.UBAfiuba
FACULTAD DE INGENIERÍA

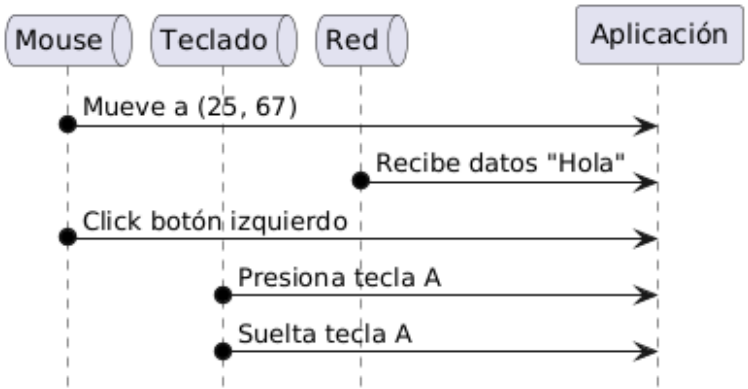
Programación orientada a eventos

TB025 - Paradigmas de Programación

Curso Essaya

Programación orientada a eventos

Es un paradigma de programación en el que el **flujo del programa** es determinado por la ocurrencia de **eventos**, que son previstos pero no planeados (es decir, no se sabe cuándo ocurrirán).



Un ejemplo común es el desarrollo de **interfaces gráficas de usuario (GUI)**. En este caso, se deben manejar eventos como clics de ratón, pulsaciones de teclado, etc. Pero no es el único contexto en el que encuentra aplicación este paradigma.

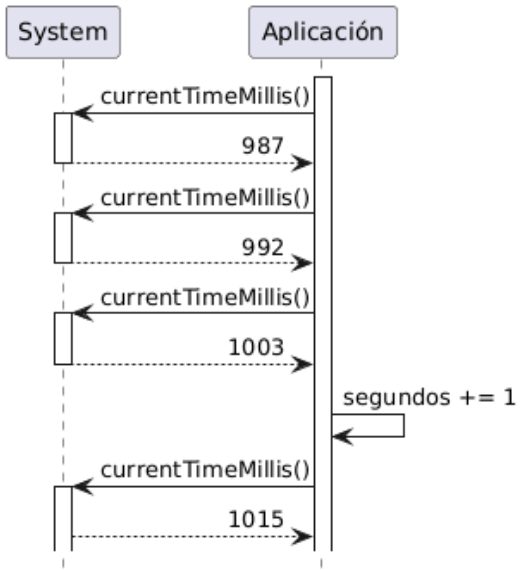
La nomenclatura varía según el framework, lenguaje, etc.:

Objeto que produce eventos: Event source, Event emitter, Event dispatcher, Publisher
Objeto que quiere ser notificado: Listener, Subscriber, Consumer

Estrategia de *polling*

La estrategia más primitiva para procesar eventos es la de **polling** (encuestar).

```
long ultimo = System.currentTimeMillis();
int segundos = 0;
while (true) {
    // pasó 1 segundo desde la última vez?
    long ahora = System.currentTimeMillis();
    if (ahora - ultimo > 1000) {
        segundos += 1;
        System.out.printf("Pasaron %ds.\n", segundos);
        ultimo = ahora;
    }
}
```



Esta estrategia es **sincrónica**: las operaciones ocurren en un orden determinado.

Esta estrategia se conoce también como **busy waiting** o **busy loop**: el encuestador usa recursos del sistema (CPU, energía) mientras espera a que ocurra el evento de interés.

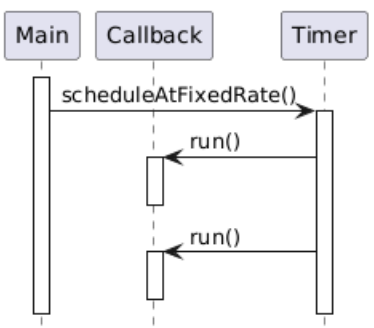
También se dice que es una estrategia de tipo **pull**: el encuestador debe “tirar” para enterarse de los eventos.

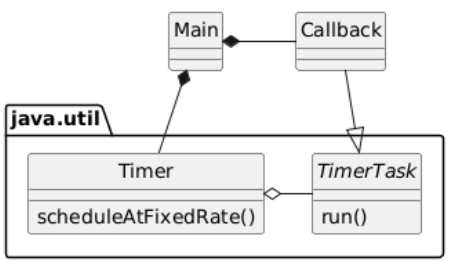
Callbacks

Un **callback** es una función que se pasa como parámetro a otra función (típicamente **no bloqueante**). El callback será invocado en algún momento posterior (es decir, en forma **asincrónica**), por ejemplo cuando ocurre un evento.

Esta estrategia es de tipo **push**: el emisor “empuja” los eventos a los consumidores.

En lenguajes orientados a objetos como Java, el pasaje de callbacks se realiza mediante interfaces o clases abstractas.






```java
import java.util.Timer;
import java.util.TimerTask;

public class Main {
    public static void main(String[] args) {
        Timer timer = new Timer();
        Callback callback = new Callback();
        // callback.run() será invocado cada 1 segundo
        timer.scheduleAtFixedRate(callback, 0, 1000);
        // El mensaje anterior fue no bloqueante
        System.out.println("Timer configurado.");
    }
}

class Callback extends TimerTask {
    int segundos = 0;

    @Override public void run() {
        System.out.printf("Pasaron %ds.\n", segundos++);
    }
}
```

Los callbacks suelen ser llamados **handlers** cuando son utilizados para manejar eventos.

Azúcar sintáctica: clases anónimas

```java
public class Main {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask callback = new Callback();
        // callback.run() será invocado cada 1 segundo
        timer.scheduleAtFixedRate(callback, 0, 1000);
        // El mensaje anterior fue no bloqueante
        System.out.println("Timer configurado.");
    }
}

class Callback extends TimerTask {
    int segundos = 0;

    @Override public void run() {
        System.out.printf("Pasaron %ds.\n", segundos++);
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask callback = new TimerTask() {
            int segundos = 0;

            @Override public void run() {
                System.out.printf("Pasaron %ds.\n", segundos++);
            }
        };
        // callback.run() será invocado cada 1 segundo
        timer.scheduleAtFixedRate(callback, 0, 1000);
        // El mensaje anterior fue no bloqueante
        System.out.println("Timer configurado.");
    }
}
```

Ejemplo: HttpClient

```
Consumer<HttpResponse<String>> callback = new Consumer<>() {
    @Override
    public void accept(HttpResponse<String> response) {
        System.out.println(response.body());
    }
};

HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("https://pokeapi.co/api/v2/pokemon/ditto"))
        .build();
try (HttpClient client = HttpClient.newHttpClient()) {
    client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
            .thenAccept(callback);
    System.out.println("Request sent");
}
```

Azúcar sintáctica: Expresión Lambda

```
Consumer<HttpResponse<String>> callback = new Consumer<>() {
    @Override public void accept(HttpResponse<String> response) {
        System.out.println(response.body());
    }
};
```

```
Consumer<HttpResponse<String>> callback = (HttpResponse<String> response) -> {
    System.out.println(response.body());
};
```

```
Consumer<HttpResponse<String>> callback = (response) -> {
    System.out.println(response.body());
};
```

```
Consumer<HttpResponse<String>> callback = response -> {
    System.out.println(response.body());
};
```

```
Consumer<HttpResponse<String>> callback = response -> System.out.println(response.body());
```

Entrada/Salida bloqueante

Las APIs de I/O son tradicionalmente **sincrónicas** y **bloqueantes**.

Por ejemplo, el siguiente pseudocódigo muestra la implementación más intuitiva de un *socket server*:

```
servidor = crearSocketServidor()
while True:
    # bloquea hasta que un cliente se conecta
    cliente = servidor.accept()

    while True:
        # bloquea hasta que el cliente envía datos o se desconecta
        s = cliente.read()
        if not s:
            # cliente desconectado
            s.close()
            break
        else:
            # envía la respuesta al cliente
            cliente.write(s);
```

La limitación principal de esta implementación es que las funciones `accept` y `read` son **bloqueantes**, y por lo tanto el servidor solo puede atender a un cliente por vez.

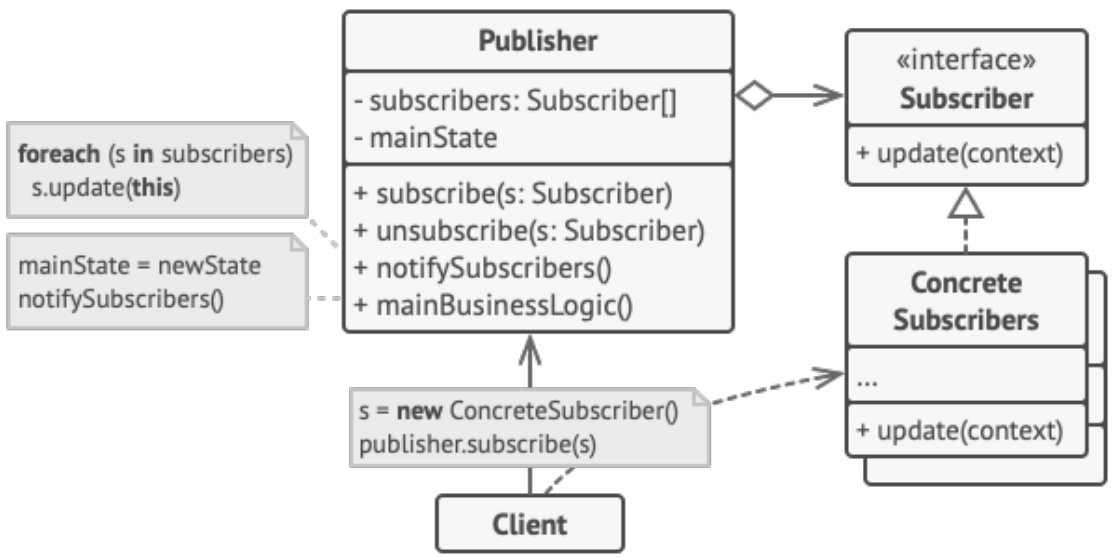
Entrada/Salida multiplexada

La estrategia de **entrada/salida multiplexada** permite bloquear hasta que ocurra algún evento de interés.

```
server = crearServerSocket()
sockets = [server]
while True:
    # bloquea hasta que ocurra algún evento de interés
    subconjunto = select(sockets)
    for socket in subconjunto:
        if type(socket) is ServerSocket:
            # aceptamos la conexion de un cliente
            cliente = server.accept()
            sockets += cliente
        else:
            # recibimos datos de un cliente
            s = socket.read()
            if not s:
                # cliente desconectado
                socket.close()
                sockets.remove(socket)
            else:
                # envía la respuesta al cliente
                socket.write(s);
```

Patrón Observer

Es un patrón de diseño de POO que permite registrar múltiples **callbacks** para ser invocados cuando ocurra un evento. De esta manera se logra una comunicación **uno a muchos** (un emisor, múltiples observadores).



Nota: en esta implementación, la publicación del evento es **sincrónica** con la ejecución de los handlers.

Patrón Observer: Ejemplo

```java
public interface Observador {
    public void nivelGanado();
}

public class Juego {
    private List<Observador> observadores = new ArrayList<>();

    public void suscribir(Observador obs) {
        observadores.add(obs);
    }

    public void notificarNivelGanado() {
        for (Observador obs : observadores) {
            obs.nivelGanado();
        }
    }
}
```

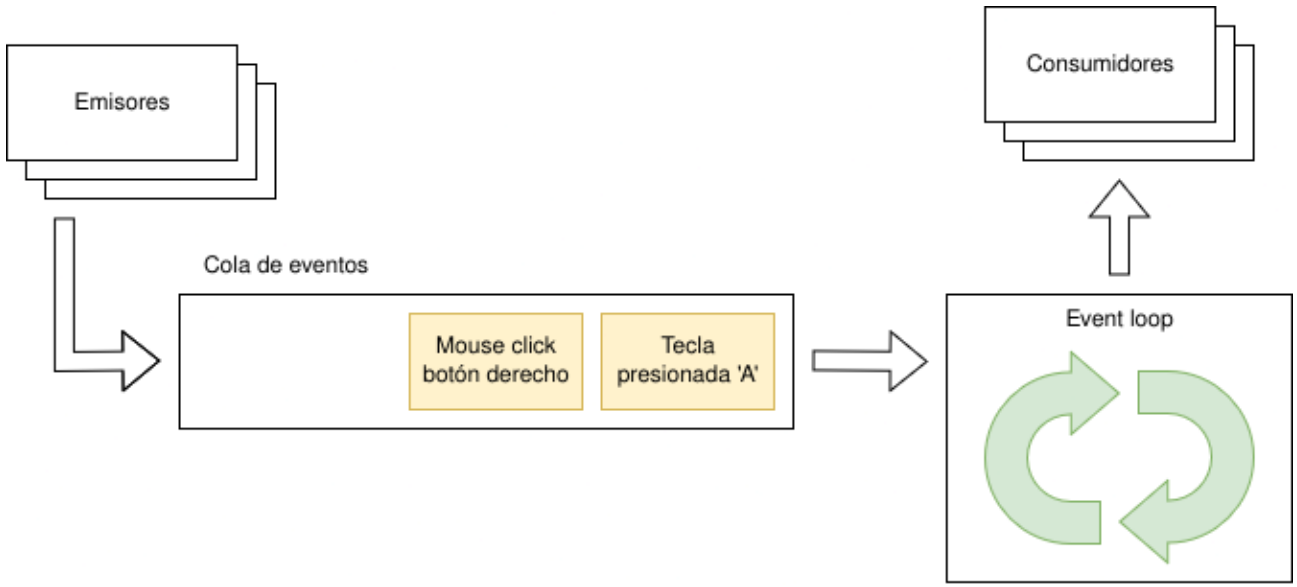
```java
public class Logros implements Observador {
    @Override public void nivelGanado() {
        System.out.println("¡Nuevo logro: Nivel ganado!");
    }
}
```

```java
public class UI implements Observador {
    @Override public void nivelGanado() {
        System.out.println("UI: Nivel ganado.");
    }
}
```

```java
Juego juego = new Juego();
Logros logros = new Logros();
UI ui = new UI();

juego.suscribir(logros);
juego.suscribir(ui);

// Simular que se gana un nivel
juego.notificarNivelGanado();
```

Event loop

Es una estrategia que permite desacoplar aun más los emisores de los consumidores. La publicación de los eventos es **asincrónica** con la invocación de los handlers.




```
function main
    initialize()
    while message != quit
        message := get_next_message()
        process_message(message)
    end while
end function
```

La mayoría de las aplicaciones interactivas modernas se basan en alguna variante de event loop.

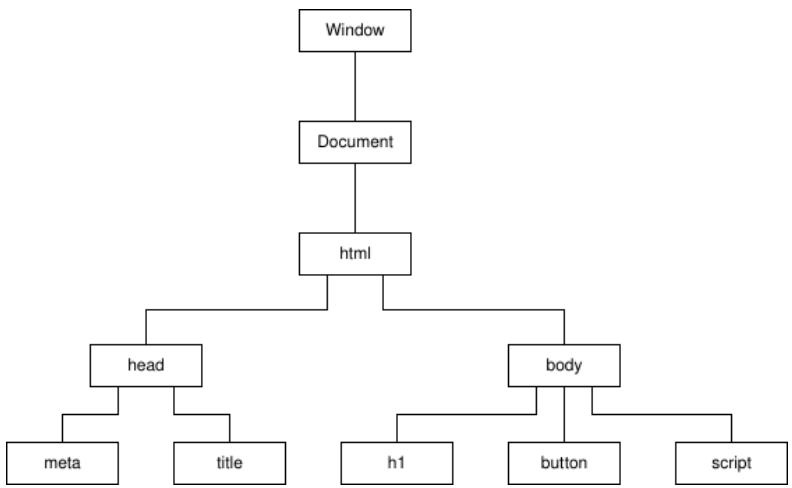
Ejemplo: Node.js

*Node.js operates on a single-thread **event loop**, using **non-blocking** I/O calls, allowing it to support tens of thousands of concurrent connections without incurring the cost of thread context switching. The design of sharing a single thread among all the requests that use the **observer pattern** is intended for building highly concurrent applications, where any function performing I/O must use a **callback**.*

```
const net = require('node:net');

const server = net.createServer((client) => {
  console.log('client connected');
  client.on('end', () => {
    console.log('client disconnected');
  });
  client.on('data', (data) => {
    client.write(data);
  });
});

server.listen(5000, () => {
  console.log('server listening');
});
```

Ejemplo: Eventos DOM

DOM (*Document Object Model*) es el modelo de datos y la API que se utiliza principalmente para representar documentos HTML.



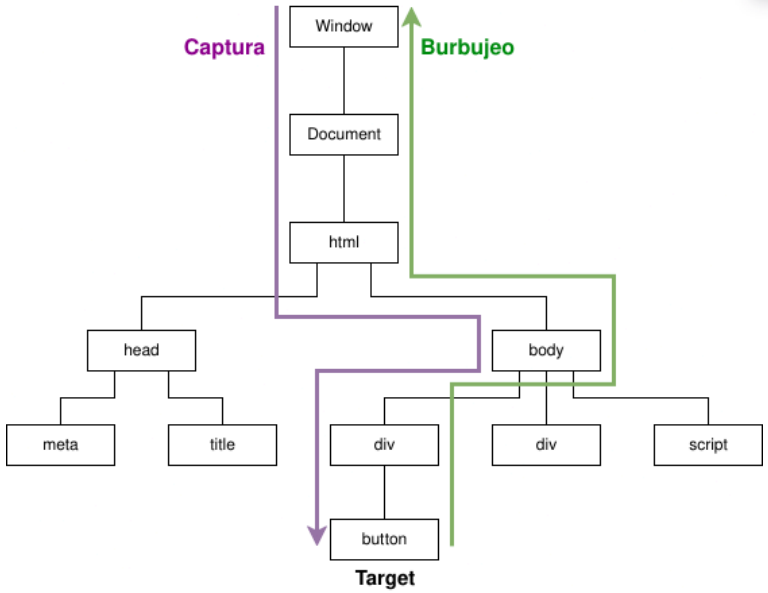

```
<!doctype html>
<html lang="en">
  <head>
      <meta charset="utf-8">
      <title>Ejemplo DOM</title>
  </head>
  <body>
      <h1>Ejemplo DOM</h1>
      <button>Cambiar color</button>

      <script>
        const btn = document.querySelector("button");

        function random(number) {
          return Math.floor(Math.random() * (number + 1));
        }

        function handler() {
          const rndCol = `rgb(${random(255)} ${random(255)} ${random(255)})`;
          document.body.style.backgroundColor = rndCol;
        };

        btn.addEventListener("click", handler);
      </script>
  </body>
</html>
```

DOM: Propagación de eventos

En el DOM, los eventos se propagan en 2 etapas:

Captura: Desde la raíz hacia un nodo **target**.
Burbujeo: Desde el nodo **target** hacia la raíz.

Este detalle es relevante cuando se registran múltiples **listeners** en el DOM: el mecanismo de propagación determina el orden en que se ejecutan los **callbacks**.



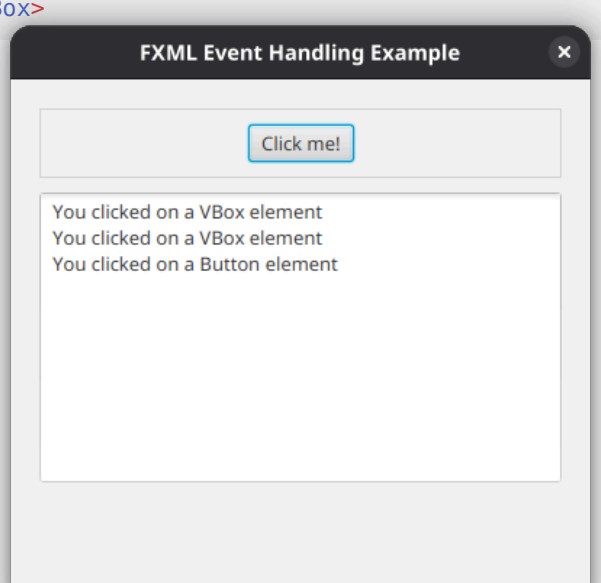

```html
<body>
  <div id="container">
    <button>Click me!</button>
  </div>
  <pre id="output"></pre>
</body>
```

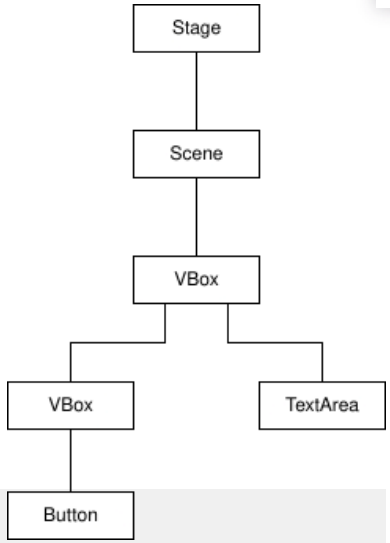
JSFiddle:

```js
const output = document.querySelector("#output");
function handleClick(e) {
  output.textContent += `You clicked on a ${e.currentTarget.tagName} element\n`;
}

const container = document.querySelector("#container");
const button = document.querySelector("button");

document.body.addEventListener("click", handleClick, { capture: true });
container.addEventListener("click", handleClick, { capture: true });
button.addEventListener("click", handleClick);
```

JavaFX ~~DOM~~ Scene graph

```
<VBox fx:id="rootPane" alignment="TOP_CENTER" spacing="10" xmlns="http://javafx.com/javafx/21"
      xmlns:fx="http://javafx.com/fxml/1" fx:controller="tb025.AppController">
    <padding>
        <Insets bottom="20.0" left="20.0" right="20.0" top="20.0"/>
    </padding>
    <VBox fx:id="container" alignment="CENTER" style="-fx-border-color: lightgray; -fx-padding: 10;">
        <Button fx:id="button" text="Click me!"/>
    </VBox>
    <TextArea fx:id="output" editable="false" prefHeight="200.0"/>
</VBox>
```






```
public class AppController implements Initializable {
    @FXML private VBox rootPane;
    @FXML private VBox container;
    @FXML private Button button;
    @FXML private TextArea output;

    @Override public void initialize(URL url, ResourceBundle resourceBundle) {
        rootPane.addEventFilter(MouseEvent.MOUSE_CLICKED, this::handleClick);
        container.addEventFilter(MouseEvent.MOUSE_CLICKED, this::handleClick);
        button.addEventHandler(MouseEvent.MOUSE_CLICKED, this::handleClick);
    }

    private void handleClick(MouseEvent event) {
        Object source = event.getSource();
        String sourceName = source.getClass().getSimpleName();
        output.appendText(String.format("You clicked on a %s element\n", sourceName));
    }
}
```

JavaFX: Eventos

```
class MyEvent extends Event {
    public static final EventType<MyEvent> EVENT_TYPE = new EventType<>("MyEvent");

    public MyEvent() {
        super(EVENT_TYPE);
    }
}
```

```
public class AppController implements Initializable {
    @FXML private VBox rootPane;
    @FXML private TextArea output;

    @Override public void initialize(URL url, ResourceBundle resourceBundle) {
        Timeline tl = new Timeline(
                new KeyFrame(
                        Duration.seconds(1),
                        new EventHandler<ActionEvent>() {
                            @Override
                            public void handle(ActionEvent actionEvent) {
                                rootPane.fireEvent(new MyEvent());
                            }
                        }
                )
        );
        tl.setCycleCount(Animation.INDEFINITE);
        tl.play();

        rootPane.addEventHandler(MyEvent.EVENT_TYPE, new EventHandler<MyEvent>() {
            @Override
            public void handle(MyEvent event) {
                output.appendText("Got MyEvent!\n");
            }
        });
    }
}
```

www.**ingenieria**.uba.ar

/ingenieriauba

/FIUBAoficial