Patrones

Introducción

- Patrones son técnicas y arquitecturas más o menos estandarizadas
- Que aportan un lenguaje común
- Y un catálogo de soluciones a problemas habituales

Patrones Básicos

Singleton

- Una clase que sólo se puede instanciar una vez
- Todos los consumidores comparten la misma instancia

Singleton

```
class ContactsAPI {

  getContacts() { /* ... */ }

  addNewContact() { /* ... */ }

}

const contactsApi = ContactsAPI.getInstance();
contactsApi.getContacts();
```

Singleton

```
class ContactsAPI {

  getContacts() { /* ... */ }

  addNewContact() { /* ... */ }

}

const contactsApi = ContactsAPI.getInstance();
contactsApi.getContacts();
```

Singleton

¿Cómo podríamos implementar ese método

`getInstance`?

Singleton

```
ContactsAPI.getInstance = (() => {
  let instance = null;
  return (...args) => {
    if (instance === null) {
      instance = new ContactsAPI(...args);
    }
    return instance;
  }
})();
```

Singleton

Es un caso perfecto para escribir un decorador.

¿Cómo sería el decorador **`@singleton`**?

Singleton

```
function singleton(Constructor) {
  let instance = null;
  Constructor.getInstance = (...args) => {
    if (instance === null) {
      instance = new Constructor(...args);
    }
    return instance;
  }
  return Constructor;
}
```

Singleton

```
@singleton
class ContactsAPI {

  getContacts() { /* ... */ }

  addNewContact() { /* ... */ }

}

const contactsApi = ContactsAPI.getInstance();
contactsApi.getContacts();
```

Singleton

```
@singleton
class ContactsAPI {

  getContacts() { /* ... */ }

  addNewContact() { /* ... */ }

}

const contactsApi = ContactsAPI.getInstance();
contactsApi.getContacts();
```

Singleton

- Se utiliza para...
  - Almacenar estado (configuración, credenciales, ...)
  - Comunicar diferentes partes de la aplicación
  - Encapsular APIs y servicios en los que no tiene sentido que haya más de una instancia

Factoría

- Un método que se encarga de crear instancias
- Controlar el instanciado
  - Limitar el número de instancias
  - Automatizar la configuración
  - Decidir qué clase instanciar

Factoría

- El método **getInstance** es una factoría

Factoría

```
class User {

  constructor(name) {
    this.name = name
  }

  setWriteAccess(canWrite) {
    this.canWrite = canWrite
  }

}
```

Factoría

```
User.createReader = (name) => {
  const user = new User(name)
  user.setWriteAccess(false)
  return user
}

User.createEditor = (name) => {
  const user = new User(name)
  user.setWriteAccess(true)
  return user
}
```

Factoría

```
class Particle { /* ... */ }

Particle._free = []
Particle._count = 0
Particle._max = 100

Particle.requestInstance = () => {
  if (Particle._count < Particle._max) {
    Particle._count++
    return new Particle()
  } else if (Particle._free.length > 0) {
    return Particle._free.pop()
  }
  throw new Error('Particle limit reached!')
}

Particle.freeInstance = (instance) => {
  Particle._free.push(instance)
}
```

Factoría

Lanzar una excepción al agotar el recurso no es la solución más práctica...

Factoría

- Ejercicio: Convierte **requestInstance** en un método asíncrono que acepte un callback...
  - Si hay recursos disponibles, ejecuta el callback inmediatamente
  - Si no, ejecutará el callback cuando algún recurso sea liberado

Factoría

```
Particle.requestInstance((particle) => {
  /* .. */
})
```

redradix

Factoría

¿Podrías hacer un decorador que convirtiera cualquier clase en un ***pool*** de instancias reutilizables?

Factoría

```
@pool(100)
class Particle {
  /* ... */
}
```

redradix

Factoría

- En resumen, usa factorías cuando...
  - Tengas clases con configuraciones complejas
  - Quieras controlar el instanciado de un recurso
  - Necesites decidir qué clase instanciar en tiempo de ejecución

Patrones de Comportamiento

Observer

- Un objeto notifica a otro los cambios de su estado
- Hay dos actores:
  - Emisor, que emite **eventos** a uno o varios **suscriptores**
  - Receptor, que se **suscribe** para ser notificado de los eventos que emitan uno o varios emisores

Observer

- Es un patrón **FUNDAMENTAL** en js
  - Toda la **interacción** está basada en eventos
  - Todos los mecanismos de **asincronía** están basados en eventos
  - Es crucial para un lenguaje que tiene *una sola hebra*

Observer

- Hay muchas implementaciones distintas...
  - DOM
  - Node.js
  - jQuery
  - ...

Observer

```
const button = document.getElementById('subscribe');

button.addEventListener('click', (e) => {
  alert('Thanks for subscribing')
})
```

Observer

```
const button = document.getElementById('subscribe');

button.addEventListener('click', (e) => {
  alert('Thanks for subscribing')
})
```

Observer

```
const button = document.getElementById('subscribe');

button.addEventListener('click', (e) => {
  alert('Thanks for subscribing')
})
```

Observer

```
const button = document.getElementById('subscribe');

button.addEventListener('click', (e) => {
  alert('Thanks for subscribing')
})
```

Observer

- Un mismo evento...
  - Se puede emitir muchas veces
  - Se puede emitir a varios receptores

Observer

```
const sting = new MyEventEmitter()

const frodo = new Hobbit('Frodo')
const sam = new Hobbit('Sam')

sting.on('glow', () => {
  frodo.say('There are orcs nearby!')
})

sting.on('glow', () => {
  sam.say('Run, Mr. Frodo!')
})
```

Observer

- Un suscriptor se puede **dar de baja** de un evento para dejar de recibir notificación

Observer

```
const warning = () => {
  console.log('Watch out!')
}

sting.on('glow', warning)

sting.emit('glow') // -> 'Watch out!'

sting.off('glow', warning)

sting.emit('glow') // -> (no output)
```

Observer

```
const warning = () => {
  console.log('Watch out!')
}

sting.on('glow', warning)

sting.emit('glow') // -> 'Watch out!'

sting.off('glow', warning)

sting.emit('glow') // -> (no output)
```

Observer

- Normalmente se puede heredar de la clase que implementa el observador

Observer

```
class Dog extends Observable {
  /* ... */
}

const toi = new Dog()

toi.on('sit', () => {
  console.log('Good boy!')
  toi.pet()
})
```

Observer

- Ejercicio: implementa la clase **`Observable`**
  - con tres métodos:
    - `.on(event, subscriber)`
    - `.off(event, subscriber)`
    - `.emit(event, param1, param2, …)`

Comando

- Nos permite representar y modelar “acciones”
  - En resumen: invocación indirecta
  - Control de la invocación
    - Convertirla en algo manejable (datos)
    - Para guardarla, modificarla, repetirla, etc...
    - Desacoplamiento

Comando

- En lugar de invocar la funcionalidad que necesitamos directamente...
- Vamos a representarla como un objeto
- Y delegar su ejecución en un intermediario

Comando

```
import userApi from 'repository/server/api/user'

userApi.saveUserData({ name: 'John'  })
```

Comando

```
import userApi from 'repository/server/api/user'

userApi.saveUserData({ name: 'John'  })
```

Comando

```
import userApi from 'repository/server/api/user'

userApi.saveUserData({ name: 'John'  })
```

redradix

Comando

```
import dispatchAction from 'dispatcher'

dispatchAction({
  type: 'SAVE_USER_DATA',
  payload: { name: 'John' }
})
```

Comando

```
import dispatchAction from 'dispatcher'
```

```
dispatchAction({
  type: 'SAVE_USER_DATA',
  payload: { name: 'John' }
})
```

Comando

```
import dispatchAction from 'dispatcher'


dispatchAction({
  type: 'SAVE_USER_DATA',
  payload: { name: 'John' }
})
```

redradix

Comando

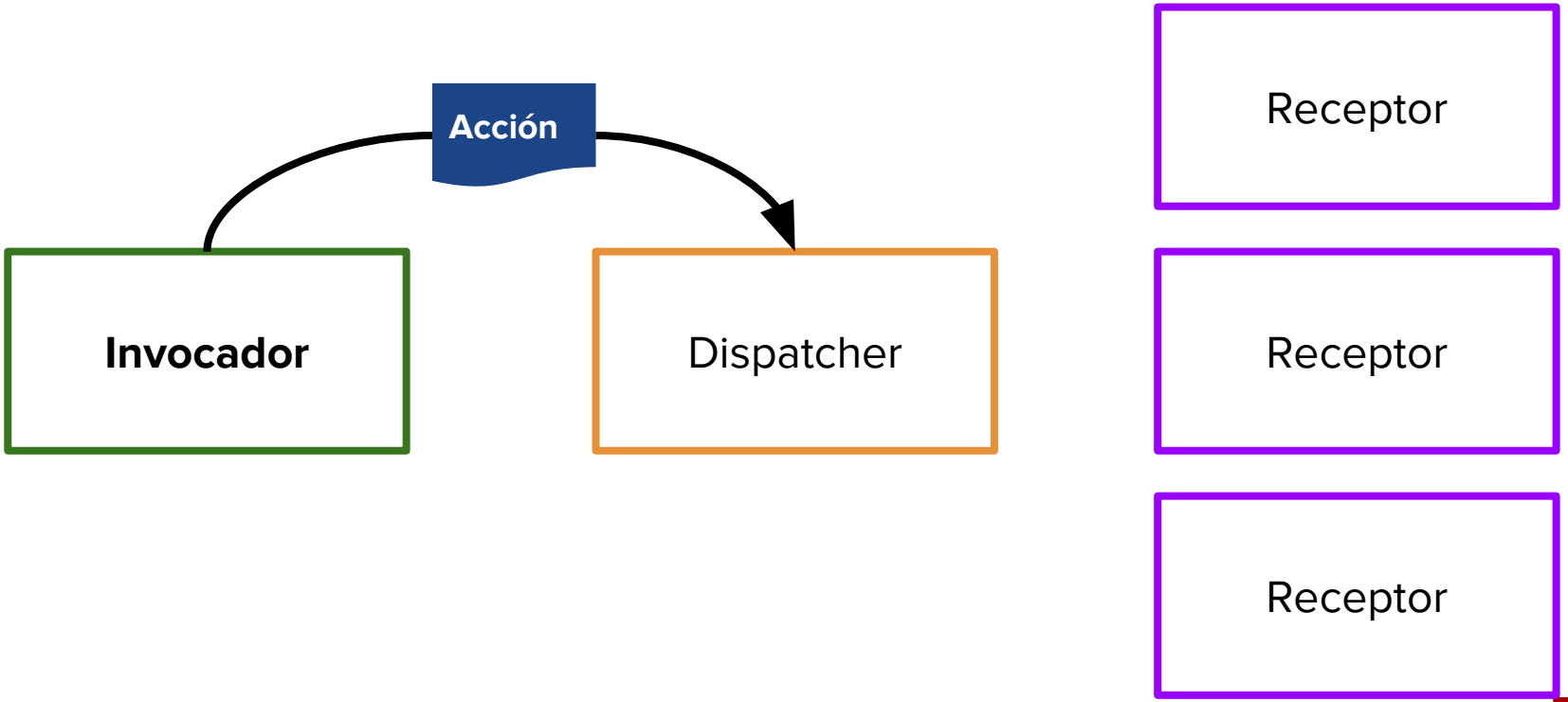
```
import dispatchAction from 'dispatcher'

dispatchAction {
  type: 'SAVE_USER_DATA',
  payload: { name: 'John' }
})
```

Comando

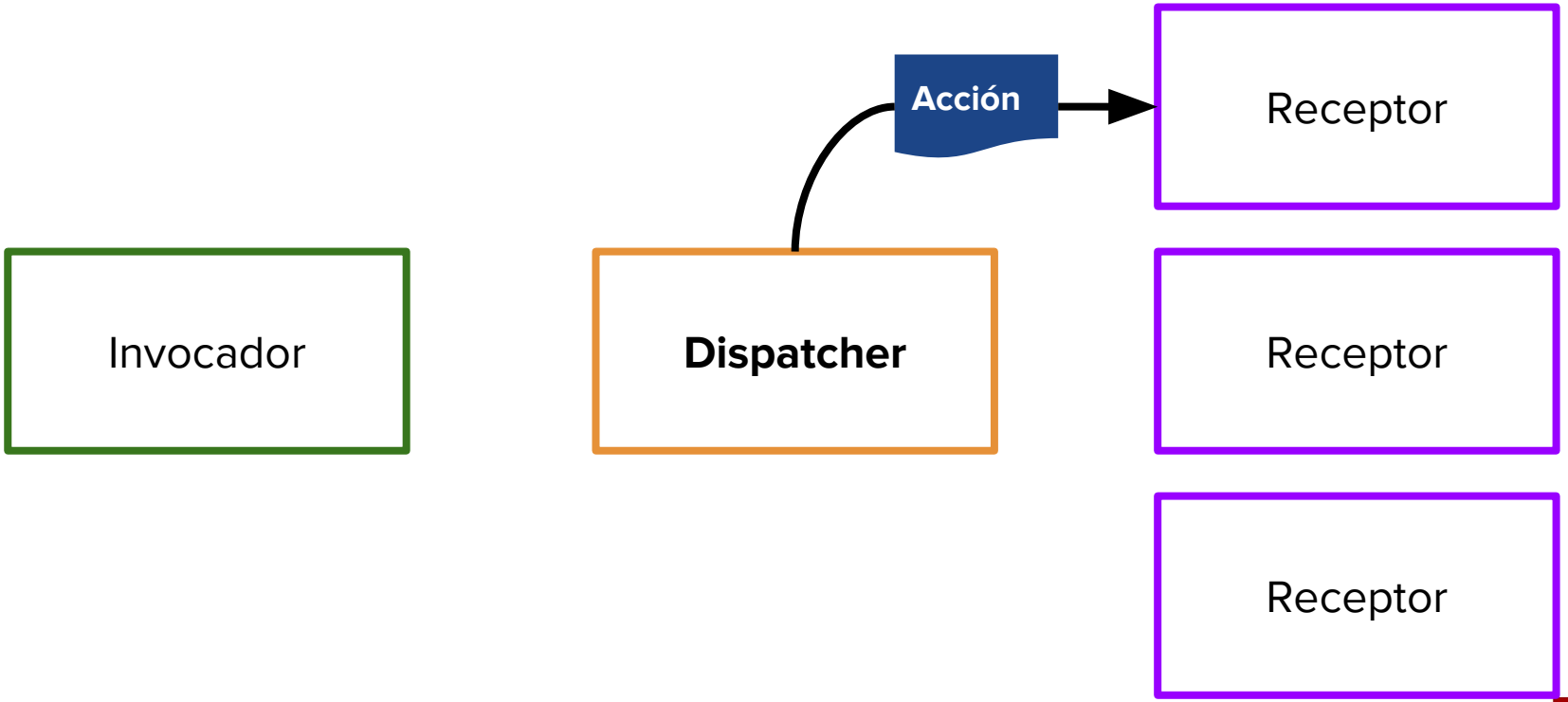
```
import dispatchAction from 'dispatcher'

dispatchAction({
  type: 'SAVE_USER_DATA',
  payload: { name: 'John' }
})
```

Comando

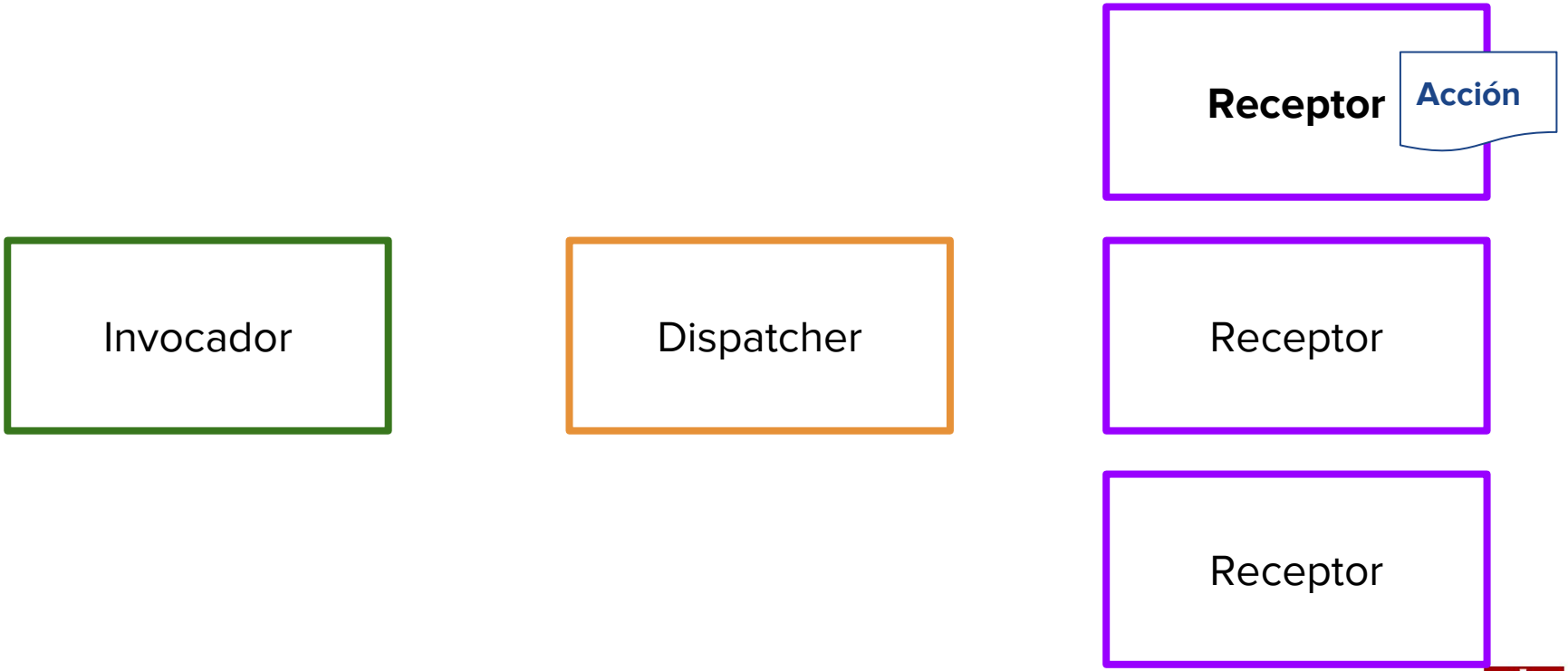
Comando



Comando



Comando



Comando

- Es un patrón muy común
  - Sincronización de operaciones entre máquinas

    (editor colaborativo, juegos en red, monitorización)
  - Botón “deshacer”
  - Control de acceso
  - Logs y repeticiones (MySQL binary log, fifa)

Comando

- Ejercicio: vamos a implementar un dispatcher
  - Los receptores se dan de alta en el dispatcher con un método `register`
  - Todos los receptores reciben todas las acciones
  - Cada receptor decide si es para él o no
  - Las acciones son objetos `{ type, payload }`

```
const dispatcher = Dispatcher.getInstance()

dispatcher.register((action) => {
  switch (action.type) {
      case: 'SAVE_USER_DATA':
      return console.log('Saving user data', action.payload, '...')
  }
})

// in another module...
dispatcher.register((action) => {
  switch (action.type) {
      case: 'LOGOUT':
      return console.log('Good bye.')
  }
})
```

```
dispatcher.dispatch({
  type: 'SAVE_USER_DATA',
  payload: { name: 'John' }
})

dispatcher.dispatch({
  type: 'LOGOUT'
})
```

Redux

Redux

- Una idea independiente del framework que uses
- Simplificación radical de la gestión del estado
- La idea es muy simple...

Redux

- Tengo un **estado central** que almacena todo el estado
- Para alterarlo, hago **dispatch** de **acciones**
- En el estado se registran los **receptores**, que
  - **reciben** las acciones que se dispachean
  - el valor que devuelvan será el **nuevo estado**

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.dispatch({ type: 'INC' })
store.getState() // -> { counter: 1 }

store.dispatch({ type: 'DEC' })
store.getState() // -> { counter: 0 }
```

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.dispatch({ type: 'INC' })
store.getState() // -> { counter: 1 }

store.dispatch({ type: 'DEC' })
store.getState() // -> { counter: 0 }
```

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.dispatch({ type: 'INC' })
store.getState() // -> { counter: 1 }

store.dispatch({ type: 'DEC' })
store.getState() // -> { counter: 0 }
```

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.dispatch({ type: 'INC' })
store.getState() // -> { counter: 1 }

store.dispatch({ type: 'DEC' })
store.getState() // -> { counter: 0 }
```

redradix

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.dispatch({ type: 'INC' })
store.getState() // -> { counter: 1 }

store.dispatch({ type: 'DEC' })
store.getState() // -> { counter: 0 }
```

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.dispatch({ type: 'INC' })
store.getState() // -> { counter: 1 }

store.dispatch({ type: 'DEC' })
store.getState() // -> { counter: 0 }
```

redradix

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.dispatch({ type: 'INC' })
store.getState() // -> { counter: 1 }

store.dispatch({ type: 'DEC' })
store.getState() // -> { counter: 0 }
```

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.dispatch({ type: 'INC' })
store.getState() // -> { counter: 1 }

store.dispatch({ type: 'DEC' })
store.getState() // -> { counter: 0 }
```

Redux

- Ejercicio: implementa la clase **Store** del ejemplo
  - Recibe un receptor como parámetro
    - Que recibe la acción despachada y el estado anterior
    - El retorno del receptor es el nuevo estado
  - Tiene dos métodos:
    - **`dispatch`**
    - **`getState`**

Redux

- Solo con esos dos métodos...
  - Herramienta de gestión de estado
  - No podemos reaccionar ante cambios del estado
  - Necesitamos hacerlo observable!

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.on('change', () => {
  const { counter } = store.getState()
  console.log(`The counter reads: ${counter}`)
})

store.dispatch({ type: 'INC' })
```

```
const initialState = { counter: 0 }

const store = new Store((action, state = initialState) => {
  switch (action.type) {
    case 'INC':
      return { counter: state.counter + 1 }
    case 'DEC':
      return { counter: state.counter - 1 }
  }
})

store.on('change', () => {
  const { counter } = store.getState()
  console.log(`The counter reads: ${counter}`)
})

store.dispatch({ type: 'INC' })
```

Redux

- Ejercicio: modifica la clase **Store** para que emita el evento ‘change’ cada vez que cambie el estado

Redux

- Con los tres métodos...
  - Podemos definir un flujo unidireccional de datos
  - Separar el código en tres partes independientes:
    - lógica de negocio
    - visualización
    - interacción