Proxy

An object that controls access to another object, called the **subject**.

- The proxy and the subject have an identical interface, and this allows us to swap one for the other transparently;
- Alternative pattern is **surrogate**.

A **proxy** intercepts all or some of the operations that are meant to be executed on the subject, augmenting or complementing their behavior.

The proxy and the subject have the same interface, and how this is transparent to the client, who can use one or the other interchangeably.

The proxy forwards each operation to the subject, enhancing its behavior with additional preprocessing or postprocessing.

※ **Proxy is not between classes; the Proxy pattern involves wrapping an actual instance of the subject, thus preserving its internal state**

Circumstances

- Data Validation :
  - The proxy validates the input before forwarding it to the subject
- Security :
  - The proxy verifies that the client is authorized to perform operation, and it passes the request to the subject only if the outcome of the check if positive
- Caching :
  - The proxy keeps an internal cache so that the proxied operations are executed on the subject only if the data is not yet present in the cache
- Lazy initialization :
  - If creating the subject is expensive, the proxy can delay it until it's really necessary
- Logging :
  - The proxy intercepts the method invocations and the relative parameters, recoding them as they happen
- Remote objects :
  - They proxy can take a remote object and make it appear local

Techniques for implementing proxies

When *proxying* an objcet, we can decide to intercept all of its methods or only some of them, while delegating the rest directly to the subject.

Example

```
class StackCalculator {
  constructor() {
    this.stack = [];
  }

  putValue(value) {
    this.stack.push(value);
  }

  getValue() {
    return this.stack.pop();
  }

  peekValue() {
    return this.stack[this.stack.length - 1];
  }

  clear() {
    this.stack = [];
  }

  divide() {
    const divisor = this.getValue();
    const dividend = this.getValue();
    const result = dividend / divisor;
    this.putValue(result);
    return result;
  }

  multiply() {
    const multiplicand = this.getValue();
    const multiplier = this.getValue();
    const result = multiplier * multiplicand;
    this.putValue(result);
    return result;
  }
}
```

-- This class implements a simplified version of a stack calculator. The idea of this calculator is to keep all operands (values) in a stack. This is not too different from how the calculator applciation on your

mobile phone is actually implemented

Examples

```
const calculator = new StackCalculator();
calculator.putValue(3);
calculator.putValue(3);
console.log(calculator.multiply());
calculator.putValue(2);
console.log(calculator.multiply());
```

Task

Leverage the **Proxy pattern** to enhance a *StackCalculator* instance by providing a more conservative behavior for division by 0 : rather than returning *Infinity*, we will throw an explicit error.

Object composition

Composition is a technique whereby an object is combined with another object for the purpose of the purpose of extending or using functionality.

```
export class SafeCalculator {
  constructor(calculator) {
    this.calculator = calculator;
  }
  // proxied method
  divide() {
    // aditional validation logic
    const divisor = this.calculator.peekValue();
    if (divisor === 0) {
      throw Error("Division by 0");
    }

    // if valid delegates to the subject
    return this.calculator.divide();
  }

  // delegated methods
  putValue(value) {
    return this.calculator.putValue(value);
  }

  getValue() {
    return this.calculator.getValue();
  }

  peekValue() {
    return this.calculator.peekValue();
  }

  clear() {
    return this.calculator.clear();
  }

  multiply() {
    return this.calculator.multiply();
  }
}
```

The *safeCalculator* object is a proxy for the original *calculator* instance. To implement this proxy using componsition, we had to intercept the methods that we were interested in manipulating **divide()** while simply delegating the rest of them to the subject. (**putValue(), getValue(), peekValue(), clear(), and multiply()**)

An alternative implementation of the proxy presented in the preceding code fragment might just use an object literal and a factory function.

```
// Alternative implementation of the proxy
export function createSafeCalculator(calculator) {
  return {
    // proxied method
    divide() {
      // additional validation logic
      const divisor = calculator.peekValue();
      if (divisor === 0) {
        throw Error("Division by 0");
      }
      // if valid delegates to the subject
      return calculator.divide();
    },
    // delegated methods
    putValue(value) {
      return calculator.putValue();
    },
    getValue() {
      return calculator.getValue();
    },
    peekValue() {
      return calculator.peekValue();
    },
    clear() {
      return calculator.clear();
    },
    multiply() {
      return calculator.multiply();
    },
  };
}
```

This implementation is simpler and more concise than the class-based one, but once again, it forces us to delegate all the methods to the subject explicitly.

※ Having to delegate many methods for complex classes can be very tedious and might make it harder to implement these techniques. One way to create a proxy that delegates most of its methods is to use a library that generates all the methods for us, such as *delegates* (nodejsdp.link/delegates).

※ A more modern and native alternative is to use the *Proxy* object, which we will discuss later in this chapter.

Object augmentation

Object augmentation (or **monkey patching**) is the simplest and the most common way of proxying just a few methods of an object. It involes modifying the subject directly by replacing a method with its

proxied implementation.

In the context of our calculator example, this could be done as follows:

```
function patchToSafeCalculator(calculator) {
  const divideOrig = calculator.divide;
  calculator.divide = () => {
    //additional validation logic
    const divisor = calculator.peekValue();
    if (divisor === 0) {
      throw Error("Division by 0");
    }
    // if valid, delegates to the subject
    return divideOrig.apply(calculator);
  };

  return calculator;
}

const calculator = new StackCalculator();
const safeCalculator = patchToSafeCalculator(calculator);
//...
```

This technique is defintely convenient when we need to proxy only one or a few methods. Plus, we do not need to implement the *multiply()* method and all other delegated methods.

Unfortunately, simplicity comes at the cost of having to mutate the *subject object* directly, which can be dangerous

※ Mutation should be avoided at all costs when the subject is shared with other parts of the codebase. "Monkey patching" the subject might create undesirable side effects that affect other components of our application. The problem is that the *calculator* instance (not safeCalculator) will also throw an error when diving by zero rather than returning *Infinity*

The build-in Proxy object

The ES2015 specification introduced a native way to create powerful proxy objects.

```
const proxy = new Proxy(target, handler);
```

Target - represents the object on which the proxy is applied (the **subject** for our canonical definition).

Handler - A special object that defines the behavior of the proxy.

※ The *handler* object contains a series of optional methods with predefined names called **trap methods** (for example, *apply, get, set,* and *has*) that are automatically called when the corresponding operations are performed on the proxy instance.

```
import { StackCalculator } from "./stackCalculator.js";

const safeCalculatorHandler = {
  get: (target, property) => {
    if (property === "divide") {
      // proxied method
      return function () {
        // additional validation logic
        const divisor = target.peekValue();
        if (divisor === 0) {
          throw Error("Division by 0");
        }
        // if valid delegates to the subject
        return target.divide();
      };
    }

    // delegate methods and properties
    return target[property];
  },
};

const calculator = new StackCalculator();
const safeCalculator = new Proxy(calculator, safeCalculatorHandler);
```

※ The **Proxy** object inherits the prototype of the subject, therefore running `safeCalculator instanceof StackCalculator` will return true.

The *Proxy* object allows us to avoid mutating the subject while giving us an easy way to proxy only the bits that we need to enhance, without having to explicitly delegate all the other properties and mthods.

Additional capabilities and limitations of the *Proxy* object

The *Proxy* object is a feature deeply integrated into the JavaScript language itself, which enables developers to intercept and customize many operations that can be performed on objects.

This characteristic opens up new and interesting scenarios that were not easily achievable before, such as *meta programming, operator overloading,* and *object virtualization.*

Example

```
const evenNumbers = new Proxy([], {
  get: (target, index) => index * 2,
  has: (target, number) => number % 2 === 0,
});

console.log(2 in evenNumbers); // true
console.log(5 in evenNumbers); // false
console.log(evenNumbers[7]); // 14
```

In this example, we are creating a virtual array that contains all even numbers. The array is considered *virtual* because we never store data in it.

The *Proxy* object supports several other interesting traps such as *set, delete,* and *constructs*, and allows us to create proxies that can be revoked on demand, disabling all the traps and restoring the orinal behavior of the target object.

While the **Proxy** object is a powerful functionality of the JavaScript language, it suffers from a very important limitation: the Proxy object cannot be fully *transpiled* or *polyfilled*. This is because some of the Proxy object traps can be implemented only at the runtime level and cannot be simply rewritten in plain JavaScript. This is something to be aware of if you are working with old broswers or old versions of Node.js that don't support the Proxy object directly.

Transpilation

- Short for transcompilation
- The action of compiling source code by translating it from one source programming language to another.
  - In the case of Javascript, this technique is used to convert a program using new capabilities of the language into an equivalent program that can also run on older runtimes that do not support these new capabilities.

Polyfill

- Code that provides an implementation for a standard API in plain JavaScript
- can be imorted in environments where this API is not available
  (generally older broswers or runtimes).
  - core-js (nodejsdp.link/corejs) is one of the most complete polyfill libraries for JavaScript

A comparison of the different proxying techniques

Composition

Composition can be considered a simple and *safe* way of creating a proxy because it leaves the subject untouched without mutating its original behavior. Its only drawback is that we have to manually delegate all the methods, even if we want to proxy only few of them.

> Object properties can be delegated using **Object.defineProperty()**.

Augmentation

Modifies the subject, which might not always be ideal, but it does not suffer from the various inconveniences relatd to delegation. For this reason, between two approaches, object augmentation is generally the preferred technique in all those circumstances in which modifying the subject is an option.

> There is at least one situation where composition is almost necessary; this is when we want to initialization of the subject, for example, to create it only when needed.(*lazy initialization*)

Proxy object

Is the go-to approach if you need to intercept function calls or have different types of access to object attributes, even dynamic ones. The Proxy object provides an advanced level of access control that is simply not available with the other techniques.

The Proxy object does not mutate the subject, so it can be safely used in contexts where the subject is shared between different components of the application.

Creating a logging Writable stream

```
export function createLoggingWritable(writable) {
  return new Proxy(writable, {
    get(target, propKey, receiver) {
      if (propKey === "write") {
        return function (...args) {
          const [chunk] = args;
          console.log("Writing", chunk);
          return writable.write(...args);
        };
      }

      return target[propKey];
    },
  });
}
```

We will build an object that acts as a proxy to a **Writable** stream, which intercepts all the calls to the **write()** method and logs a message every time this happens.

We created a factory that returns a proxied version of the *writable* object passed as an argument.

The proxy did not change the original interface of the stream or its external behavior but if we run the preceding code, we will not see that every chunk that is written into the **writableProxy** stream is transparently logged to the console.

Change observer with Proxy

The **Change Observer pattern** is a design pattern in which an object (the subject) notifies one or more observers of any state changes, so that they can "react" to changes as soon as they happen.

> The **Change Observer** pattern should not be confused with the **Observer pattern**. The **Change Observer** pattern focuses on allowing the detection of property changes, while the **Observer pattern** is a more genertic pattern that adopts an event emitter to propagate information about events happening in the system.

Proxy is an effective tool to create observable objects.

```
export function createObservable(target, observer) {
  const observable = new Proxy(target, {
    set(obj, prop, value) {
      if (value !== obj[prop]) {
        const prev = obj[prop];
        obj[prop] = value;
        observer({ prop, prev, curr: value });
      }
      return true;
    },
  });
  return observable;
}
```

createObservable() accepts a target object (the object to observe for changes) and an observer (a function to invoke every time a change is detected.)
The proxy implements the **set** trap, which is triggered every time a property is set. The implementation compares the current value with the new one and, if they are different, the target object is mutated, and the observer gets notified.

```
import { createObservable } from "./create-observable.js";

function calculateTotal(invoice) {
  return invoice.subtotal - invoice.discount + invoice.tax;
}

const invoice = {
  subtotal: 100,
  discount: 10,
  tax: 20,
};

let total = calculateTotal(invoice);
console.log(`Starting total: ${total}`);

const obsInvoice = createObservable(invoice, ({ prop, prev, curr }) => {
  total = calculateTotal(invoice);
  console.log(`TOTAL: ${total} (${prop} changed: ${prev} -> ${curr})`);
});

obsInvoice.subtotal = 200; // TOTAL : 210
obsInvoice.discount = 20; // TOTAL : 200
obsInvoice.discount = 20; // no change: doesn't notify
obsInvoice.tax = 30; // TOTAL : 210

console.log(`Final total: ${total}`);
```

> **Observable** are the cornerstone of **reactive programming (RP)** and **functional reactive programming (FRP)**. More about these styles of programming nodejsdp.link/reactive-manifesto

In the wild

The **Proxy pattern** and more specifically the **Change Observer** pattern are widely adopted patterns. Some popular projects that take advantage of these patterns

- LoopBack
  - A popular Node.js web framework that usese the Proxy pattern to provide the capability to intercept and enhance method calls on controllers. This capability can be used to build custom validation or authentication mechanisms.
- Vue.js Version 3
  - JavaScript reactive UI framework which has reimplemented observable properties using Proxy pattern with the Proxy object.
- MobX
  - Reactive state management library commonly used in frontend applications in combination with React or Vue.js.