This then uses the `Function.call` method to invoke that function with `getObject(arg0)` as the receiver.

But wait, there's still a JS function shim here even with `final`! That's true, and this is simply a fact of future WebAssembly proposals not being implemented yet. The semantics, though, match the future component model proposal because the method being called is determined exactly once, and it's located on the prototype chain rather than being resolved at runtime when the function is called.

Interaction with future proposals

If you're curious to see how our JS function shim will be eliminated entirely, let's take a look at the generated bindings. We're starting off with this:

```
const __wbg_bar_target = Foo.prototype.bar;

export function __wbg_bar_a81456386e6b526f(arg0, arg1, arg2) {
    let varg1 = getStringFromWasm(arg1, arg2);
                                                    return
addHeapObject(__wbg_bar_target.call(getObject(arg0), varg1));
}
```

... and once the [reference types proposal] is implemented then we won't need some of these pesky functions. That'll transform our generated JS shim to look like:

```
const __wbg_bar_target = Foo.prototype.bar;

export function __wbg_bar_a81456386e6b526f(arg0, arg1, arg2) {
    let varg1 = getStringFromWasm(arg1, arg2);
    return __wbg_bar_target.call(arg0, varg1);
}
```

Getting better! Next up we need the component model proposal. Note that the proposal is undergoing some changes right now so it's tough to link to reference documentation, but it suffices to say that it'll empower us with at least two different features.

First, component model promises to provide the concept of "argument conversions". The `arg1` and `arg2` values here are actually a pointer and a length to a utf-8 encoded string, and with component model we'll be able to annotate that this import should take those two arguments and convert them to a JS string (that is, the *host* should do this, the WebAssembly engine). Using that feature we can further trim this down to:

```
const __wbg_bar_target = Foo.prototype.bar;

export function __wbg_bar_a81456386e6b526f(arg0, varg1) {
```

```
    return __wbg_bar_target.call(arg0, varg1);
}
```

And finally, the second promise of the component model proposal is that we can flag a function call to indicate the first argument is the `this` binding of the function call. Today the `this` value of all called imported functions is `undefined`, and this flag (configured with component model) will indicate the first argument here is actually the `this`.

With that in mind we can further transform this to:

```
export const __wbg_bar_a81456386e6b526f = Foo.prototype.bar;
```

and voila! We, with reference types and component model, now have no JS function shim at all necessary to call the imported function. Additionally future Wasm proposals to the ES module system may also mean that don't even need the `export const ...` here too.

It's also worth pointing out that with all these Wasm proposals implemented the default way to import the `bar` function (aka `structural`) would generate a JS function shim that looks like:

```
export function __wbg_bar_a81456386e6b526f(varg1) {
    return this.bar(varg1);
}
```

where this import is still subject to runtime prototype chain lookups and such.

`indexing_getter`, `indexing_setter`, and `indexing_deleter`

These three attributes indicate that a method is an dynamically intercepted getter, setter, or deleter on the receiver object itself, rather than a direct access of the receiver's properties. It is equivalent calling the Proxy handler for the `obj[prop]` operation with some dynamic `prop` variable in JavaScript, rather than a normal static property access like `obj.prop` on a normal JavaScript `Object`.

This is useful for binding to `Proxy`s and some builtin DOM types that dynamically intercept property accesses.

- `indexing_getter` corresponds to `obj[prop]` operation in JavaScript. The function annotated must have a `this` receiver parameter, a single parameter that is used for indexing into the receiver (`prop`), and a return type.
- `indexing_setter` corresponds to the `obj[prop] = val` operation in JavaScript. The function annotated must have a `this` receiver parameter, a parameter for indexing into the receiver (`prop`), and a value parameter (`val`).
- `indexing_deleter` corresponds to `delete obj[prop]` operation in JavaScript. The function annotated must have a `this` receiver and a single parameter for indexing into the receiver (`prop`).

These must always be used in conjunction with the `structural` and `method` flags.

For example, consider this JavaScript snippet that uses `Proxy`:

```
const foo = new Proxy({}, {
    get(obj, prop) {
        return prop in obj ? obj[prop] : prop.length;
```

```
    },
    set(obj, prop, value) {
        obj[prop] = value;
    },
    deleteProperty(obj, prop) {
        delete obj[prop];
    },
});

foo.ten;
// 3

foo.ten = 10;
foo.ten;
// 10

delete foo.ten;
foo.ten;
// 3
```

To bind that in `wasm-bindgen` in Rust, we would use the `indexing_*` attributes on methods:

```
#[wasm_bindgen]
extern "C" {
    type Foo;
    #[wasm_bindgen(thread_local_v2)]
    static FOO: Foo;

    #[wasm_bindgen(method, structural, indexing_getter)]
    fn get(this: &Foo, prop: &str) -> u32;

    #[wasm_bindgen(method, structural, indexing_setter)]
    fn set(this: &Foo, prop: &str, val: u32);

    #[wasm_bindgen(method, structural, indexing_deleter)]
    fn delete(this: &Foo, prop: &str);
```

```
}

FOO.with(|foo| {
    assert_eq!(foo.get("ten"), 3);

    foo.set("ten", 10);
    assert_eq!(foo.get("ten"), 10);

    foo.delete("ten");
    assert_eq!(foo.get("ten"), 3);
});
```

`js_class = "Blah"`

The `js_class` attribute can be used in conjunction with the `method` attribute to bind methods of imported JavaScript classes that have been renamed on the Rust side.

```
#[wasm_bindgen]
extern "C" {
    // We don't want to import JS strings as `String`, since Rust already has a
      //  `String`  type  in  its  prelude,  so  rename  it  as `JsString`.
    #[wasm_bindgen(js_name = String)]
    type JsString;

    // This is a method on the JavaScript "String" class, so specify that with
    // the `js_class` attribute.
      #[wasm_bindgen(method,  js_class  =  "String",  js_name  = charAt)]
    fn char_at(this: &JsString, index: u32) -> JsString;
}
```

`js_name = blah`

The `js_name` attribute can be used to bind to a different function in JavaScript than the identifier that's defined in Rust.

Most often, this is used to convert a camel-cased JavaScript identifier into a snake-cased Rust identifier:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_name = jsOftenUsesCamelCase)]
    fn js_often_uses_camel_case() -> u32;
}
```

Sometimes, it is used to bind to JavaScript identifiers that are not valid Rust identifiers, in which case `js_name = "some string"` is used instead of `js_name = ident`:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_name = "$$$")]
    fn cash_money() -> u32;
}
```

However, you can also use `js_name` to define multiple signatures for polymorphic JavaScript functions:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console, js_name = log)]
    fn console_log_str(s: &str);

    #[wasm_bindgen(js_namespace = console, js_name = log)]
    fn console_log_u32(n: u32);

    #[wasm_bindgen(js_namespace = console, js_name = log)]
    fn console_log_many(a: u32, b: &JsValue);
}
```

All of these functions will call `console.log` in JavaScript, but each identifier will have only one signature in Rust.

Note that if you use `js_name` when importing a type you'll also need to use the `js_class` attribute when defining methods on the type:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_name = String)]
    type JsString;
    #[wasm_bindgen(method, getter, js_class = "String")]
    pub fn length(this: &JsString) -> u32;
}
```

The `js_name` attribute can also be used in situations where a JavaScript module uses `export default`. In this case, setting the `js_name` attribute to "default" on the `type` declaration, and the `js_class` attribute to "default" on any methods on the exported object will generate the correct imports.

For example, a module that would be imported directly in JavaScript:

```
import Foo from "bar";

let f = new Foo();
```

Could be accessed using this definition in Rust:

```
#[wasm_bindgen(module = "bar")]
extern "C" {
    #[wasm_bindgen(js_name = default)]
    type Foo;
    #[wasm_bindgen(constructor, js_class = default)]
    pub fn new() -> Foo;
}
```

`js_namespace = blah`

This attribute indicates that the JavaScript type is accessed through the given namespace. For example, the `WebAssembly.Module` APIs are all accessed through the `WebAssembly` namespace. `js_namespace` can be applied to any import (function or type) and whenever the generated JavaScript attempts to reference a name (like a class or function name) it'll be accessed through this namespace.

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);

    type Foo;
    #[wasm_bindgen(constructor, js_namespace = Bar)]
    fn new() -> Foo;
}

log("hello, console!");
Foo::new();
```

This is an example of how to bind namespaced items in Rust. The `log` and `Foo::new` functions will be available in the Rust module and will be invoked as `console.log` and `new Bar.Foo` in JavaScript.

It is also possible to access the JavaScript object under the nested namespace. `js_namespace` also accepts the array of the string to specify the namespace.

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = ["window", "document"])]
    fn write(s: &str);
}
```

```
write("hello, document!");
```

This example shows how to bind `window.document.write` in Rust.

If all items in the `extern "C" { … }` block have the same `js_namespace = …`:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = Math)]
    fn random() -> f64;
    #[wasm_bindgen(js_namespace = Math)]
    fn log(a: f64) -> f64;
    // ...
}
```

Then that macro argument can also be moved to the outer block:

```
#[wasm_bindgen(js_namespace = Math)]
extern "C" {
    #[wasm_bindgen]
    fn random() -> f64;
    #[wasm_bindgen]
    fn log(a: f64) -> f64;
    // ...
}
```

`js_namespace = …` on an individual item takes precedence over the outer block's `js_namespace = …`.

`method`

The `method` attribute allows you to describe methods of imported JavaScript objects. It is applied on a function that has `this` as its first parameter, which is a shared reference to an imported JavaScript type.

```
#[wasm_bindgen]
extern "C" {
    type Set;

    #[wasm_bindgen(method)]
    fn has(this: &Set, element: &JsValue) -> bool;
}
```

This generates a `has` method on `Set` in Rust, which invokes the `Set.prototype.has` method in JavaScript.

```
let set: Set = ...;
let elem: JsValue = ...;
if set.has(&elem) {
    ...
}
```

`module = "blah"`

The `module` attributes configures the module from which items are imported. For example,

```
#[wasm_bindgen(module = "wu/tang/clan")]
extern "C" {
    type ThirtySixChambers;
}
```

generates JavaScript import glue like:

```
import { ThirtySixChambers } from "wu/tang/clan";
```

If a `module` attribute is not present, then the global scope is used instead. For example,

```
#[wasm_bindgen]
extern "C" {
    fn illmatic() -> u32;
}
```

generates JavaScript import glue like:

```
let illmatic = this.illmatic;
```

Note that if the string specified with `module` starts with `./`, `../`, or `/` then it's interpreted as a path to a local JS snippet. If this doesn't work for your use case you might be interested in the `raw_module` attribute

`raw_module = "blah"`

This attribute performs exactly the same purpose as the `module` attribute on JS imports, but it does not attempt to interpret paths starting with `./`, `../`, or `/` as JS snippets. For example:

```
#[wasm_bindgen(raw_module = "./some/js/file.js")]
extern "C" {
    fn the_function();
}
```

Note that if you use this attribute with a relative or absolute path, it's likely up to the final bundler or project to assign meaning to that path. This typically means that the JS file or module will be resolved relative to the final location of the Wasm file itself. That means that `raw_module` is likely unsuitable for libraries on crates.io, but may be usable within end-user applications.

`no_deref`

The `no_deref` attribute can be used to say that no `Deref` impl should be generated for an imported type. If this attribute is not present, a `Deref` impl will be generated with a `Target` of the type's first `extends` attribute, or `Target = JsValue` if there are no `extends` attributes.

```
#[wasm_bindgen]
extern "C" {
    type Foo;

    #[wasm_bindgen(method)]
    fn baz(this: &Foo)

    #[wasm_bindgen(extends = Foo, no_deref)]
    type Bar;
}

fn do_stuff(bar: &Bar) {
    bar.baz() // Does not compile
}
```

`no_upcast`

The `no_upcast` attribute disables automatic generation of `Upcast` trait implementations for an imported type.

By default, wasm-bindgen automatically generates the following `Upcast` implementations for imported types:

- `Upcast<JsValue>` - all types can upcast to JsValue
- `Upcast<Self>` - identity upcast for non-generic types
- Structural covariance for generic types (`Type<T>` → `Type<U>` when `T: Upcast<U>`)
- `Upcast<SuperClass>` for each type in the `extends` attribute

Use `no_upcast` when you need to provide custom `Upcast` implementations, for example when a type has special covariance rules.

```
use wasm_bindgen::convert::Upcast;

#[wasm_bindgen]
extern "C" {
    // Automatic Upcast implementations are generated
    #[wasm_bindgen(extends = Object)]
    type NormalType;

    // No automatic Upcast implementations - must be provided
manually
    #[wasm_bindgen(extends = Object, no_upcast)]
    type CustomType<T>;
}

// Provide custom Upcast implementations for CustomType
impl<T> Upcast<JsValue> for CustomType<T> {}
impl<T> Upcast<Object> for CustomType<T> {}
// ... additional custom implementations
```

This is useful for types that have:

- Complex generic covariance rules (e.g., contravariant type parameters)
- Custom inheritance hierarchies
- Special relationship with other types that require manual `Upcast` implementations

`no_promising`

The `no_promising` attribute can be used to prevent automatic generation of the `Promising` trait implementation for an imported type.

This allows for custom thenable types (objects with a `then` method) when working with generic types in Wasm Bindgen where you need to specify what type the thenable resolves to.

Promising Trait

The `Promising` trait is used by generic Promise methods like `promise.then_map` to determine the resolution type of chained operations.

By default, wasm-bindgen assumes `impl Promising for T { type Resolution = T }`, meaning `Promise.resolve(t)` resolves to the type itself. However, for Promise-like objects, `Promise.resolve(t)` does not return a Promise, but its inner type, so this trait specifies what they actually resolve to.

Example: Custom Thenable Implementation

```
use wasm_bindgen::{Promising, prelude::*};

#[wasm_bindgen]
extern "C" {
    // Typing a custom thenable that resolves to Number
    #[wasm_bindgen(no_promising)]
    type NumericThenable;

    #[wasm_bindgen(method)]
              fn    then(this:    &NumericThenable,    callback:
&js_sys::Function) -> NumericThenable;
}

// Manually specify what this thenable resolves to via the
Promising trait
impl Promising for NumericThenable {
    type Resolution = js_sys::Number;
}

// Now then_map can correctly infer types
fn example(thenable: NumericThenable) {
    // Normally Promise.resolve accepts a Promise<Number> or a
Number:
               let    promise:    js_sys::Promise<Number>    =
js_sys::Promise::resolve(Number::from(5));

       let  promise  =  js_sys::Promise::resolve(thenable);  //
Correctly reflects as a Promise<Number>
}
```

Without `no_promising` and the manual implementation, the type system wouldn't know that `NumericThenable` resolves to `Number`, breaking generic inference for Promise operations.

`reexport`

The `reexport` attribute allows imported JavaScript items to be re-exported from your wasm module, making them available to JavaScript code that imports your module.

```
#[wasm_bindgen(module = "my-utils")]
extern "C" {
    #[wasm_bindgen(reexport)]
    fn helper_function() -> u32;
}
```

generates JavaScript export glue like:

```
import { helper_function } from "my-utils";
export { helper_function };
```

You can also provide a custom export name:

```
#[wasm_bindgen(module = "lodash")]
extern "C" {
    #[wasm_bindgen(reexport = "findIndex")]
    fn find_index(arr: &JsValue, val: &JsValue) -> i32;
}
```

which generates:

```
import { find_index } from "lodash";
export { find_index as findIndex };
```

Only top-level imports can be re-exported, `reexport` cannot be used on methods, constructors, or static methods.

`static_method_of = Blah`

The `static_method_of` attribute allows one to specify that an imported function is a static method of the given imported JavaScript class. For example, to bind to JavaScript's `Date.now()` static method, one would use this attribute:

```
#[wasm_bindgen]
extern "C" {
    type Date;

    #[wasm_bindgen(static_method_of = Date)]
    pub fn now() -> f64;
}
```

The `now` function becomes a static method of the imported type in the Rust bindings as well:

```
let instant = Date::now();
```

This is similar to the `js_namespace` attribute, but the usage from within Rust is different since the method also becomes a static method of the imported type. Additionally this attribute also specifies that the `this` parameter when invoking the method is expected to be the JS class, e.g. always invoked as `Date.now()` instead of `const x = Date.now; x()`.

`structural`

> **Note**: As of RFC 5 this attribute is the default for all imported functions. This attribute is largely ignored today and is only retained for backwards compatibility and learning purposes.
>
> The inverse of this attribute, the `final` attribute is more functionally interesting than `structural` (as `structural` is simply the default)

The `structural` flag can be added to `method` annotations, indicating that the method being accessed (or property with getters/setters) should be accessed in a structural, duck-type-y fashion. Rather than walking the constructor's prototype chain once at load time and caching the property result, the prototype chain is dynamically walked on every access.

```
#[wasm_bindgen]
extern "C" {
    type Duck;

    #[wasm_bindgen(method, structural)]
    fn quack(this: &Duck);

    #[wasm_bindgen(method, getter, structural)]
    fn is_swimming(this: &Duck) -> bool;
}
```

The constructor for the type here, `Duck`, is not required to exist in JavaScript (it's not referenced). Instead `wasm-bindgen` will generate shims that will access the passed in JavaScript value's `quack` method or its `is_swimming` property.

```
// Without `structural`, get the method directly off the
prototype at load time:
const Duck_prototype_quack = Duck.prototype.quack;
function quack(duck) {
  Duck_prototype_quack.call(duck);
```

```
}

// With `structural`, walk the prototype chain on every access:
function quack(duck) {
  duck.quack();
}
```

typescript_type

The `typescript_type` allows us to use typescript declarations in `typescript_custom_section` as arguments for rust functions! For example:

```
#[wasm_bindgen(typescript_custom_section)]
const ITEXT_STYLE: &'static str = r#"
interface ITextStyle {
    bold: boolean;
    italic: boolean;
    size: number;
}
"#;

#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(typescript_type = "ITextStyle")]
    pub type ITextStyle;
}

#[wasm_bindgen]
#[derive(Default)]
pub struct TextStyle {
    pub bold: bool,
    pub italic: bool,
    pub size: i32,
}

#[wasm_bindgen]
impl TextStyle {
    #[wasm_bindgen(constructor)]
    pub fn new(i: ITextStyle) -> TextStyle {
        let _js_value: JsValue = i.into();
        // parse JsValue
```

```
        TextStyle::default()
    }

    pub fn optional_new(_i: Option<ITextStyle>) -> TextStyle {
        // parse JsValue
        TextStyle::default()
    }
}
```

We can write our `typescript` code like:

```
import { ITextStyle, TextStyle } from "./my_awesome_module";

const style: TextStyle = new TextStyle({
  bold: true,
  italic: true,
  size: 42,
});

const optional_style: TextStyle = TextStyle.optional_new();
```

Variadic Parameters

In javascript, both the types of function arguments, and the number of function arguments are dynamic. For example

```
function sum(...rest) {
    let i;
    // the old way
    let old_way = 0;
    for (i=0; i<arguments.length; i++) {
        old_way += arguments[i];
    }
    // the new way
    let new_way = 0;
    for (i=0; i<rest.length; i++) {
        new_way += rest[i];
    }
    // both give the same answer
    assert(old_way === new_way);
    return new_way;
}
```

This function doesn't translate directly into rust, since we don't currently support variadic arguments on the Wasm target. To bind to it, we use a slice as the last argument, and annotate the function as variadic:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(variadic)]
    fn sum(args: &[i32]) -> i32;
}
```

when we call this function, the last argument will be expanded as the javascript expects.

For functions that accept heterogeneous types (like `console.log`), you can use `&[JsValue]`:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console, variadic)]
    fn log(args: &[JsValue]);
}

// Usage:
log(&[JsValue::from("Hello"),                JsValue::from(42),
JsValue::TRUE]);
```

To export a rust function to javascript with a variadic argument, we will use the same bindgen variadic attribute and assume that the last argument will be the variadic array. For example the following rust function:

```
#[wasm_bindgen(variadic)]
pub  fn  variadic_function(arr: &JsValue) -> JsValue {
        arr.into()
}
```

will generate the following TS interface

```
export  function  variadic_function(...arr:  any):  any;
```

Vendor-prefixed APIs

On the web new APIs often have vendor prefixes while they're in an experimental state. For example the `AudioContext` API is known as `webkitAudioContext` in Safari at the time of this writing. The `vendor_prefix` attribute indicates these alternative names, which are used if the normal name isn't defined.

For example to use `AudioContext` you might do:

```
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(vendor_prefix = webkit)]
    type AudioContext;

    // methods on `AudioContext` ...
}
```

Whenever `AudioContext` is used it'll use `AudioContext` if the global namespace defines it or alternatively it'll fall back to `webkitAudioContext`.

Note that `vendor_prefix` cannot be used with `module = "..."` or `js_namespace = ...`, so it's basically limited to web-platform APIs today.

`#[wasm_bindgen]` on Rust Exports

This section enumerates the attributes available for customizing bindings for Rust functions and `struct`s exported to JavaScript.

`constructor`

When attached to a Rust "constructor" it will make the generated JavaScript bindings callable as `new Foo()`.

For example, consider this exported Rust type and `constructor` annotation:

```
#[wasm_bindgen]
pub struct Foo {
    contents: u32,
}

#[wasm_bindgen]
impl Foo {
    #[wasm_bindgen(constructor)]
    pub fn new() -> Foo {
        Foo { contents: 0 }
    }

    pub fn get_contents(&self) -> u32 {
        self.contents
    }
}
```

This can be used in JavaScript as:

```
import { Foo } from './my_module';

const f = new Foo();
console.log(f.get_contents());
```

Caveats

In versions `>=v0.2.48, <0.2.88` of `wasm-bindgen`, there is a bug which breaks inheritance of exported Rust structs from JavaScript side (see #3213). If you want to inherit from a Rust struct such as:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub struct Parent {
    msg: String,
}

#[wasm_bindgen]
impl Parent {
    #[wasm_bindgen(constructor)]
    fn new() -> Self {
        Parent {
            msg: String::from("Hello from Parent!"),
        }
    }
}
```

You will need to reset the prototype of `this` back to the `Child` class prototype after calling the `Parent`'s constructor via `super`.

```
import { Parent } from './my_module';

class Child extends Parent {
    constructor() {
        super();
        Object.setPrototypeOf(this, Child.prototype);
    }
}
```

This is no longer required as of v0.2.88.

`js_name = Blah`

The `js_name` attribute can be used to export a different name in JS than what something is named in Rust. It can be applied to both exported Rust functions and types.

For example, this is often used to convert between Rust's snake-cased identifiers into JavaScript's camel-cased identifiers:

```
#[wasm_bindgen(js_name = doTheThing)]
pub fn do_the_thing() -> u32 {
    42
}
```

This can be used in JavaScript as:

```
import { doTheThing } from './my_module';

const x = doTheThing();
console.log(x);
```

Like imports, `js_name` can also be used to rename types exported to JS:

```
#[wasm_bindgen(js_name = Foo)]
pub struct JsFoo {
    // ..
}
```

to be accessed like:

```
import { Foo } from './my_module';

// ...
```

Note that attaching methods to the JS class `Foo` should be done via the `js_class` attribute:

```
#[wasm_bindgen(js_name = Foo)]
pub struct JsFoo { /* ... */ }

#[wasm_bindgen(js_class = Foo)]
```

```
impl JsFoo {
    // ...
}
```

It can also be used to rename parameters of exported functions and methods:

```
#[wasm_bindgen]
pub fn foo(
    #[wasm_bindgen(js_name = "firstArg")]
    arg1: String,
) {
    // function body
}

#[wasm_bindgen]
pub struct Foo {
    // properties
}

#[wasm_bindgen]
impl Foo {
    pub fn foo(
        &self,
        #[wasm_bindgen(js_name = "firstArg")]
        arg1: u32,
    ) {
        // function body
    }
}
```

Which will generate the following JS bindings:

```
/**
 * @param {string} firstArg
 */
export function foo(firstArg) {
    // ...
```

```
}

export class Foo {
    /**
     * @param {number} firstArg
     */
    foo(firstArg) {
        // ...
    }
}
```

`js_namespace = blah`

This attribute indicates that the exported Rust function or type should be placed within the given JavaScript namespace. Instead of exporting items individually, they will be grouped together in a namespace object. This is useful for organizing related exports and avoiding namespace pollution.

```
#[wasm_bindgen(js_namespace = math)]
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}

#[wasm_bindgen(js_namespace = math)]
pub fn multiply(a: i32, b: i32) -> i32 {
    a * b
}
```

This will generate JavaScript that exports a namespace object containing the functions:

```
export const math = { add, multiply };
```

Which can be used from JavaScript as:

```
import { math } from './my_module';

const sum = math.add(2, 3);
const product = math.multiply(4, 5);
```

Default Export

This feature can be used to define a default export object:

```
#[wasm_bindgen(js_namespace = "default")]
pub struct Counter {
    value: i32,
}
```

resulting in the output:

```
export default {
    Counter
}
```

which can be imported with:

```
import module from './my_module';

const counter = new module.Counter();
```

Nested Namespaces

It is also possible to create nested namespaces by providing an array of strings to specify the namespace path:

```
#[wasm_bindgen(js_namespace = ["utils", "string"])]
pub fn concat(a: &str, b: &str) -> String {
    format!("{}{}", a, b)
}

#[wasm_bindgen(js_namespace = ["utils", "string"])]
pub fn uppercase(s: &str) -> String {
    s.to_uppercase()
}
```

This will generate nested namespace objects:

```
const utils = {
    string: { concat, uppercase }
};
export { utils };
```

Which can be accessed in JavaScript as:

```
import { utils } from './my_module';

const result = utils.string.concat("hello", "world");
const upper = utils.string.uppercase("hello");
```

`js_class = Blah`

The `js_class` attribute is used to indicate that all the methods inside an `impl` block should be attached to the specified JS class instead of inferring it from the self type in the `impl` block. The `js_class` attribute is most frequently paired with the `js_name` attribute on structs:

```
#[wasm_bindgen(js_name = Foo)]
pub struct JsFoo { /* ... */ }

#[wasm_bindgen(js_class = Foo)]
impl JsFoo {
    #[wasm_bindgen(constructor)]
    pub fn new() -> JsFoo { /* ... */ }

    pub fn foo(&self) { /* ... */ }
}
```

which is accessed like:

```
import { Foo } from './my_module';

const x = new Foo();
x.foo();
```

`readonly`

When attached to a `pub` struct field this indicates that it's read-only from JavaScript, and a setter will not be generated and exported to JavaScript.

```
#[wasm_bindgen]
pub fn make_foo() -> Foo {
    Foo {
        first: 10,
        second: 20,
    }
}

#[wasm_bindgen]
pub struct Foo {
    pub first: u32,

    #[wasm_bindgen(readonly)]
    pub second: u32,
}
```

Here the `first` field will be both readable and writable from JS, but the `second` field will be a `readonly` field in JS where the setter isn't implemented and attempting to set it will throw an exception.

```
import { make_foo } from "./my_module";

const foo = make_foo();

// Can both get and set `first`.
foo.first = 99;
console.log(foo.first);

// Can only get `second`.
console.log(foo.second);
```

skip

When attached to a `pub` struct field this indicates that field will not be exposed to JavaScript, and neither getter nor setter will be generated in ES6 class.

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub struct Foo {
    pub bar: u32,

    #[wasm_bindgen(skip)]
    pub baz: u32,
}

#[wasm_bindgen]
impl Foo {
    pub fn new() -> Self {
        Foo {
            bar: 1,
            baz: 2
        }
    }
}
```

Here the `bar` field will be both readable and writable from JS, but the `baz` field will be `undefined` in JS.

```
import('./pkg/').then(rust => {
    let foo = rust.Foo.new();

    // bar is accessible by getter
    console.log(foo.bar);
    // field marked with `skip` is undefined
    console.log(foo.baz);
```

```
    // you can shadow it
    foo.baz = 45;
    // so accessing by getter will return `45`
    // but it won't affect real value in rust memory
    console.log(foo.baz);
});
```

skip_jsdoc

When attached to a function or a method, prevents `wasm-bindgen` from auto-generating JSDoc-style doc comments. By default, `wasm-bindgen` adds `@param` and `@returns` annotations to doc comments in the generated JS files. A `skip_jsdoc` annotation prevents this, allowing you to supply your own doc comments.

The following rust uses `skip_jsdoc` to omit one of the auto-generated doc comments.

```
use wasm_bindgen::prelude::*;

/// Autogenerated docs.
#[wasm_bindgen]
pub fn foo(arg: u32) -> u32 { arg + 1 }

/// Manually written docs.
///
/// @param {number} arg - A descriptive description.
/// @returns {number} Something a bit bigger.
#[wasm_bindgen(skip_jsdoc)]
pub fn bar(arg: u32) -> u32 { arg + 2 }
```

The `wasm-bindgen`-generated JS interface of the above code will look something like this:

```
/**
* Autogenerated docs.
*
* @param {number} arg
* @returns {number}
*/
export function foo(arg) { /* ... */ }

/**
```

```
* Manually written docs.
*
* @param {number} arg - A descriptive description.
* @returns {number} Something a bit bigger.
*/
export function bar(arg) { /* ... */ }
```

`start`

When attached to a function this attribute will configure the `start` section of the Wasm executable to be emitted, executing the tagged function as soon as the Wasm module is instantiated.

```
#[wasm_bindgen(start)]
fn start() {
    // executed automatically ...
}
```

The `start` section of the Wasm executable will be configured to execute the `start` function here as soon as it can. Note that due to various practical limitations today the start section of the executable may not literally point to `start`, but the `start` function here should be started up automatically when the wasm module is loaded.

Multiple start functions

Multiple `#[wasm_bindgen(start)]` functions can be specified across a module and its dependencies. They will be chained together and all executed during initialization. The execution order is arbitrary, so functions should not rely on other start functions having run before them.

```
#[wasm_bindgen(start)]
fn init_logging() {
    // ...
}

#[wasm_bindgen(start)]
fn init_state() {
    // ...
}
```

Private start functions

By default, start functions are also exported to JS as regular functions. To register a start function that runs at initialization but is not exported to JS, use the `private` attribute:

```
#[wasm_bindgen(start, private)]
fn my_init() {
    // runs at startup, but not callable from JS
}
```

Caveats

- The `start` function must take no arguments and must either return `()` or `Result<(), JsValue>`
- The `start` function will not be executed when testing.

`main`

When attached to the `main` function this attribute will adjust it to properly throw errors if they can be.

```
#[wasm_bindgen(main)]
fn main() -> Result<(), JsValue> {
    Err(JsValue::from("this error message will be thrown"))
}
```

The attribute also allows using `async fn main()` in Cargo binaries.

```
#[wasm_bindgen(main)]
async fn main() {
    // ...
    future.await;
}
```

This attribute is only intended to be used on the `main` function of binaries or examples. Unlike `#[wasm_bindgen(start)]`, it will not cause an arbitrary function to be executed on start in a library.

The return type support is modeled after `Termination`. `()` and `Infallible` are supported, but `Termination` itself is not. In order, wasm-bindgen will first detect a `Result<(), impl Into<JsValue>>` and will throw proper `JsValue`s, `Result<(), impl Debug>` will convert an error to a string and throw that.

`this`

Typically, `this` bindings are achieved via `impl` block definitions for methods.

But since all regular JavaScript functions may accept arbitrary `this` bindings, it is possible to support this as well for free functions.

The `#[wasm_bindgen(this)]` attribute can be applied to exported Rust functions to make them receive the JavaScript `this` value as their first parameter. This allows creating functions that can be called with `.call()` or `.apply()` to set the `this` context.

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen(this)]
pub fn get_property(this: &JsValue, property: &str) -> JsValue
{
    js_sys::Reflect::get(this, &property.into()).unwrap()
}

#[wasm_bindgen(this)]
pub fn add_to_count(foo: &JsValue, value: u32) -> u32 {
    let current = js_sys::Reflect::get(foo, &"count".into())
        .unwrap()
        .as_f64()
        .unwrap() as u32;
    current + value
}
```

With the above called via `get_property.call({ str: 'foo' }, 'str')` or `add_to_count.apply({ count: 1 }, [7])` respectively.

Can only be used on functions, not methods or static methods within `impl` blocks, and the function must have at least one parameter.

typescript_custom_section

When added to a `const &'static str`, it will append the contents of the string to the `.d.ts` file exported by `wasm-bindgen-cli` (when the `--typescript` flag is enabled).

```
#[wasm_bindgen(typescript_custom_section)]
const TS_APPEND_CONTENT: &'static str = r#"

export type Coords = { "latitude": number, "longitude": number, };

"#;
```

The primary target for this feature is for code generation. For example, you can author a macro that allows you to export a TypeScript definition alongside the definition of a struct or Rust type.

```
#[derive(MyTypescriptExport)]
struct Coords {
    latitude: u32,
    longitude: u32,
}
```

The proc_derive_macro "MyTypescriptExport" can export its own `#[wasm_bindgen(typescript_custom_section)]` section, which would then be picked up by wasm-bindgen-cli. This would be equivalent to the contents of the TS_APPEND_CONTENT string in the first example.

This feature allows plain data objects to be typechecked in Rust and in TypeScript by outputting a type definition generated at compile time.

`getter` and `setter`

The `getter` and `setter` attributes can be used in Rust `impl` blocks to define properties in JS that act like getters and setters of a field. For example:

```
#[wasm_bindgen]
pub struct Baz {
    field: i32,
}

#[wasm_bindgen]
impl Baz {
    #[wasm_bindgen(constructor)]
    pub fn new(field: i32) -> Baz {
        Baz { field }
    }

    #[wasm_bindgen(getter)]
    pub fn field(&self) -> i32 {
        self.field
    }

    #[wasm_bindgen(setter)]
    pub fn set_field(&mut self, field: i32) {
        self.field = field;
    }
}
```

Can be combined in `JavaScript` like in this snippet:

```
const obj = new Baz(3);
assert.equal(obj.field, 3);
obj.field = 4;
assert.equal(obj.field, 4);
```

You can also configure the name of the property that is exported in JS like so:

```
#[wasm_bindgen]
impl Baz {
    #[wasm_bindgen(getter = anotherName)]
    pub fn field(&self) -> i32 {
        self.field
    }

    #[wasm_bindgen(setter = anotherName)]
    pub fn set_field(&mut self, field: i32) {
        self.field = field;
    }
}
```

Getters are expected to take no arguments other than `&self` and return the field's type. Setters are expected to take one argument other than `&mut self` (or `&self`) and return no values.

The name for a `getter` is by default inferred from the function name it's attached to. The default name for a `setter` is the function's name minus the `set_` prefix, and if `set_` isn't a prefix of the function it's an error to not provide the name explicitly.

`inspectable`

By default, structs exported from Rust become JavaScript classes with a single `ptr` property. All other properties are implemented as getters, which are not displayed when calling `toJSON`.

The `inspectable` attribute can be used on Rust structs to provide a `toJSON` and `toString` implementation that display all readable fields. For example:

```
#[wasm_bindgen(inspectable)]
pub struct Baz {
    pub field: i32,
    private: i32,
}

#[wasm_bindgen]
impl Baz {
    #[wasm_bindgen(constructor)]
    pub fn new(field: i32) -> Baz {
        Baz { field, private: 13 }
    }
}
```

Provides the following behavior as in this JavaScript snippet:

```
const obj = new Baz(3);
assert.deepStrictEqual(obj.toJSON(), { field: 3 });
obj.field = 4;
assert.strictEqual(obj.toString(), '{"field":4}');
```

One or both of these implementations can be overridden as desired. Note that the generated `toString` calls `toJSON` internally, so overriding `toJSON` will affect its output as a side effect.

```
#[wasm_bindgen]
impl Baz {
    #[wasm_bindgen(js_name = toJSON)]
```

```
    pub fn to_json(&self) -> i32 {
        self.field
    }

    #[wasm_bindgen(js_name = toString)]
    pub fn to_string(&self) -> String {
        format!("Baz: {}", self.field)
    }
}
```

Note that the output of `console.log` will remain unchanged and display only the `ptr` field in browsers. It is recommended to call `toJSON` or `JSON.stringify` in these situations to aid with logging or debugging. Node.js does not suffer from this limitation, see the section below.

`inspectable` Classes in Node.js

When the `nodejs` target is used, an additional `[util.inspect.custom]` implementation is provided which calls `toJSON` internally. This method is used for `console.log` and similar functions to display all readable fields of the Rust struct.

skip_typescript

By default, Rust exports exposed to JavaScript will generate TypeScript definitions (unless `--no-typescript` is used). The `skip_typescript` attribute can be used to disable type generation per function, enum, struct, or field. For example:

```
#[wasm_bindgen(skip_typescript)]
pub enum MyHiddenEnum {
    One,
    Two,
    Three
}

#[wasm_bindgen]
pub struct MyPoint {
    pub x: u32,

    #[wasm_bindgen(skip_typescript)]
    pub y: u32,
}

#[wasm_bindgen]
impl MyPoint {

    #[wasm_bindgen(skip_typescript)]
    pub fn stringify(&self) -> String {
        format!("({}, {})", self.x, self.y)
    }
}
```

Will generate the following `.d.ts` file:

```
/* tslint:disable */
/* eslint-disable */
export class MyPoint {
```

```
  free(): void;
  x: number;
}
```

When combined with [the `typescript_custom_section` attribute], this can be used to manually specify more specific function types instead of using the generated definitions.

`getter_with_clone`

By default, Rust exports exposed to JavaScript will generate getters that require fields to implement `Copy`. The `getter_with_clone` attribute can be used to generate getters that require `Clone` instead. This attribute can be applied per struct or per field. For example:

```
#[wasm_bindgen]
pub struct Foo {
    #[wasm_bindgen(getter_with_clone)]
    pub bar: String,
}

#[wasm_bindgen(getter_with_clone)]
pub struct Foo {
    pub bar: String,
    pub baz: String,
}
```

`private`

The `private` attribute can be applied to exported structs and enums to generate the exported binding but not export it on the public exports of the JavaScript module. This allows defining types that can be used as arguments or return values in exported functions without exposing them in the public exported API.

```
#[wasm_bindgen(private)]
pub struct Config {
    pub timeout: i32,
}

#[wasm_bindgen]
pub fn create_config(timeout: i32) -> Config {
    Config { timeout }
}

#[wasm_bindgen]
pub fn apply_config(config: Config) -> i32 {
    config.timeout * 2
}
```

This will generate JavaScript where `Config` is defined internally but not exported:

```
import * as app from './app';

// app.Config is undefined - not exported
console.log(app.Config); // undefined

// But the functions that use Config are exported
const config = app.create_config(100);
const result = app.apply_config(config);
```

The TypeScript definitions will still export the type under its name as a type-only export:

```
export function create_config(timeout: number): Config;
export function apply_config(config: Config): number;

class Config {
    timeout: number;
}

export type { Config };
```

The `private` attribute is only supported on structs and enums, not on functions.

`unchecked_return_type`, `unchecked_param_type`, and `unchecked_optional_param_type`

Return and parameter types of exported functions and methods can be overwritten with `#[wasm_bindgen(unchecked_return_type)]` and `#[wasm_bindgen(unchecked_param_type)]`.

Parameters can also be marked as optional using `#[wasm_bindgen(unchecked_optional_param_type)]`, which generates TypeScript's `?:` syntax and JSDoc's `[paramName]` syntax.

> **Note**: Types that are provided using `#[wasm_bindgen(unchecked_return_type)]`, `#[wasm_bindgen(unchecked_param_type)]`, and `#[wasm_bindgen(unchecked_optional_param_type)]` aren't checked for their contents. They will end up in a function signature and JSDoc exactly as they have been specified. E.g. `#[wasm_bindgen(unchecked_return_type = "number")]` on a function returning `String` will return a `string`, not a `number`, even if the TS signature and JSDoc will say otherwise.

> **Note**: `unchecked_param_type` and `unchecked_optional_param_type` are mutually exclusive and cannot be used on the same parameter. As in TypeScript, a required parameter cannot follow an optional parameter.

```
#[wasm_bindgen(unchecked_return_type = "Foo")]
pub fn foo(
    #[wasm_bindgen(unchecked_param_type = "Bar")]
    arg1: JsValue,
) -> JsValue {
```

```
    // function body
}

#[wasm_bindgen]
pub fn greet(
    #[wasm_bindgen(unchecked_optional_param_type = "string")]
    name: JsValue,
) -> JsValue {
    if name.is_undefined() {
        "Hello, World!".into()
    } else {
                                        format!("Hello,     {}!",
name.as_string().unwrap_or_default()).into()
    }
}

#[wasm_bindgen]
pub struct Foo {
    // properties
}

#[wasm_bindgen]
impl Foo {
    #[wasm_bindgen(unchecked_return_type = "Baz")]
    pub fn foo(
        &self,
        #[wasm_bindgen(unchecked_param_type = "Bar")]
        arg1: JsValue,
    ) -> JsValue {
        // function body
    }
}
```

Which will generate the following JS bindings:

```
/**
 * @param {Bar} arg1
```

```
 * @returns {Foo}
 */
export function foo(arg1) {
    // ...
}

/**
 * @param {string} [name]
 * @returns {any}
 */
export function greet(name) {
    // ...
}

export class Foo {
    /**
     * @param {Bar} arg1
     * @returns {Baz}
     */
    foo(arg1) {
        // ...
    }
}
```

And the following TypeScript definitions:

```
export function foo(arg1: Bar): Foo;
export function greet(name?: string): any;
```

`return_description` and `param_description`

Descriptions to return and parameter documentation can be added with `#[wasm_bindgen(return_description)]` and `#[wasm_bindgen(param_description)]`.

```
/// Adds `arg1` and `arg2`.
#[wasm_bindgen(return_description = "the result of the addition of `arg1` and `arg2`")]
pub fn add(
    #[wasm_bindgen(param_description = "the first number")]
    arg1: u32,
    #[wasm_bindgen(param_description = "the second number")]
    arg2: u32,
) -> u32 {
    arg1 + arg2
}

#[wasm_bindgen]
pub struct FooList {
    // properties
}

#[wasm_bindgen]
impl FooList {
    /// Returns the number at the given index.
      #[wasm_bindgen(return_description = "the number at the given index")]
    pub fn number(
        &self,
          #[wasm_bindgen(param_description = "the index of the number to be returned")]
```

```
        index: u32,
    ) -> u32 {
        // function body
    }
}
```

Which will generate the following JS bindings:

```
/**
 * Adds `arg1` and `arg2`.
 *
 * @param {number} arg1 - the first number
 * @param {number} arg2 - the second number
 * @returns {number} the result of the addition of `arg1` and
`arg2`
 */
export function add(arg1, arg2) {
    // ...
}

export class FooList {
    /**
     * Returns the number at the given index.
     *
      * @param {number} index - the index of the number to be
returned
     * @returns {number} the number at the given index
     */
    number(index) {
        // ...
    }
}
```

The `web-sys` Crate

The `web-sys` crate provides raw `wasm-bindgen` imports for all of the Web's APIs. This includes:

- `window.fetch`
- `Node.prototype.appendChild`
- WebGL
- WebAudio
- and many more!

It's sort of like the `libc` crate, but for the Web.

It does *not* include the JavaScript APIs that are guaranteed to exist in all standards-compliant ECMAScript environments, such as `Array`, `Date`, and `eval`. Bindings for these APIs can be found in the `js-sys` crate.

API Documentation

Read the `web-sys` API documentation here!

Using `web-sys`

Add `web-sys` as a dependency to your `Cargo.toml`

```
[dependencies]
wasm-bindgen = "0.2"

[dependencies.web-sys]
version = "0.3"
features = [
]
```

Enable the cargo features for the APIs you're using

To keep build times super speedy, `web-sys` gates each Web interface behind a cargo feature. Find the type or method you want to use in the API documentation; it will list the features that must be enabled to access that API.

For example, if we're looking for the `window.resizeTo` function, we would search for `resizeTo` in the API documentation. We would find the `web_sys::Window::resize_to` function, which requires the `Window` feature. To get access to that function, we enable the `Window` feature in `Cargo.toml`:

```
[dependencies.web-sys]
version = "0.3"
features = [
  "Window"
]
```

Call the method!

```
use wasm_bindgen::prelude::*;
use web_sys::Window;

#[wasm_bindgen]
pub fn make_the_window_small() {
    // Resize the window to 500px by 500px.
    let window = web_sys::window().unwrap();
    window.resize_to(500, 500)
        .expect("could not resize the window");
}
```

Cargo Features in `web-sys`

To keep `web-sys` building as fast as possible, there is a cargo feature for every type defined in `web-sys`. To access that type, you must enable its feature. To access a method, you must enable the feature for its `self` type and the features for each of its argument types. In the API documentation, every method lists the features that are required to enable it.

For example, the `WebGlRenderingContext::compile_shader` function requires these features:

- `WebGlRenderingContext`, because that is the method's `self` type
- `WebGlShader`, because it takes an argument of that type

Function Overloads

Many Web APIs are overloaded to take different types of arguments or to skip arguments completely. `web-sys` contains multiple bindings for these functions that each specialize to a particular overload and set of argument types.

For example, the `fetch` API can be given a URL string, or a `Request` object, and it might also optionally be given a `RequestInit` options object. Therefore, we end up with these `web-sys` functions that all bind to the `window.fetch` function:

- `Window::fetch_with_str`
- `Window::fetch_with_request`
- `Window::fetch_with_str_and_init`
- `Window::fetch_with_request_and_init`

Note that different overloads can use different interfaces, and therefore can require different sets of cargo features to be enabled.

Type Translations in `web-sys`

Most of the types specified in WebIDL (the interface definition language for all Web APIs) have relatively straightforward translations into `web-sys`, but it's worth calling out a few in particular:

- `BufferSource` and `ArrayBufferView` - these two types show up in a number of APIs that generally deal with a buffer of bytes. We bind them in `web-sys` with two different types, `js_sys::Object` and `&mut [u8]`. Using `js_sys::Object` allows passing in arbitrary JS values which represent a view of bytes (like any typed array object), and `&mut [u8]` allows using a raw slice in Rust. Unfortunately we must pessimistically assume that JS will modify all slices as we don't currently have information of whether they're modified or not.
- Callbacks are all represented as `js_sys::Function`. This means that all callbacks going through `web-sys` are a raw JS value. You can work with this by either juggling actual `js_sys::Function` instances or you can create a `Closure<dyn FnMut(...)>`, extract the underlying `JsValue` with `as_ref`, and then use `JsCast::unchecked_ref` to convert it to a `js_sys::Function`.

Inheritance in `web-sys`

Inheritance between JS classes is the bread and butter of how the DOM works on the web, and as a result it's quite important for `web-sys` to provide access to this inheritance hierarchy as well! There are few ways you can access the inheritance hierarchy when using `web-sys`.

Accessing parent classes using `Deref`

Like smart pointers in Rust, all types in `web_sys` implement `Deref` to their parent JS class. This means, for example, if you have a `web_sys::Element` you can create a `web_sys::Node` from that implicitly:

```
let element: &Element = ...;

element.append_child(..); // call a method on `Node`

method_expecting_a_node(&element);   //   coerce   to   `&Node`
implicitly

let node: &Node = &element; // explicitly coerce to `&Node`
```

Using `Deref` allows ergonomic transitioning up the inheritance hierarchy to the parent class and beyond, giving you access to all the methods using the `.` operator.

Accessing parent classes using `AsRef`

In addition to `Deref`, the `AsRef` trait is implemented for all types in `web_sys` for all types in the inheritance hierarchy. For example for the `HtmlAnchorElement` type you'll find:

```
impl AsRef<HtmlElement> for HtmlAnchorElement
impl AsRef<Element> for HtmlAnchorElement
impl AsRef<Node> for HtmlAnchorElement
impl AsRef<EventTarget> for HtmlAnchorElement
```

```
impl AsRef<Object> for HtmlAnchorElement
impl AsRef<JsValue> for HtmlAnchorElement
```

You can use `.as_ref()` to explicitly get a reference to any parent class from from a type in `web_sys`. Note that because of the number of `AsRef` implementations you'll likely need to have type inference guidance as well.

Accessing child classes using `JsCast`

Finally the `wasm_bindgen::JsCast` trait can be used to implement all manner of casts between types. It supports static unchecked casts between types as well as dynamic runtime-checked casts (using `instanceof`) between types.

More documentation about this can be found on the trait itself

Unstable APIs

It's common for browsers to implement parts of a web API while the specification for that API is still being written. The API may require frequent changes as the specification continues to be developed, so the WebIDL is relatively unstable.

This causes some challenges for `web-sys` because it means `web-sys` would have to make breaking API changes whenever the WebIDL changes. It also means that previously published `web-sys` versions would be invalid, because the browser API may have been changed to match the updated WebIDL.

To avoid frequent breaking changes for unstable APIs, `web-sys` hides all unstable APIs through an attribute that looks like:

```
#[cfg(web_sys_unstable_apis)]
pub struct Foo;
```

By hiding unstable APIs through an attribute, it's necessary for crates to explicitly opt-in to these reduced stability guarantees in order to use these APIs. Specifically, these APIs do not follow semver and may break whenever the WebIDL changes.

Crates can opt-in to unstable APIs at compile-time by passing the `cfg` flag `web_sys_unstable_apis`.

Typically the `RUSTFLAGS` environment variable is used to do this. For example:

```
RUSTFLAGS=--cfg=web_sys_unstable_apis cargo run
```

Alternatively, you can create a cargo config file to set its rustflags:

Within `./.cargo/config.toml`:

```
[build]
rustflags = ["--cfg=web_sys_unstable_apis"]
```

Testing on `wasm32-unknown-unknown` with `wasm-bindgen-test`

The `wasm-bindgen-test` crate is an experimental test harness for Rust programs compiled to Wasm using `wasm-bindgen` and the `wasm32-unknown-unknown` target.

Goals

- Write tests for Wasm as similar as possible to how you normally would write `#[test]`-style unit tests for native targets.
- Run the tests with the usual `cargo test` command but with an explicit wasm target:

```
cargo test --target wasm32-unknown-unknown
```

Using `wasm-bindgen-test`

Add `wasm-bindgen-test` to Your `Cargo.toml`'s `[dev-dependencies]`

```
[dev-dependencies]
wasm-bindgen-test = "0.3.0"
```

Note that the `0.3.0` track of `wasm-bindgen-test` supports Rust 1.39.0+, which is currently the nightly channel (as of 2019-09-05). If you want support for older compilers use the `0.2.*` track of `wasm-bindgen-test`.

Write Some Tests

Create a `$MY_CRATE/tests/wasm.rs` file:

```
use wasm_bindgen_test::*;

#[wasm_bindgen_test]
fn pass() {
    assert_eq!(1, 1);
}

#[wasm_bindgen_test]
fn fail() {
    assert_eq!(1, 2);
}

// On a target other then `wasm32-unknown-unknown`, the `#[test]` attribute
// will be used instead, allowing this test to run on any target.
#[wasm_bindgen_test(unsupported = test)]
fn all_targets() {
    assert_eq!(1, 2);
}
```

Writing tests is the same as normal Rust `#[test]`s, except we are using the `#[wasm_bindgen_test]` attribute.

One other difference is that the tests **must** be in the root of the crate, or within a `pub mod`. Putting them inside a private module will not work.

Execute Your Tests

Run the tests with `wasm-pack test`. By default, the tests are generated to target Node.js, but you can configure tests to run inside headless browsers as well.

```
$ wasm-pack test --node
    Finished dev [unoptimized + debuginfo] target(s) in 0.11s
                  Running    /home/.../target/wasm32-unknown-
unknown/debug/deps/wasm-4a309ffe6ad80503.wasm
running 2 tests

test wasm::pass ... ok
test wasm::fail ... FAILED

failures:

---- wasm::fail output ----
    error output:
        panicked at 'assertion failed: `(left == right)`
          left: `1`,
         right: `2`', crates/test/tests/wasm.rs:14:5

    JS exception that was thrown:
        RuntimeError: unreachable
            at __rust_start_panic (wasm-function[1362]:33)
            at rust_panic (wasm-function[1357]:30)
                                                          at
std::panicking::rust_panic_with_hook::h56e5e464b0e7fc22 (wasm-
function[1352]:444)
                                                          at
std::panicking::continue_panic_fmt::had70ba48785b9a8f   (wasm-
function[1350]:122)
                                                          at
std::panicking::begin_panic_fmt::h991e7d1ca9bf9c0c      (wasm-
function[1351]:95)
```

```
                    at wasm::fail::ha4c23c69dfa0eea9 (wasm-
function[88]:477)
                                                          at
core::ops::function::FnOnce::call_once::h633718dad359559a
(wasm-function[21]:22)
                                                          at
wasm_bindgen_test::__rt::Context::execute::h2f669104986475eb
(wasm-function[13]:291)
            at __wbg_test_fail_1 (wasm-function[87]:57)
                at module.exports.__wbg_apply_2ba774592c5223a7
(/home/alex/code/wasm-bindgen/target/wasm32-unknown-
unknown/wbg-tmp/wasm-4a309ffe6ad80503.js:61:66)


failures:

    wasm::fail

test result: FAILED. 1 passed; 1 failed; 0 ignored

error: test failed, to rerun pass '--test wasm'
```

That's it!

Appendix: Using `wasm-bindgen-test` without `wasm-pack`

⚠️ **The recommended way to use `wasm-bindgen-test` is with `wasm-pack`, since it will handle installing the test runner, installing a WebDriver client for your browser, and informing `cargo` how to use the custom test runner.** However, you can also manage those tasks yourself, if you wish.

In addition to the steps above, you must also do the following.

Install the Test Runner

The test runner comes along with the main `wasm-bindgen` CLI tool. Make sure to replace "X.Y.Z" with the same version of `wasm-bindgen` that you already have in `Cargo.toml`!

```
cargo install wasm-bindgen-cli --vers "X.Y.Z"
```

Configure `.cargo/config` to use the Test Runner

Add this to `$MY_CRATE/.cargo/config`:

```
[target.wasm32-unknown-unknown]
runner = 'wasm-bindgen-test-runner'
```

Run the Tests

Run the tests by passing `--target wasm32-unknown-unknown` to `cargo test`:

```
cargo test --target wasm32-unknown-unknown
```

Running doctests requires at least Rust v1.89.

Writing Asynchronous Tests

Not all tests can execute immediately and some may need to do "blocking" work like fetching resources and/or other bits and pieces. To accommodate this asynchronous tests are also supported through the `futures` and `wasm-bindgen-futures` crates.

Writing an asynchronous test is pretty simple, just use an `async` function! `js_sys::Promise` implements `IntoFuture`, so you can `.await` it directly:

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen_test]
async fn my_async_test() {
    // Create a promise that is ready on the next tick of the micro task queue.
    let promise = js_sys::Promise::resolve(&JsValue::from(42));

    // Await the promise directly — no conversion wrapper needed.
    let x = promise.await.unwrap();
    assert_eq!(x, 42);
}
```

If you need a named `Future` value, `JsFuture` is available from `js_sys::futures` or from `wasm_bindgen_futures`:

```
use wasm_bindgen_futures::JsFuture;

let x = JsFuture::from(promise).await.unwrap();
```

Rust compiler compatibility

Note that `async` functions are only supported in stable from Rust 1.39.0 and beyond.

If you're using the `futures` crate from crates.io in its 0.1 version then you'll want to use the `0.3.*` version of `wasm-bindgen-futures` and the `0.2.8` version of `wasm-bindgen-test`. In those modes you'll also need to use `#[wasm_bindgen_test(async)]` instead of using an `async` function. In general we'd recommend using the nightly version with `async` since the user experience is much improved!

Testing in Headless Browsers

Configure via Environment Variables

By default tests run on Node.js. To target browsers you can use the `WASM_BINDGEN_USE_BROWSER` environment variable:

```
WASM_BINDGEN_USE_BROWSER=1 cargo test --target wasm32-unknown-unknown
```

The following configurations are available:

- `WASM_BINDGEN_USE_DEDICATED_WORKER`: for dedicated workers
- `WASM_BINDGEN_USE_SHARED_WORKER`: for shared workers
- `WASM_BINDGEN_USE_SERVICE_WORKER`: for service workers
- `WASM_BINDGEN_USE_DENO`: for Deno
- `WASM_BINDGEN_USE_NODE_EXPERIMENTAL`: for Node.js but as an ES module

Force Configuration

Tests can also be forced to run in a certain environment by using the `wasm_bindgen_test_configure!` macro:

```
use wasm_bindgen_test::wasm_bindgen_test_configure;

// Run in a browser.
wasm_bindgen_test_configure!(run_in_browser);
// Or run in a dedicated worker.
wasm_bindgen_test_configure!(run_in_dedicated_worker);
// Or run in a shared worker.
wasm_bindgen_test_configure!(run_in_shared_worker);
// Or run in a service worker.
wasm_bindgen_test_configure!(run_in_service_worker);
// Or run in Node.js but as an ES module.
wasm_bindgen_test_configure!(run_in_node_experimental);
```

Note that this will ignore any environment variable set.

Configuring Which Browser is Used

To control which browser is used for headless testing, use the appropriate flag with `wasm-pack test`:

- `wasm-pack test --chrome` — Run the tests in Chrome. This machine must have Chrome installed.
- `wasm-pack test --firefox` — Run the tests in Firefox. This machine must have Firefox installed.
- `wasm-pack test --safari` — Run the tests in Safari. This machine must have Safari installed.

If multiple browser flags are passed, the tests will be run under each browser.

Running the Tests in the Headless Browser

Once the tests are configured to run in a headless browser, just run `wasm-pack test` with the appropriate browser flags and `--headless`:

```
wasm-pack test --headless --chrome --firefox --safari
```

Configuring Headless Browser capabilities

Either add the file `webdriver.json` to the root of your crate or ensure the environment variable `WASM_BINDGEN_TEST_WEBDRIVER_JSON` points to one. Each browser has own section for capabilities. For example:

```
{
  "moz:firefoxOptions": {
    "prefs": {
      "media.navigator.streams.fake": true,
      "media.navigator.permission.disabled": true
    },
    "args": []
  },
  "goog:chromeOptions": {
    "args": [
      "--use-fake-device-for-media-stream",
      "--use-fake-ui-for-media-stream"
    ]
  }
}
```

Full list supported capabilities can be found:

- for Chrome - here
- for Firefox - here

Note that the `headless` argument is always enabled for both browsers.

Debugging Headless Browser Tests

Omitting the `--headless` flag will disable headless mode, and allow you to debug failing tests in your browser's devtools.

Appendix: Testing in headless browsers without `wasm-pack`

⚠️ **The recommended way to use `wasm-bindgen-test` is with `wasm-pack`, since it will handle installing the test runner, installing a WebDriver client for your browser, and informing `cargo` how to use the custom test runner.** However, you can also manage those tasks yourself, if you wish.

Configuring Which Browser is Used

If one of the following environment variables is set, then the corresponding WebDriver and browser will be used. If none of these environment variables are set, then the `$PATH` is searched for a suitable WebDriver implementation.

`GECKODRIVER=path/to/geckodriver`

Use Firefox for headless browser testing, and `geckodriver` as its WebDriver.

The `firefox` binary must be on your `$PATH`.

Get `geckodriver` here

`CHROMEDRIVER=path/to/chromedriver`

Use Chrome for headless browser testing, and `chromedriver` as its WebDriver.

The `chrome` binary must be on your `$PATH`.

Get `chromedriver` here

`SAFARIDRIVER=path/to/safaridriver`

Use Safari for headless browser testing, and `safaridriver` as its WebDriver.

This is installed by default on Mac OS. It should be able to find your Safari installation by default.

Running the Tests in the Remote Headless Browser

Tests can be run on a remote webdriver. To do this, the above environment variables must be set as URL to the remote webdriver. For example:

```
CHROMEDRIVER_REMOTE=http://remote.host/
```

Running the Tests in the Headless Browser

Once the tests are configured to run in a headless browser and the appropriate environment variables are set, executing the tests for headless browsers is the same as executing them for Node.js:

```
cargo test --target wasm32-unknown-unknown
```

Debugging Headless Browser Tests

Set the `NO_HEADLESS=1` environment variable and the browser tests will not run headless. Instead, the tests will start a local server that you can visit in your Web browser of choices, and headless testing should not be used. You can then use your browser's devtools to debug.

Setting Up Continuous Integration with `wasm-bindgen-test`

This page contains example configurations for running `wasm-bindgen-test`-based tests in various CI services.

Is your favorite CI service missing? Send us a pull request!

Travis CI

```
language: rust
rust    : nightly

addons:
  firefox: latest
  chrome : stable

install:
            -      curl      https://rustwasm.github.io/wasm-
pack/installer/init.sh -sSf | sh

script:

   # this will test the non Wasm targets if your crate has
those, otherwise remove this line.
  #
  - cargo test

  - wasm-pack test --firefox --headless
  - wasm-pack test --chrome  --headless
```

AppVeyor

```
install:
  - ps: Install-Product node 10
          -     appveyor-retry     appveyor     DownloadFile
https://win.rustup.rs/ -FileName rustup-init.exe
  - rustup-init.exe -y --default-host x86_64-pc-windows-msvc -
-default-toolchain nightly
  - set PATH=%PATH%;C:\Users\appveyor\.cargo\bin
  - rustc -V
  - cargo -V
  - rustup target add wasm32-unknown-unknown
  - cargo install wasm-bindgen-cli

build: false

test_script:
   # Test in Chrome. chromedriver is installed by default in
appveyor.
  - set CHROMEDRIVER=C:\Tools\WebDriver\chromedriver.exe
  - cargo test --target wasm32-unknown-unknown
  - set CHROMEDRIVER=
  # Test in Firefox. geckodriver is also installed by default.
  - set GECKODRIVER=C:\Tools\WebDriver\geckodriver.exe
  - cargo test --target wasm32-unknown-unknown
```

GitHub Actions

```
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Install
                  run:  curl  https://rustwasm.github.io/wasm-
pack/installer/init.sh -sSf | sh

      - run: cargo test
      - run: wasm-pack test --headless --chrome
      - run: wasm-pack test --headless --firefox
```

Generating Coverage Data

You can ask the runner to generate coverage data from functions marked as `#[wasm_bindgen_test]` in the `.profraw` format.

Coverage is still in an experimental state, requires Rust Nightly, may be unreliable and could experience breaking changes at any time.

Enabling the feature

To enable this feature, you need to enable `cfg(wasm_bindgen_unstable_test_coverage)`.

Generating the data

`RUSTFLAGS` that need to be present

Make sure you are using `RUSTFLAGS=-Cinstrument-coverage -Zno-profiler-runtime`.

Due to the current limitation of `llvm-cov`, we can't collect profiling symbols from the generated `.wasm` files. Instead, we can grab them from the LLVM IR with `--emit=llvm-ir` by using Clang. Usage of Clang or any LLVM tools must match the LLVM version used by Rust.

Arguments to the test runner

Like with Rust test coverage, you can use the `LLVM_PROFILE_FILE` environment variable to specify a path for the generated `.profraw` files.

Target features

This feature relies on the minicov crate, which provides a profiling runtime for WebAssembly. It in turn uses cc to compile the runtime to Wasm, which currently doesn't support accounting for target feature. Use e.g. `CFLAGS_wasm32_unknown_unknown="-matomics -mbulk-memory"` to account for that.

Example

Requires rust >= 1.87.0 and wasm-bindgen-test >= 0.3.57.

Install cargo-llvm-cov and run with:

```
CARGO_TARGET_WASM32_UNKNOWN_UNKNOWN_RUNNER=wasm-bindgen-test-runner \
CARGO_TARGET_WASM32_UNKNOWN_UNKNOWN_RUSTFLAGS="-Cinstrument-coverage -Zno-profiler-runtime -Clink-args=--no-gc-sections --cfg=wasm_bindgen_unstable_test_coverage" \
cargo +nightly llvm-cov test --target wasm32-unknown-unknown
```

Attribution

These methods have originally been pioneered by Hacken OÜ, see their guide as well.

Benchmark

You can now code wasm benchmarks in the same way you code write them for other platforms. We modified the criterion to support wasm while retaining only some basic bench capabilities. Thanks to the criterion, we were able to obtain stable, accurate, and reliable benching results!

I'll assume that we have a crate, `mycrate`, whose `lib.rs` contains the following code:

```
pub fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 1,
        1 => 1,
        n => fibonacci(n - 1) + fibonacci(n - 2),
    }
}
```

Step 1 - Add Dependency to Cargo.toml

To enable benchmarks, add the following to your `Cargo.toml` file:

```
[dev-dependencies]
wasm-bindgen-test = "0.3.56"
```

Step 2 - Add Benchmark

As an example, we'll benchmark our implementation of the Fibonacci function. Create a benchmark file at `$PROJECT/benches/my_benchmark.rs` with the following contents:

```
use wasm_bindgen_test::{wasm_bindgen_bench, Criterion};
use mycrate::fibonacci;

#[wasm_bindgen_bench]
fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("fib 20", |b| b.iter(|| fibonacci(20)));
}
```

Step 3 - Run Benchmark

To run this benchmark, use the following commands:

```
export        CARGO_TARGET_WASM32_UNKNOWN_UNKNOWN_RUNNER=wasm-bindgen-test-runner
cargo bench --target wasm32-unknown-unknown
```

You should see output similar to this:

```
      Running benches/my_benchmark.rs (target/wasm32-unknown-unknown/release/deps/my_benchmark-18dc40eb54e84f66.wasm)
Warming up for 3.0000 s
Collecting 100 samples in estimated 5.0216 s (545k iterations)
fib 20                  time:   [9.1197 µs 9.1268 µs 9.1339 µs]
Found 2 outliers among 100 measurements (2.00%)
  2 (2.00%) high mild
```

Step 4 - Optimize

This fibonacci function is quite inefficient. We can do better:

```
pub fn fibonacci(n: u64) -> u64 {
    let mut a = 0;
    let mut b = 1;

    match n {
        0 => b,
        _ => {
            for _ in 0..n {
                let c = a + b;
                a = b;
                b = c;
            }
            b
        }
    }
}
```

Running the benchmark now produces output like this:

```
      Running benches/my_benchmark.rs (target/wasm32-unknown-
unknown/release/deps/my_benchmark-18dc40eb54e84f66.wasm)
Warming up for 3.0000 s
Collecting 100 samples in estimated 5.0000 s (3.2B iterations)
fib 20                  time:   [1.5432 ns 1.5434 ns 1.5438
ns]
                        change: [−99.983% −99.983% −99.983%]
(p = 0.00 < 0.05)
                       Performance has improved.
Found 13 outliers among 100 measurements (13.00%)
  1 (1.00%) low mild
  5 (5.00%) high mild
  7 (7.00%) high severe
```

As you can see, Criterion is statistically confident that our optimization has made an improvement. If we introduce a performance regression, Criterion will instead print a message indicating this.

Step 5 - Asynchronous function

It also supports benchmarking asynchronous functions:

```
use wasm_bindgen_test::{wasm_bindgen_bench, Criterion};

#[wasm_bindgen_bench]
async fn bench(c: &mut Criterion) {
    c.bench_async_function(
        "bench desc",
         Box::pin(
             b.iter_future(|| async {
                 // Code to benchmark goes here
             })
         )
    ).await;
}
```

Step 6 - Run benchmark in browser

Similar to test, you can use `wasm_bindgen_test_configure!` to configure the execution environment.

Step 7 - Configuration

- `WASM_BINDGEN_BENCH_RESULT`: Path for the custom benchmark result file.

Contributing to `wasm-bindgen`

This section contains instructions on how to get this project up and running for development. You may want to browse the [unpublished guide documentation] for `wasm-bindgen` as well as it may have more up-to-date information.

Prerequisites

1. Rust. Install Rust. Once Rust is installed, run

   ```
   rustup target add wasm32-unknown-unknown
   ```

2. The tests for this project use Node. Make sure you have node >= 10 installed, as that is when WebAssembly support was introduced. Install Node.

Code Formatting

Although formatting rules are not mandatory, it is encouraged to run `cargo fmt` (`rustfmt`) with its default rules within a PR to maintain a more organized code base. If necessary, a PR with a single commit that formats the entire project is also welcome.

Running `wasm-bindgen`'s Tests

Wasm Tests on Node and Headless Browsers

These are the largest test suites, and most common to run in day to day `wasm-bindgen` development. These tests are compiled to Wasm and then run in Node.js or a headless browser via the WebDriver protocol.

```
WASM_BINDGEN_SPLIT_LINKED_MODULES=1 cargo test --target wasm32-unknown-unknown
```

See the `wasm-bindgen-test` crate's `README.md` for details and configuring which headless browser is used.

Sanity Tests for `wasm-bindgen` on the Native Host Target

This small test suite just verifies that exported `wasm-bindgen` methods can still be used on the native host's target.

```
cargo test
```

The Web IDL Frontend's Tests

```
cargo test -p webidl-tests --target wasm32-unknown-unknown
```

The Macro UI Tests

These tests assert that we have reasonable error messages that point to the right source spans when the `#[wasm_bindgen]` proc-macro is misused.

You can run these tests by running `cargo test` for the `wasm-bindgen-macro` crate:

```
cargo test -p wasm-bindgen-macro
```

The `js-sys` Tests

See the `js-sys` testing page.

The `web-sys` Tests

See the `web-sys` testing page.

Design of `wasm-bindgen`

This section is intended to be a deep-dive into how `wasm-bindgen` internally works today, specifically for Rust. If you're reading this far in the future it may no longer be up to date, but feel free to open an issue and we can try to answer questions and/or update this!

Foundation: ES Modules

The first thing to know about `wasm-bindgen` is that it's fundamentally built on the idea of ES Modules. In other words this tool takes an opinionated stance that Wasm files *should be viewed as ES modules*. This means that you can `import` from a Wasm file, use its `export`-ed functionality, etc, from normal JS files.

Now unfortunately at the time of this writing the interface of Wasm interop isn't very rich. Wasm modules can only call functions or export functions that deal exclusively with `i32`, `i64`, `f32`, and `f64`. Bummer!

That's where this project comes in. The goal of `wasm-bindgen` is to enhance the "ABI" of Wasm modules with richer types like classes, JS objects, Rust structs, strings, etc. Keep in mind, though, that everything is based on ES Modules! This means that the compiler is actually producing a "broken" Wasm file of sorts. The wasm file emitted by rustc, for example, does not have the interface we would like to have. Instead it requires the `wasm-bindgen` tool to postprocess the file, generating a `foo.js` and `foo_bg.wasm` file. The `foo.js` file is the desired interface expressed in JS (classes, types, strings, etc) and the `foo_bg.wasm` module is simply used as an implementation detail (it was lightly modified from the original `foo.wasm` file).

As more features are stabilized in WebAssembly over time (like component model) the JS file is expected to get smaller and smaller. It's unlikely to ever disappear, but `wasm-bindgen` is designed to follow the WebAssembly spec and proposals closely to optimize JS/Rust as much as possible.

Foundation #2: Unintrusive in Rust

On the more Rust-y side of things the `wasm-bindgen` crate is designed to ideally have as minimal impact on a Rust crate as possible. Ideally a few `#[wasm_bindgen]` attributes are annotated in key locations and otherwise you're off to the races. The attribute strives to both not invent new syntax and work with existing idioms today.

For example a library might exposed a function in normal Rust that looks like:

```
pub fn greet(name: &str) -> String {
    // ...
}
```

And with `#[wasm_bindgen]` all you need to do in exporting it to JS is:

```
#[wasm_bindgen]
pub fn greet(name: &str) -> String {
    // ...
}
```

Additionally the design here with minimal intervention in Rust should allow us to easily take advantage of the upcoming component model proposal. Ideally you'd simply upgrade `wasm-bindgen`-the-crate as well as your toolchain and you're immediately getting raw access to component model! (this is still a bit of a ways off though...)

Polyfill for "JS objects in wasm"

One of the main goals of `wasm-bindgen` is to allow working with and passing around JS objects in wasm, but that's not allowed today! While indeed true, that's where the polyfill comes in.

The question here is how we shoehorn JS objects into a `u32` for Wasm to use. The current strategy for this approach is to maintain a module-local variable in the generated `foo.js` file: a `heap`.

Temporary JS objects on the "stack"

The first slots in the `heap` in `foo.js` are considered a stack. This stack, like typical program execution stacks, grows down. JS objects are pushed on the bottom of the stack, and their index in the stack is the identifier that's passed to wasm. A stack pointer is maintained to figure out where the next item is pushed.

JS objects are then only removed from the bottom of the stack as well. Removal is simply storing null then incrementing a counter. Because of the "stack-y" nature of this scheme it only works for when Wasm doesn't hold onto a JS object (aka it only gets a "reference" in Rust parlance).

Let's take a look at an example.

```
// foo.rs
#[wasm_bindgen]
pub fn foo(a: &JsValue) {
    // ...
}
```

Here we're using the special `JsValue` type from the `wasm-bindgen` library itself. Our exported function, `foo`, takes a *reference* to an object. This notably means that it can't persist the object past the lifetime of this function call.

Now what we actually want to generate is a JS module that looks like (in TypeScript parlance)

```
// foo.d.ts
export function foo(a: any);
```

and what we actually generate looks something like:

```
// foo.js
import * as wasm from './foo_bg';

const heap = new Array(32);
heap.push(undefined, null, true, false);
let stack_pointer = 32;

function addBorrowedObject(obj) {
  stack_pointer -= 1;
  heap[stack_pointer] = obj;
  return stack_pointer;
}

export function foo(arg0) {
  const idx0 = addBorrowedObject(arg0);
  try {
    wasm.foo(idx0);
  } finally {
    heap[stack_pointer++] = undefined;
  }
}
```

Here we can see a few notable points of action:

- The Wasm file was renamed to `foo_bg.wasm`, and we can see how the JS module generated here is importing from the Wasm file.
- Next we can see our `heap` module variable which is to store all JS values reference-able from wasm.
- Our exported function `foo`, takes an arbitrary argument, `arg0`, which is converted to an index with the `addBorrowedObject` object function. The index is then passed to Wasm so Wasm can operate with it.

- Finally, we have a `finally` which frees the stack slot as it's no longer used, popping the value that was pushed at the start of the function.

It's also helpful to dig into the Rust side of things to see what's going on there! Let's take a look at the code that `#[wasm_bindgen]` generates in Rust:

```
// what the user wrote
pub fn foo(a: &JsValue) {
    // ...
}

#[export_name = "foo"]
pub extern "C" fn __wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {
        ManuallyDrop::new(JsValue::__from_idx(arg0))
    };
    let arg0 = &*arg0;
    foo(arg0);
}
```

And as with the JS, the notable points here are:

- The original function, `foo`, is unmodified in the output
- A generated function here (with a unique name) is the one that's actually exported from the Wasm module
- Our generated function takes an integer argument (our index) and then wraps it in a `JsValue`. There's some trickery here that's not worth going into just yet, but we'll see in a bit what's happening under the hood.

Long-lived JS objects

The above strategy is useful when JS objects are only temporarily used in Rust, for example only during one function call. Sometimes, though, objects may have a dynamic lifetime or otherwise need to be stored on Rust's heap. To cope with this there's a second half of management of JS objects, naturally corresponding to the other side of the JS `heap` array.

JS Objects passed to Wasm that are not references are assumed to have a dynamic lifetime inside of the Wasm module. As a result the strict push/pop of the stack won't work and we need more permanent storage for the JS objects. To cope with this we build our own "slab allocator" of sorts.

A picture (or code) is worth a thousand words so let's show what happens with an example.

```
// foo.rs
#[wasm_bindgen]
pub fn foo(a: JsValue) {
    // ...
}
```

Note that the `&` is missing in front of the `JsValue` we had before, and in Rust parlance this means it's taking ownership of the JS value. The exported ES module interface is the same as before, but the ownership mechanics are slightly different. Let's see the generated JS's slab in action:

```
import * as wasm from './foo_bg'; // imports from Wasm file

const heap = new Array(32);
heap.push(undefined, null, true, false);
let heap_next = 36;

function addHeapObject(obj) {
  if (heap_next === heap.length)
    heap.push(heap.length + 1);
  const idx = heap_next;
  heap_next = heap[idx];
  heap[idx] = obj;
  return idx;
}

export function foo(arg0) {
  const idx0 = addHeapObject(arg0);
  wasm.foo(idx0);
}
```

```
export function __wbindgen_object_drop_ref(idx) {
  heap[idx ] = heap_next;
  heap_next = idx;
}
```

Unlike before we're now calling `addHeapObject` on the argument to `foo` rather than `addBorrowedObject`. This function will use `heap` and `heap_next` as a slab allocator to acquire a slot to store the object, placing a structure there once it's found. Note that this is going on the right-half of the array, unlike the stack which resides on the left half. This discipline mirrors the stack/heap in normal programs, roughly.

Another curious aspect of this generated module is the `__wbindgen_object_drop_ref` function. This is one that's actually imported to wasm rather than used in this module! This function is used to signal the end of the lifetime of a `JsValue` in Rust, or in other words when it goes out of scope. Otherwise though this function is largely just a general "slab free" implementation.

And finally, let's take a look at the Rust generated again too:

```
// what the user wrote
pub fn foo(a: JsValue) {
    // ...
}

#[export_name = "foo"]
pub extern "C" fn __wasm_bindgen_generated_foo(arg0: u32) {
    let arg0 = unsafe {
        JsValue::__from_idx(arg0)
    };
    foo(arg0);
}
```

Ah that looks much more familiar! Not much interesting is happening here, so let's move on to...

Anatomy of `JsValue`

Currently the `JsValue` struct is actually quite simple in Rust, it's:

```
pub struct JsValue {
    idx: u32,
}

// "private" constructors

impl Drop for JsValue {
    fn drop(&mut self) {
        unsafe {
            __wbindgen_object_drop_ref(self.idx);
        }
    }
}
```

Or in other words it's a newtype wrapper around a `u32`, the index that we're passed from wasm. The destructor here is where the `__wbindgen_object_drop_ref` function is called to relinquish our reference count of the JS object, freeing up our slot in the `slab` that we saw above.

If you'll recall as well, when we took `&JsValue` above we generated a wrapper of `ManuallyDrop` around the local binding, and that's because we wanted to avoid invoking this destructor when the object comes from the stack.

Working with heap in reality

The above explanations are pretty close to what happens today, but in reality there's a few differences especially around handling constant values like `undefined`, `null`, etc. Be sure to check out the actual generated JS and the generation code for the full details!

Exporting a function to JS

Alright now that we've got a good grasp on JS objects and how they're working, let's take a look at another feature of `wasm-bindgen`: exporting functionality with types that are richer than just numbers.

The basic idea around exporting functionality with more flavorful types is that the Wasm exports won't actually be called directly. Instead the generated `foo.js` module will have shims for all exported functions in the Wasm module.

The most interesting conversion here happens with strings so let's take a look at that.

```
#[wasm_bindgen]
pub fn greet(a: &str) -> String {
    format!("Hello, {}!", a)
}
```

Here we'd like to define an ES module that looks like

```
// foo.d.ts
export function greet(a: string): string;
```

To see what's going on, let's take a look at the generated shim

```
import * as wasm from './foo_bg';

function passStringToWasm(arg) {
  const buf = new TextEncoder('utf-8').encode(arg);
  const len = buf.length;
  const ptr = wasm.__wbindgen_malloc(len, 1);
  let array = new Uint8Array(wasm.memory.buffer);
  array.set(buf, ptr);
  return [ptr, len];
}

function getStringFromWasm(ptr, len) {
  const mem = new Uint8Array(wasm.memory.buffer);
```

```
  const slice = mem.slice(ptr, ptr + len);
  const ret = new TextDecoder('utf-8').decode(slice);
  return ret;
}

export function greet(arg0) {
  const [ptr0, len0] = passStringToWasm(arg0);
  try {
    const ret = wasm.greet(ptr0, len0);
    const ptr = wasm.__wbindgen_boxed_str_ptr(ret);
    const len = wasm.__wbindgen_boxed_str_len(ret);
    const realRet = getStringFromWasm(ptr, len);
    wasm.__wbindgen_boxed_str_free(ret);
    return realRet;
  } finally {
    wasm.__wbindgen_free(ptr0, len0, 1);
  }
}
```

Phew, that's quite a lot! We can sort of see though if we look closely what's happening:

- Strings are passed to Wasm via two arguments, a pointer and a length. Right now we have to copy the string onto the Wasm heap which means we'll be using `TextEncoder` to actually do the encoding. Once this is done we use an internal function in `wasm-bindgen` to allocate space for the string to go, and then we'll pass that ptr/length to Wasm later on.
- Returning strings from Wasm is a little tricky as we need to return a ptr/len pair, but Wasm currently only supports one return value (multiple return values is being standardized). To work around this in the meantime, we're actually returning a pointer to a ptr/len pair, and then using functions to access the various fields.
- Some cleanup ends up happening in wasm. The `__wbindgen_boxed_str_free` function is used to free the return value of `greet` after it's been decoded onto the JS heap (using

`TextDecoder`). The `__wbindgen_free` is then used to free the space we allocated to pass the string argument once the function call is done.

Next let's take a look at the Rust side of things as well. Here we'll be looking at a mostly abbreviated and/or "simplified" in the sense of this is what it compiles down to:

```
pub extern "C" fn greet(a: &str) -> String {
    format!("Hello, {}!", a)
}

#[export_name = "greet"]
pub extern "C" fn __wasm_bindgen_generated_greet(
    arg0_ptr: *const u8,
    arg0_len: usize,
) -> *mut String {
    let arg0 = unsafe {
            let slice = ::std::slice::from_raw_parts(arg0_ptr,
arg0_len);
        ::std::str::from_utf8_unchecked(slice)
    };
    let _ret = greet(arg0);
    Box::into_raw(Box::new(_ret))
}
```

Here we can see again that our `greet` function is unmodified and has a wrapper to call it. This wrapper will take the ptr/len argument and convert it to a string slice, while the return value is boxed up into just a pointer and is then returned up to was for reading via the `__wbindgen_boxed_str_*` functions.

So in general exporting a function involves a shim both in JS and in Rust with each side translating to or from Wasm arguments to the native types of each language. The `wasm-bindgen` tool manages hooking up all these shims while the `#[wasm_bindgen]` macro takes care of the Rust shim as well.

Most arguments have a relatively clear way to convert them, bit if you've got any questions just let me know!

Exporting a struct to JS

So far we've covered JS objects, importing functions, and exporting functions. This has given us quite a rich base to build on so far, and that's great! We sometimes, though, want to go even further and define a JS `class` in Rust. Or in other words, we want to expose an object with methods from Rust to JS rather than just importing/exporting free functions.

The `#[wasm_bindgen]` attribute can annotate both a `struct` and `impl` blocks to allow:

```
#[wasm_bindgen]
pub struct Foo {
    internal: i32,
}

#[wasm_bindgen]
impl Foo {
    #[wasm_bindgen(constructor)]
    pub fn new(val: i32) -> Foo {
        Foo { internal: val }
    }

    pub fn get(&self) -> i32 {
        self.internal
    }

    pub fn set(&mut self, val: i32) {
        self.internal = val;
    }
}
```

This is a typical Rust `struct` definition for a type with a constructor and a few methods. Annotating the struct with `#[wasm_bindgen]` means that we'll generate necessary trait impls to convert this type to/from the JS boundary. The annotated `impl` block here means that the functions inside

will also be made available to JS through generated shims. If we take a look at the generated JS code for this we'll see:

```
import * as wasm from './js_hello_world_bg';

export class Foo {
    static __construct(ptr) {
        return new Foo(ptr);
    }

    constructor(ptr) {
        this.ptr = ptr;
    }

    free() {
        const ptr = this.ptr;
        this.ptr = 0;
        wasm.__wbg_foo_free(ptr);
    }

    static new(arg0) {
        const ret = wasm.foo_new(arg0);
        return Foo.__construct(ret)
    }

    get() {
        const ret = wasm.foo_get(this.ptr);
        return ret;
    }

    set(arg0) {
        const ret = wasm.foo_set(this.ptr, arg0);
        return ret;
    }
}
```

That's actually not much! We can see here though how we've translated from Rust to JS:

- Associated functions in Rust (those without `self`) turn into `static` functions in JS.
- Methods in Rust turn into methods in wasm.
- Manual memory management is exposed in JS as well. The `free` function is required to be invoked to deallocate resources on the Rust side of things.

To be able to use `new Foo()`, you'd need to annotate `new` as `#[wasm_bindgen(constructor)]`.

One important aspect to note here, though, is that once `free` is called the JS object is "neutered" in that its internal pointer is nulled out. This means that future usage of this object should trigger a panic in Rust.

The real trickery with these bindings ends up happening in Rust, however, so let's take a look at that.

```
// original input to `#[wasm_bindgen]` omitted ...

#[export_name = "foo_new"]
pub extern "C" fn __wasm_bindgen_generated_Foo_new(arg0: i32)
-> u32 {
    let ret = Foo::new(arg0);
    Box::into_raw(Box::new(WasmRefCell::new(ret))) as u32
}

#[export_name = "foo_get"]
pub extern "C" fn __wasm_bindgen_generated_Foo_get(me: u32) ->
i32 {
    let me = me as *mut WasmRefCell<Foo>;
    wasm_bindgen::__rt::assert_not_null(me);
    let me = unsafe { &*me };
    return me.borrow().get();
}
```

```
#[export_name = "foo_set"]
pub extern "C" fn __wasm_bindgen_generated_Foo_set(me: u32,
arg1: i32) {
    let me = me as *mut WasmRefCell<Foo>;
    wasm_bindgen::__rt::assert_not_null(me);
    let me = unsafe { &*me };
    me.borrow_mut().set(arg1);
}

#[no_mangle]
pub unsafe extern "C" fn __wbindgen_foo_free(me: u32) {
    let me = me as *mut WasmRefCell<Foo>;
    wasm_bindgen::__rt::assert_not_null(me);
    (*me).borrow_mut(); // ensure no active borrows
    drop(Box::from_raw(me));
}
```

As with before this is cleaned up from the actual output but it's the same idea as to what's going on! Here we can see a shim for each function as well as a shim for deallocating an instance of `Foo`. Recall that the only valid wasm types today are numbers, so we're required to shoehorn all of `Foo` into a `u32`, which is currently done via `Box` (like `std::unique_ptr` in C++). Note, though, that there's an extra layer here, `WasmRefCell`. This type is the same as `RefCell` and can be mostly glossed over.

The purpose for this type, if you're interested though, is to uphold Rust's guarantees about aliasing in a world where aliasing is rampant (JS). Specifically the `&Foo` type means that there can be as much aliasing as you'd like, but crucially `&mut Foo` means that it is the sole pointer to the data (no other `&Foo` to the same instance exists). The `RefCell` type in libstd is a way of dynamically enforcing this at runtime (as opposed to compile time where it usually happens). Baking in `WasmRefCell` is the same idea here, adding runtime checks for aliasing which are typically happening at compile time. This is currently a Rust-specific feature which isn't actually in the `wasm-bindgen` tool itself, it's just in the Rust-generated code (aka the `#[wasm_bindgen]` attribute).

Importing a function from JS

Now that we've exported some rich functionality to JS it's also time to import some! The goal here is to basically implement JS `import` statements in Rust, with fancy types and all.

First up, let's say we invert the function above and instead want to generate greetings in JS but call it from Rust. We might have, for example:

```
#[wasm_bindgen(module = "./greet")]
extern "C" {
    fn greet(a: &str) -> String;
}

fn other_code() {
    let greeting = greet("foo");
    // ...
}
```

The basic idea of imports is the same as exports in that we'll have shims in both JS and Rust doing the necessary translation. Let's first see the JS shim in action:

```
import * as wasm from './foo_bg';

import { greet } from './greet';

// ...

export function __wbg_f_greet(ptr0, len0, wasmretptr) {
                const        [retptr,        retlen]        =
passStringToWasm(greet(getStringFromWasm(ptr0, len0)));
    (new  Uint32Array(wasm.memory.buffer))[wasmretptr  /  4]  =
retlen;
  return retptr;
}
```

The `getStringFromWasm` and `passStringToWasm` are the same as we saw before, and like with `__wbindgen_object_drop_ref` far above we've got this weird export from our module now! The `__wbg_f_greet` function is what's generated by `wasm-bindgen` to actually get imported in the `foo.wasm` module.

The generated `foo.js` we see imports from the `./greet` module with the `greet` name (was the function import in Rust said) and then the `__wbg_f_greet` function is shimming that import.

There's some tricky ABI business going on here so let's take a look at the generated Rust as well. Like before this is simplified from what's actually generated.

```
extern "C" fn greet(a: &str) -> String {
    extern "C" {
            fn __wbg_f_greet(a_ptr: *const u8, a_len: usize,
ret_len: *mut usize) -> *mut u8;
    }
    unsafe {
        let a_ptr = a.as_ptr();
        let a_len = a.len();
        let mut __ret_strlen = 0;
         let mut __ret_strlen_ptr = &mut __ret_strlen as *mut
usize;
                  let  _ret  =  __wbg_f_greet(a_ptr,  a_len,
__ret_strlen_ptr);
        String::from_utf8_unchecked(
                       Vec::from_raw_parts(_ret,  __ret_strlen,
__ret_strlen)
        )
    }
}
```

Here we can see that the `greet` function was generated but it's largely just a shim around the `__wbg_f_greet` function that we're calling. The ptr/len pair for the argument is passed as two arguments and for the return

value we're receiving one value (the length) indirectly while directly receiving the returned pointer.

Importing a class from JS

Just like with functions after we've started exporting we'll also want to import! Now that we've exported a `class` to JS we'll want to also be able to import classes in Rust as well to invoke methods and such. Since JS classes are in general just JS objects the bindings here will look pretty similar to the JS object bindings describe above.

As usual though, let's dive into an example!

```
#[wasm_bindgen(module = "./bar")]
extern "C" {
    type Bar;

    #[wasm_bindgen(constructor)]
    fn new(arg: i32) -> Bar;

    #[wasm_bindgen(js_namespace = Bar)]
    fn another_function() -> i32;

    #[wasm_bindgen(method)]
    fn get(this: &Bar) -> i32;

    #[wasm_bindgen(method)]
    fn set(this: &Bar, val: i32);

    #[wasm_bindgen(method, getter)]
    fn property(this: &Bar) -> i32;

    #[wasm_bindgen(method, setter)]
    fn set_property(this: &Bar, val: i32);
}

fn run() {
    let bar = Bar::new(Bar::another_function());
    let x = bar.get();
```

```
    bar.set(x + 3);

    bar.set_property(bar.property() + 6);
}
```

Unlike our previous imports, this one's a bit more chatty! Remember that one of the goals of `wasm-bindgen` is to use native Rust syntax wherever possible, so this is mostly intended to use the `#[wasm_bindgen]` attribute to interpret what's written down in Rust. Now there's a few attribute annotations here, so let's go through one-by-one:

- `#[wasm_bindgen(module = "./bar")]` - seen before with imports this is declare where all the subsequent functionality is imported from. For example the `Bar` type is going to be imported from the `./bar` module.
- `type Bar` - this is a declaration of JS class as a new type in Rust. This means that a new type `Bar` is generated which is "opaque" but is represented as internally containing a `JsValue`. We'll see more on this later.
- `#[wasm_bindgen(constructor)]` - this indicates that the binding's name isn't actually used in JS but rather translates to `new Bar()`. The return value of this function must be a bare type, like `Bar`.
- `#[wasm_bindgen(js_namespace = Bar)]` - this attribute indicates that the function declaration is namespaced through the `Bar` class in JS.
- `#[wasm_bindgen(static_method_of = SomeJsClass)]` - this attribute is similar to `js_namespace`, but instead of producing a free function, produces a static method of `SomeJsClass`.
- `#[wasm_bindgen(method)]` - and finally, this attribute indicates that a method call is going to happen. The first argument must be a JS struct, like `Bar`, and the call in JS looks like `Bar.prototype.set.call(...)`.

With all that in mind, let's take a look at the JS generated.

```
import * as wasm from './foo_bg';

```

```
import { Bar } from './bar';

// other support functions omitted...

export function __wbg_s_Bar_new() {
  return addHeapObject(new Bar());
}

const another_function_shim = Bar.another_function;
export function __wbg_s_Bar_another_function() {
  return another_function_shim();
}

const get_shim = Bar.prototype.get;
export function __wbg_s_Bar_get(ptr) {
  return shim.call(getObject(ptr));
}

const set_shim = Bar.prototype.set;
export function __wbg_s_Bar_set(ptr, arg0) {
  set_shim.call(getObject(ptr), arg0)
}

const                      property_shim                      =
Object.getOwnPropertyDescriptor(Bar.prototype,
'property').get;
export function __wbg_s_Bar_property(ptr) {
  return property_shim.call(getObject(ptr));
}

const                    set_property_shim                    =
Object.getOwnPropertyDescriptor(Bar.prototype,
'property').set;
export function __wbg_s_Bar_set_property(ptr, arg0) {
  set_property_shim.call(getObject(ptr), arg0)
}
```

Like when importing functions from JS we can see a bunch of shims are generated for all the relevant functions. The `new` static function has the `#[wasm_bindgen(constructor)]` attribute which means that instead of any particular method it should actually invoke the `new` constructor instead (as we see here). The static function `another_function`, however, is dispatched as `Bar.another_function`.

The `get` and `set` functions are methods so they go through `Bar.prototype`, and otherwise their first argument is implicitly the JS object itself which is loaded through `getObject` like we saw earlier.

Some real meat starts to show up though on the Rust side of things, so let's take a look:

```
pub struct Bar {
    obj: JsValue,
}

impl Bar {
    fn new() -> Bar {
        extern "C" {
            fn __wbg_s_Bar_new() -> u32;
        }
        unsafe {
            let ret = __wbg_s_Bar_new();
            Bar { obj: JsValue::__from_idx(ret) }
        }
    }

    fn another_function() -> i32 {
        extern "C" {
            fn __wbg_s_Bar_another_function() -> i32;
        }
        unsafe {
            __wbg_s_Bar_another_function()
        }
    }
```

```
fn get(&self) -> i32 {
    extern "C" {
        fn __wbg_s_Bar_get(ptr: u32) -> i32;
    }
    unsafe {
        let ptr = self.obj.__get_idx();
        let ret = __wbg_s_Bar_get(ptr);
        return ret
    }
}

fn set(&self, val: i32) {
    extern "C" {
        fn __wbg_s_Bar_set(ptr: u32, val: i32);
    }
    unsafe {
        let ptr = self.obj.__get_idx();
        __wbg_s_Bar_set(ptr, val);
    }
}

fn property(&self) -> i32 {
    extern "C" {
        fn __wbg_s_Bar_property(ptr: u32) -> i32;
    }
    unsafe {
        let ptr = self.obj.__get_idx();
        let ret = __wbg_s_Bar_property(ptr);
        return ret
    }
}

fn set_property(&self, val: i32) {
    extern "C" {
        fn __wbg_s_Bar_set_property(ptr: u32, val: i32);
```

```
        }
        unsafe {
            let ptr = self.obj.__get_idx();
            __wbg_s_Bar_set_property(ptr, val);
        }
    }
}

impl WasmBoundary for Bar {
    // ...
}

impl ToRefWasmBoundary for Bar {
    // ...
}
```

In Rust we're seeing that a new type, `Bar`, is generated for this import of a class. The type `Bar` internally contains a `JsValue` as an instance of `Bar` is meant to represent a JS object stored in our module's stack/slab. This then works mostly the same way that we saw JS objects work in the beginning.

When calling `Bar::new` we'll get an index back which is wrapped up in `Bar` (which is itself just a `u32` in memory when stripped down). Each function then passes the index as the first argument and otherwise forwards everything along in Rust.

Rust Type conversions

Previously we've been seeing mostly abridged versions of type conversions when values enter Rust. Here we'll go into some more depth about how this is implemented. There are two categories of traits for converting values, traits for converting values from Rust to JS and traits for the other way around.

From Rust to JS

First up let's take a look at going from Rust to JS:

```
pub trait IntoWasmAbi: WasmDescribe {
    type Abi: WasmAbi;
    fn into_abi(self) -> Self::Abi;
}
```

And that's it! This is actually the only trait needed currently for translating a Rust value to a JS one. There's a few points here:

- We'll get to `WasmDescribe` later in this section.
- The associated type `Abi` is the type of the raw data that we actually want to pass to JS. The bound `WasmAbi` is implemented for primitive types like `u32` and `f64`, which can be represented directly as WebAssembly values, as well of a couple of other types like `WasmSlice`:

  ```
  pub struct WasmSlice {
      pub ptr: u32,
      pub len: u32,
  }
  ```

  This struct, which is how things like strings are represented in FFI, isn't a WebAssembly primitive type, and so it can't be mapped directly to a WebAssembly parameter / return value. This is why `WasmAbi` lets types specify how they can be split up into multiple WebAssembly parameters:

  ```
  impl WasmAbi for WasmSlice {
      fn split(self) -> (u32, u32, (), ()) {
          (self.ptr, self.len, (), ())
      }

      // some other details to specify return type of
  `split`, go in the other direction
  }
  ```

This means that a `WasmSlice` gets split up into two `u32` parameters. The extra unit types on the end are there because Rust doesn't let us make `WasmAbi` generic over variable-length tuples, so we just take tuples of 4 elements. The unit types still end up getting passed to/from JS, but the C ABI just completely ignores them and doesn't generate any arguments.

Since we can't return multiple values, when returning a `WasmSlice` we instead put the two `u32`s into a `#[repr(C)]` struct and return that.

- And finally we have the `into_abi` function, returning the `Abi` associated type which will be actually passed to JS.

This trait is implemented for all types that can be converted to JS and is unconditionally used during codegen. For example you'll often see `IntoWasmAbi for Foo` but also `IntoWasmAbi for &'a Foo`.

The `IntoWasmAbi` trait is used in two locations. First it's used to convert return values of Rust exported functions to JS. Second it's used to convert the Rust arguments of JS functions imported to Rust.

From JS to Rust

Unfortunately the opposite direction from above, going from JS to Rust, is a bit more complicated. Here we've got three traits:

```
pub trait FromWasmAbi: WasmDescribe {
    type Abi: WasmAbi;
    unsafe fn from_abi(js: Self::Abi) -> Self;
}

pub trait RefFromWasmAbi: WasmDescribe {
    type Abi: WasmAbi;
    type Anchor: Deref<Target=Self>;
    unsafe fn ref_from_abi(js: Self::Abi) -> Self::Anchor;
}

pub trait RefMutFromWasmAbi: WasmDescribe {
    type Abi: WasmAbi;
    type Anchor: DerefMut<Target=Self>;
    unsafe fn ref_mut_from_abi(js: Self::Abi) -> Self::Anchor;
}
```

The `FromWasmAbi` is relatively straightforward, basically the opposite of `IntoWasmAbi`. It takes the ABI argument (typically the same as `IntoWasmAbi::Abi`) to produce an instance of `Self`. This trait is implemented primarily for types that *don't* have internal lifetimes or are references.

The latter two traits here are mostly the same, and are intended for generating references (both shared and mutable references). They look almost the same as `FromWasmAbi` except that they return an `Anchor` type which implements a `Deref` trait rather than `Self`.

The `Ref*` traits allow having arguments in functions that are references rather than bare types, for example `&str`, `&JsValue`, or `&[u8]`. The `Anchor` here is required to ensure that the lifetimes don't persist beyond one function call and remain anonymous.

The `From*` family of traits are used for converting the Rust arguments in Rust exported functions to JS. They are also used for the return value in JS functions imported into Rust.

Communicating types to `wasm-bindgen`

The last aspect to talk about when converting Rust/JS types amongst one another is how this information is actually communicated. The `#[wasm_bindgen]` macro is running over the syntactical (unresolved) structure of the Rust code and is then responsible for generating information that `wasm-bindgen` the CLI tool later reads.

To accomplish this a slightly unconventional approach is taken. Static information about the structure of the Rust code is serialized via JSON (currently) to a custom section of the Wasm executable. Other information, like what the types actually are, unfortunately isn't known until later in the compiler due to things like associated type projections and typedefs. It also turns out that we want to convey "rich" types like `FnMut(String, Foo, &JsValue)` to the `wasm-bindgen` CLI, and handling all this is pretty tricky!

To solve this issue the `#[wasm_bindgen]` macro generates **executable functions** which "describe the type signature of an import or export". These executable functions are what the `WasmDescribe` trait is all about:

```
pub trait WasmDescribe {
    fn describe();
}
```

While deceptively simple this trait is actually quite important. When you write, an export like this:

```
#[wasm_bindgen]
fn greet(a: &str) {
    // ...
}
```

In addition to the shims we talked about above which JS generates the macro *also* generates something like:

```
#[no_mangle]
pub extern "C" fn __wbindgen_describe_greet() {
    <dyn Fn(&str)>::describe();
}
```

Or in other words it generates invocations of `describe` functions. In doing so the `__wbindgen_describe_greet` shim is a programmatic description of the type layouts of an import/export. These are then executed when `wasm-bindgen` runs! These executions rely on an import called `__wbindgen_describe` which passes one `u32` to the host, and when called multiple times gives a `Vec<u32>` effectively. This `Vec<u32>` can then be reparsed into an `enum Descriptor` which fully describes a type.

All in all this is a bit roundabout but shouldn't have any impact on the generated code or runtime at all. All these descriptor functions are pruned from the emitted Wasm file.

`js-sys`

The `js-sys` crate provides bindings to all the global APIs guaranteed to exist in every JavaScript environment by the ECMAScript standard.

For documentation on js-sys types and their generic type parameters, see the js-sys reference.

For API documentation, see docs.rs.

Testing

See the main Testing page for information on testing wasm-bindgen crates, including js-sys.

Adding More APIs

The js-sys crate includes bindings for all APIs in the ECMAScript standard.

If you find a missing API or a new API has reached TC39 stage 4, please file an issue.

For documentation on existing js-sys types, see the js-sys reference.

`web-sys`

The `web-sys` crate provides raw bindings to all of the Web's APIs, and its source lives at `wasm-bindgen/crates/web-sys`.

The `web-sys` crate is **entirely** mechanically generated inside `build.rs` using `wasm-bindgen`'s WebIDL frontend and the WebIDL interface definitions for Web APIs. This means that `web-sys` isn't always the most ergonomic crate to use, but it's intended to provide verified and correct bindings to the web platform, and then better interfaces can be iterated on crates.io!

Documentation for the published version of this crate is available on docs.rs but you can also check out the master branch documentation for the crate.

`web-sys` Overview

The `web-sys` crate has this file and directory layout:

```
.
├── build.rs
├── Cargo.toml
├── README.md
├── src
│   └── lib.rs
└── webidls
    └── enabled
        └── ...
```

`webidls/enabled/*.webidl`

These are the WebIDL interfaces that we will actually generate bindings for (or at least bindings for *some* of the things defined in these files).

`build.rs`

The `build.rs` invokes `wasm-bindgen`'s WebIDL frontend on all the WebIDL files in `webidls/enabled`. It writes the resulting bindings into the cargo build's out directory.

`src/lib.rs`

The only thing `src/lib.rs` does is include the bindings generated at compile time in `build.rs`. Here is the whole `src/lib.rs` file:

```
//! Raw API bindings for Web APIs
//!
//! This is a procedurally generated crate from browser WebIDL
which provides a
//! binding to all APIs that browsers provide on the web.
//!
//! This crate by default contains very little when compiled
as almost all of
```

```
//! its exposed APIs are gated by Cargo features. The exhaustive list of
//! features can be found in `crates/web-sys/Cargo.toml`, but the rule of thumb
//! for `web-sys` is that each type has its own cargo feature (named after the
//! type). Using an API requires enabling the features for all types used in the
//! API, and APIs should mention in the documentation what features they
//! require.

#![doc(html_root_url = "https://docs.rs/web-sys/0.3")]
#![no_std]
#![allow(deprecated)]

extern crate alloc;

mod features;
#[allow(unused_imports)]
pub use features::*;

pub use js_sys;
pub use wasm_bindgen;

/// Getter for the `Window` object
///
/// [MDN Documentation]
///
/// *This API requires the following crate features to be activated: `Window`*
///
/// [MDN Documentation]: https://developer.mozilla.org/en-US/docs/Web/API/Window
#[cfg(feature = "Window")]
pub fn window() -> Option<Window> {
```

```
    use wasm_bindgen::JsCast;

    js_sys::global().dyn_into::<Window>().ok()
}
```

Cargo features

When compiled the crate is almost empty by default, which probably isn't what you want! Due to the very large number of APIs, this crate uses features to enable portions of its API to reduce compile times. The list of features in `Cargo.toml` all correspond to types in the generated functions. Enabling a feature enables that type. All methods should indicate what features need to be activated to use the method.

Testing

You can test the `web-sys` crate by running `cargo test` within the `crates/web-sys` directory in the `wasm-bindgen` repository:

```
cd wasm-bindgen/crates/web-sys
cargo test --target wasm32-unknown-unknown --all-features
```

The Wasm tests all run within a headless browser. See the `wasm-bindgen-test` crate's `README.md` for details and configuring which headless browser is used.

Logging

The `wasm_bindgen_webidl` crate (used by `web-sys`'s `build.rs`) uses `env logger` for logging, which can be enabled by setting the `RUST_LOG=wasm_bindgen_webidl` environment variable while building the `web-sys` crate.

Make sure to enable "very verbose" output during `cargo build` to see these logs within `web-sys`'s build script output.

```
cd crates/web-sys
RUST_LOG=wasm_bindgen_webidl cargo build -vv
```

If `wasm_bindgen_webidl` encounters WebIDL constructs that it doesn't know how to translate into `wasm-bindgen` AST items, it will emit warn-level logs.

```
WARN 2018-07-06T18:21:49Z: wasm_bindgen_webidl: Unsupported
WebIDL interface: ...
```

Supporting More Web APIs in `web-sys`

1. Ensure that the `.webidl` file describing the interface exists somewhere within the `crates/web-sys/webidls/enabled` directory.

   First, check to see whether we have the WebIDL definition file for your API:

   ```
   grep -rn MyWebApi crates/web-sys/webidls
   ```

   - If your interface is defined in a `.webidl` file that is inside the `crates/web-sys/webidls/enabled` directory, skip to step (3).
   - If your interface isn't defined in any file yet, find the WebIDL definition in the relevant standard and add it as a new `.webidl` file in `crates/web-sys/webidls/enabled`. Make sure that it is a standard Web API! We don't want to add non-standard APIs to this crate.
   - If your interface is defined in a `.webidl` file within any of the `crates/web-sys/webidls/unavailable_*` directories, you need to move it into `crates/web-sys/webidls/enabled`, e.g.:

     ```
     cd crates/web-sys
     git mv webidls/unavailable_enum_ident/MyWebApi.webidl
     webidls/enabled/MyWebApi.webidl
     ```

2. Regenerate the `web-sys` crate auto-generated bindings, which you can do with the following commands:

   ```
   cd crates/web-sys
   cargo run --release --package wasm-bindgen-webidl --
   webidls src/features ./Cargo.toml
   ```

   You can then use `git diff` to ensure the bindings look correct.

Publishing New `wasm-bindgen` Releases

1. ☐ Compile the `publish.rs` script:

   ```
   rustc publish.rs
   ```

2. ☐ Bump every crate's minor version:

   ```
   # Make sure you are in the root of the wasm-bindgen repo!
   ./publish bump
   ```

3. ☐ Update CHANGELOG.md to add the to-be-released version number, compare URL and release date. See this example

4. ☐ Send a pull request for the version bump.

5. ☐ After the pull request's CI is green and it has been merged, publish to cargo:

   ```
   # Make sure you are in the root of the wasm-bindgen repo!
   ./publish publish
   ```

Governance of the `wasm-bindgen` Organization

This document describes the lightweight governance structure used in the `wasm-bindgen` organization.

- Code of Conduct
- Membership
    - Core Team
- Changes Policy

Code of Conduct

We abide by the Rust Code of Conduct and ask that you do as well, with the moderation role performed by the [core team].

Membership

We welcome new collaborators in the project! The project is not owned by any specific entity or group and belongs to the community as a whole.

If you make two or three significant contributions, you should seriously consider requesting collaborator membership.

If you are a collaborator and notice that someone has made two or three significant contributions to it, you should also seriously consider nominating them for collaborator status!

To request to join as a Collaborator, post a collaborator request issue stating your intent to join the project, and it will be reviewed by other members.

Core Team

The core team is responsible for project governance and moderation, operating based on a majority vote.

The project as a whole seeks to operate on consensus, such that core votes are typically only needed in the event of governance changes and process escalations.

Core team members are themselves selected by a core team vote, and are required to maintain active participation as core members. Active participation is currently defined as regular attendance at monthly core meetings.

Changes Policy

When merging code changes, the following PR policy applies:

- We operate on a full consensus contribution model based on always assuming good faith.
- All collaborators, including core team members, have equal contribution rights.
- All pull requests must be reviewed and approved of by at least one other collaborator, and ensuring adequate time for reviews before merging.
- All explicit requests for changes on PRs by other collaborators must be resolved prior to merging.
- In case of disagreements, and as a last resort when all other consensus-building methods have been exhausted, escalation to a core team vote may be initiated for the next core team meeting.

Team

`wasm-bindgen` follows the project governance described here.

The current project team members are listed below.

We welcome new members - if you are interested in joining the team, see the membership section of the governance document.

Core

- daxpedda
- Ingvar Stepanyan
- Guy Bedford

Collaborators

- Hood Chatham
- Dan Carney
- Spxg
- Drew Crawford
- Magic Akari
- Logan Gatlin
- Jesper Håkansson

Former Members

`wasm-bindgen` was initially developed under the now sunsetted Rust and WebAssembly Working Group, which included these former team members.