.UBAfiuba
FACULTAD DE INGENIERÍA

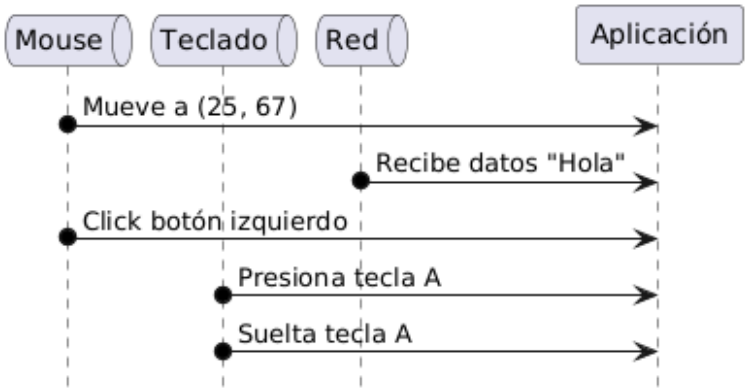
# Programación orientada a eventos

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).



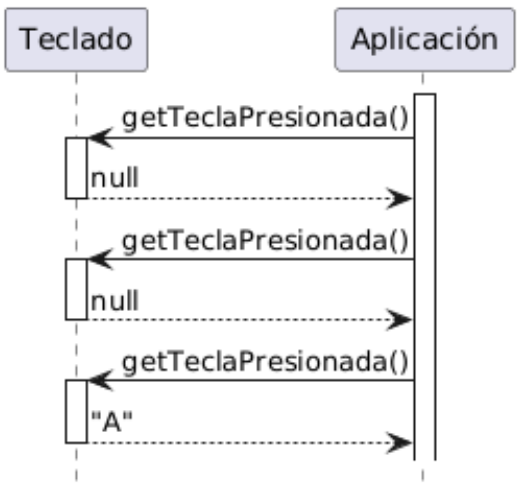

Un ejemplo común es el desarrollo de **interfaces gráficas de usuario (GUI)**. En este caso, se deben manejar eventos como clics de ratón, pulsaciones de teclado, etc. Pero no es el único contexto en el que encuentra aplicación este paradigma.

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

# Mensajes sincrónicos y *polling*

Una estrategia posible para procesar eventos es la de **polling** (encuestar).

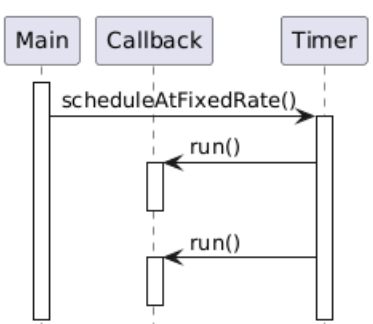


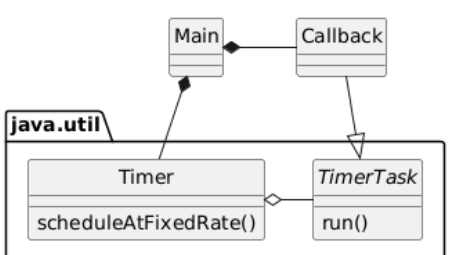

Esta estrategia es **sincrónica** y **bloqueante**: el encuestador está bloqueado en un *busy loop* hasta que ocurra el evento de interés.

# 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 **asíncrona**), por ejemplo cuando ocurre un evento.

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 implements 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.

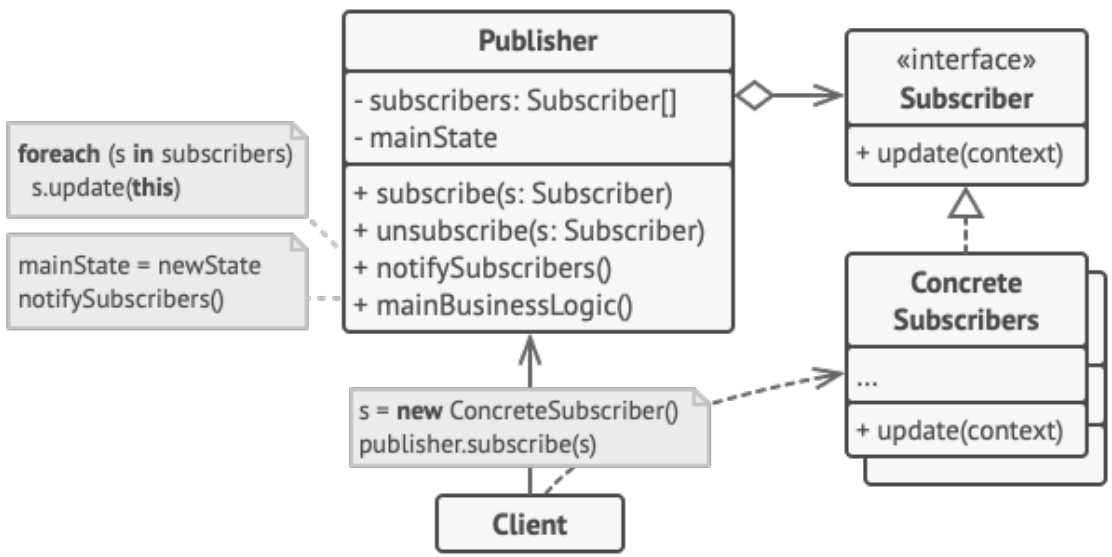
# Azúcar sintáctica: clases anónimas

```java
public class Main {
    public static void main(String[] args) {
        Timer timer = new Timer();
        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 implements 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();
        Callback 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.");
    }
}
```

# 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 callbacks.

# 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();
        }
    }
}
```

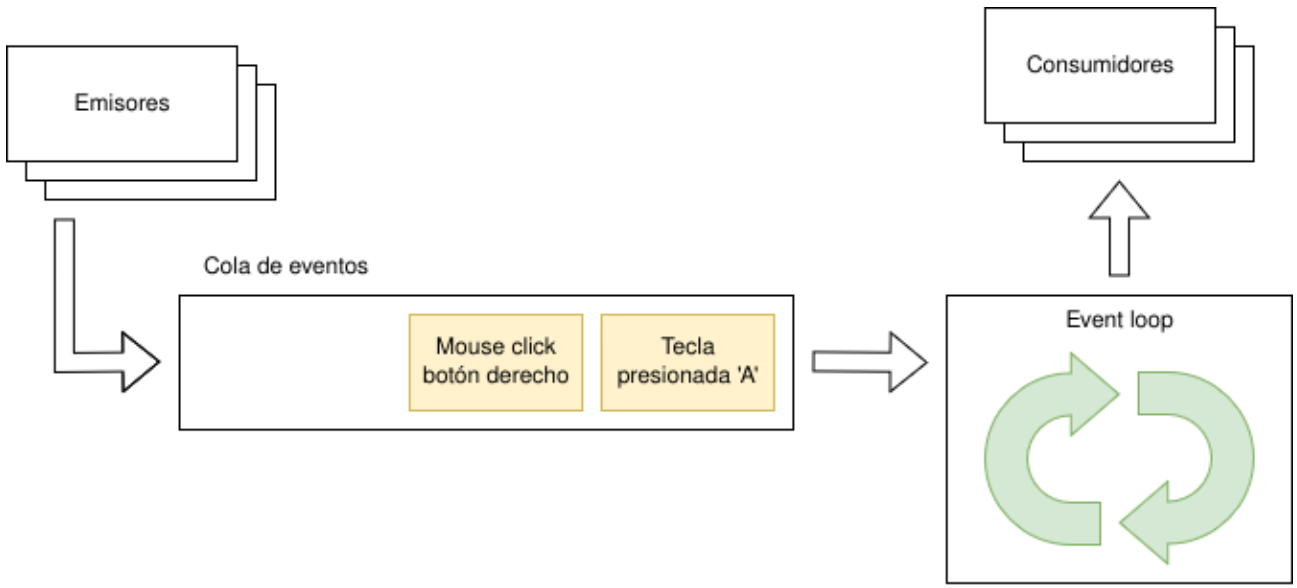
```java
public class Logros implements Observador {
    @Override public void nivelGanado() {
        System.out.println("¡Nuevo logro: Nivel ganado!");
    }
}
```

```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 del evento es **asíncrona** con la invocación del **handler**.




```
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.

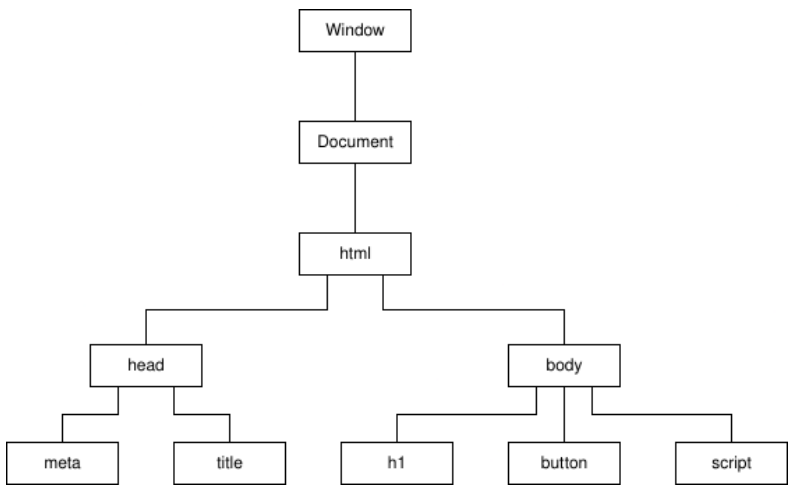
# Ejemplo: Node.js

*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**.*

```
import { createServer } from 'node:http';

function processRequest(req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World!\n');
};

function onServerListening() {
  console.log('Listening on 127.0.0.1:3000');
}

const server = createServer(processRequest);
server.listen(3000, '127.0.0.1', onServerListening);
```

# Ejemplo: Eventos DOM

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




```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>
```

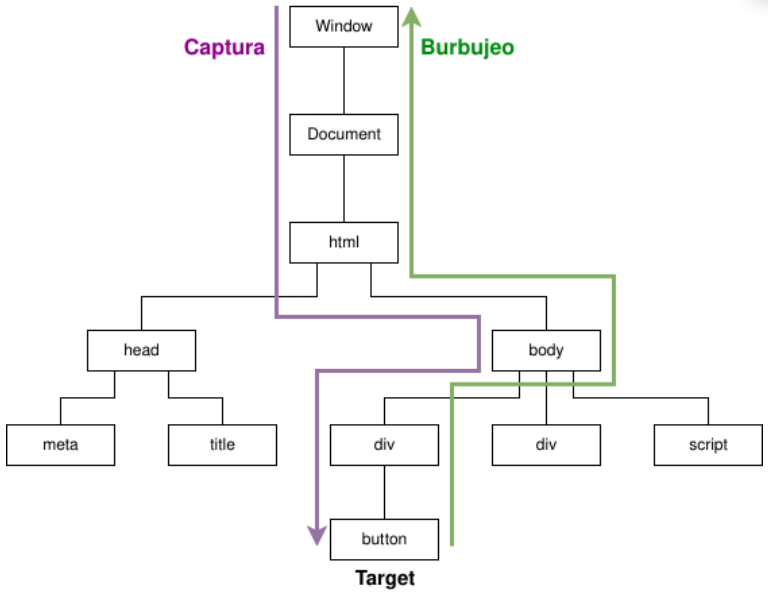
Para probar en el navegador: `data:text/html, <!doctype html> ...`

# 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**.

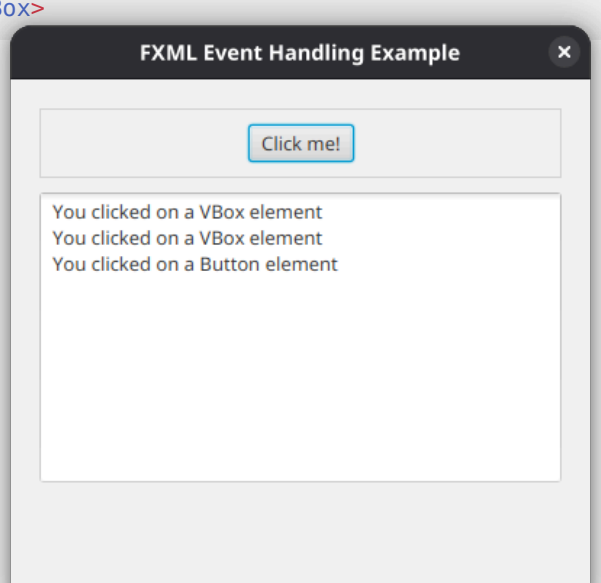


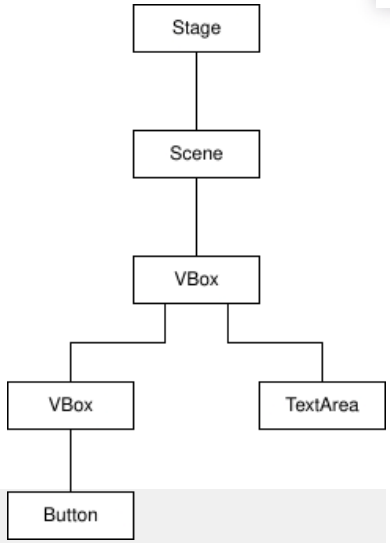

```html
<body>
  <div id="container">
    <button>Click me!</button>
  </div>
  <pre id="output"></pre>
</body>
```

```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");
            }
        });
    }
}
```

# Azúcar sintáctica: Expresión Lambda

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

```
rootPane.addEventHandler(MyEvent.EVENT_TYPE, (MyEvent event) -> {
    output.appendText("Got MyEvent!\n");
});
```

```
rootPane.addEventHandler(MyEvent.EVENT_TYPE, (event) -> {
    output.appendText("Got MyEvent!\n");
});
```

```
rootPane.addEventHandler(MyEvent.EVENT_TYPE, event -> {
    output.appendText("Got MyEvent!\n");
});
```

```
rootPane.addEventHandler(MyEvent.EVENT_TYPE, _ -> {
    output.appendText("Got MyEvent!\n");
});
```

```
rootPane.addEventHandler(MyEvent.EVENT_TYPE, _ -> output.appendText("Got MyEvent!\n"));
```

www.**ingenieria**.uba.ar

/ingenieriauba

/FIUBAoficial