Flujo de Funcionamiento del Sistema

1. Inicialización del Sistema

// Secuencia de arranque

SistemaOlimpico() → cargarConfiguracion() → conectarBD() → cargarDatos()

Proceso detallado:

1. **Constructor SistemaOlimpico()**: Inicializa listas y configuración
2. **Carga de db.properties**: Lee configuración de base de datos
3. **Conexión a BD**: Establece conexión usando Database.java
4. **Inicialización de tablas**: Ejecuta scripts DDL si no existen
5. **Carga de datos**: Recupera atletas y entrenamientos desde BD

2. Arquitectura de Persistencia

Doble Estrategia de Almacenamiento:

```
// Memoria Principal (Listas)
private List<Atleta> atletas = new ArrayList<>();
// Base de Datos (MySQL/SQLite)
Database db = new Database(url, user, password);
// Archivos JSON (Backup)
private static final String ARCHIVO_ATLETAS_JSON = "atletas.json";
```

Métodos de Programación Utilizados

1. Programación Orientada a Objetos (POO)

Encapsulación:

```
public class Atleta {
   private String id;  // Atributos privados
   private String nombreCompleto;
```

```
    // Getters y Setters públicos
    public String getId() { return id; }
    public void setNombreCompleto(String nombre) {
        this.nombreCompleto = nombre;
    }
}
```

Herencia y Polimorfismo:

```
// Uso de interfaces implícitas mediante HttpHandler
public class WebServer {
    server.createContext("/api/atletas", exchange -> {
        // Implementación polimórfica del handler
    });
}
```

Composición:

```
public class SistemaOlimpico {
    private List<Atleta> atletas;         // Composición
    private Database db;                  // Composición
    private CalculadoraPlanilla calculadora;// Composición
}
```

2. Programación Funcional

Expresiones Lambda:

```
// Ordenamiento con lambda
Collections.sort(entrenamientosOrdenados,
    (e1, e2) -> e1.getFecha().compareTo(e2.getFecha()));
// Filtrado con Streams
```

```
List<Entrenamiento> internacionales = entrenamientos.stream()
   .filter(Entrenamiento::esInternacional)
   .collect(Collectors.toList());
```

Method References:

```
// Referencia a método
resultados.stream()
   .map(WebServer::atletaToJson)  // Method reference
   .toArray(String[]::new);
```

3. Programación Concurrente

Sincronización:

```
public synchronized List<Atleta> getAtletasApi() {
   return new ArrayList<>(this.atletas);  // Copia defensiva
}
public synchronized void agregarAtletaDesdeApi(Atleta atleta) {
   this.atletas.add(atleta);
   if (db != null) db.guardarAtleta(atleta);
}
```

Manejo de Hilos en Servidor Web:

```
server.setExecutor(java.util.concurrent.Executors.newFixedThreadPool(4));
```

4. Programación Reactiva (Patrón Observer)

Eventos HTTP:

```
server.createContext("/api/atletas", exchange -> {
   // Manejo reactivo de peticiones HTTP
   if ("GET".equals(exchange.getRequestMethod())) {
      manejarGetAtletas(exchange);
   } else if ("POST".equals(exchange.getRequestMethod())) {
```

```
        manejarPostAtleta(exchange);
    }
});
```

5. Programación Declarativa vs Imperativa

Estilo Declarativo (SQL):

```
String sql = "SELECT * FROM atletas WHERE disciplina = ?";
// El QUÉ se quiere, no el CÓMO
```

Estilo Imperativo (Java):

```
for (Atleta atleta : atletas) {         // CÓMO hacerlo
    if (atleta.getDisciplina().equals("atletismo")) {
        resultados.add(atleta);
    }
}
```

Patrones de Diseño Implementados

1. Singleton (Implícito)

```
public class SistemaOlimpico {
    // Una instancia principal que gestiona todo el sistema
    private static SistemaOlimpico instancia;
    public static void main(String[] args) {
        SistemaOlimpico sistema = new SistemaOlimpico(); // Singleton de aplicación
    }
}
```

2. MVC (Model-View-Controller)

Modelo: Atleta, Entrenamiento, Database

Vista: index.html, styles.css, app.js

Controlador: SistemaOlimpico, WebServer

3. Data Access Object (DAO)

```
public class Database {
   // Patrón DAO para abstraer el acceso a datos
   public void guardarAtleta(Atleta a) { ... }
   public List<Atleta> cargarAtletas() { ... }
   public void deleteAtleta(String id) { ... }
```

4. Factory Method

```
// En Database.java - creación de conexiones
private Connection connect() throws SQLException {
   return DriverManager.getConnection(url, user, password);
}
```

5. Strategy Pattern

```
// Diferentes estrategias de persistencia
if (db != null) {
   // Estrategia BD
   db.guardarAtleta(atleta);
} else {
   // Estrategia archivos JSON
   guardarAtletasJSON();
}
```

Flujo de Datos Detallado

Caso de Uso: Registrar Nuevo Atleta

1. Desde Consola:

```
private void registrarAtleta(Scanner scanner) {
   // 1. Recoger datos
   String id = "ATL" + (atletas.size() + 1);
   System.out.print("Nombre completo: ");
   String nombre = scanner.nextLine();
   // 2. Crear objeto
   Atleta atleta = new Atleta(id, nombre, edad, disciplina, departamento, nacionalidad,
fechaIngreso);
   // 3. Persistir en memoria
   atletas.add(atleta);
   // 4. Persistir en BD (si existe)
   if (db != null) {
      db.guardarAtleta(atleta);
   }
   // 5. Feedback al usuario
   System.out.println("Atleta registrado: " + atleta);
}
```

2. Desde API Web:

```
// WebServer.java
server.createContext("/api/atletas", exchange -> {
   if ("POST".equals(exchange.getRequestMethod())) {
      // 1. Parsear JSON del body
      String body = new String(exchange.getRequestBody().readAllBytes(), "UTF-8");
      Map<String,String> data = parseJsonSimple(body);
      // 2. Crear objeto Atleta
```

```
    Atleta a = new Atleta(id, nombre, edad, disciplina, departamento, nacionalidad, fechaIngreso);
    // 3. Delegar al sistema principal
    sistema.agregarAtletaDesdeApi(a);
    // 4. Responder con JSON
    exchange.getResponseHeaders().set("Content-Type", "application/json");
    exchange.sendResponseHeaders(201, json.length());
  }
});
```

3. Persistencia en Database:

```
public void guardarAtleta(Atleta a) {
  String sql = "INSERT OR REPLACE INTO atletas VALUES(?,?,?,?,?,?,?)";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, a.getId());
    ps.setString(2, a.getNombreCompleto());
    // ... más parámetros
    ps.executeUpdate();
  }
}
```

Algoritmos Clave del Sistema

1. Cálculo de Planilla

```
public double calcularPagoMensual(Atleta atleta, int mes, int año) {
  // 1. Contar entrenamientos del mes
  int countEntrenamientosMes = contarEntrenamientosMes(atleta, mes, año);
```

```
    // 2. Contar entrenamientos internacionales
    int countExtranjeros = contarEntrenamientosInternacionales(atleta, mes, año);
    // 3. Calcular bonos
    double pagoBase = countEntrenamientosMes * pagoBasePorEntrenamiento;
    double bonoExtranjeroTotal = countExtranjeros * bonoExtranjero;
    double bonoMejor = calcularBonoMejorMarca(atleta, mes, año);
    // 4. Sumar total
    return pagoBase + bonoExtranjeroTotal + bonoMejor;
}
```

2. Búsqueda y Filtrado

```
private void buscarAtleta(Scanner scanner) {
    String criterio = scanner.nextLine().toLowerCase();
    List<Atleta> resultados = new ArrayList<>();
    for (Atleta atleta : atletas) {
        // Búsqueda lineal O(n) - adecuado para datasets pequeños
        if (atleta.getNombreCompleto().toLowerCase().contains(criterio) ||
            atleta.getId().toLowerCase().contains(criterio)) {
            resultados.add(atleta);
        }
    }
}
```

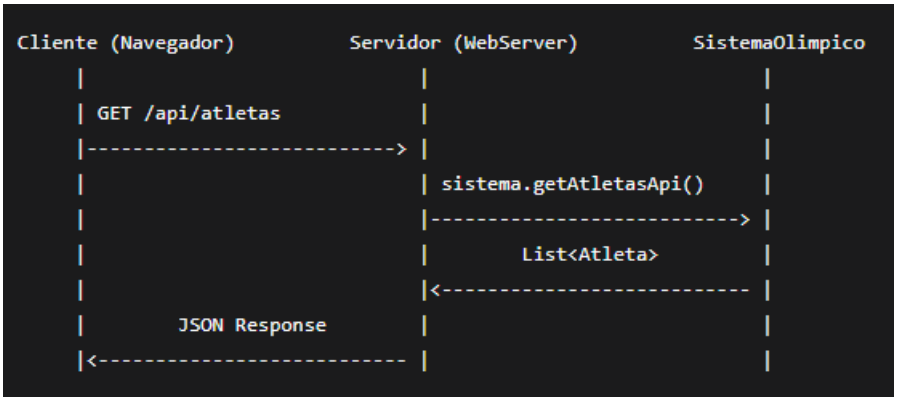
3. Algoritmo de Mejor Marca

```
private double calcularBonoMejorMarca(Atleta atleta, int mes, int año) {
    double mejorMarcaHistorica = 0;
    // Encontrar mejor marca histórica (antes del mes actual)
    for (Entrenamiento ent : atleta.getEntrenamientos()) {
```

```
        if (esAntesDelMes(ent.getFecha(), mes, año)) {
            if (ent.getValorRendimiento() > mejorMarcaHistorica) {
                mejorMarcaHistorica = ent.getValorRendimiento();
            }
        }
    }
    // Verificar si se superó en el mes actual
    for (Entrenamiento ent : atleta.getEntrenamientos()) {
        if (esDelMes(ent.getFecha(), mes, año)) {
            if (ent.getValorRendimiento() > mejorMarcaHistorica) {
                return bonoMejorMarca; // Bono aplicado
            }
        }
    }
    return 0.0; // Sin bono
}
```

Comunicación Cliente-Servidor

Flujo HTTP Completo:



Ejemplo de JSON Intercambiado:

// Request (POST /api/atletas)

```
{
  "nombre": "Juan Pérez",
  "edad": "25",
  "disciplina": "atletismo",
  "departamento": "Guatemala",
  "nacionalidad": "guatemalteco"
}
// Response
{
  "id": "ATL5",
  "nombreCompleto": "Juan Pérez",
  "edad": 25,
  "disciplina": "atletismo",
  "departamento": "Guatemala",
  "nacionalidad": "guatemalteco",
  "fechaIngreso": "2024-01-15"
}
```

Manejo de Errores y Validaciones

Estrategias de Validación:

1. Validación en Frontend (JavaScript):

```
document.getElementById('formAtleta').addEventListener('submit', async (e)=>{
    e.preventDefault();
    const fd = new FormData(e.target);
    const payload = Object.fromEntries(fd.entries());
```

```
    // Validación básica
    if (!payload.nombre || !payload.edad) {
        alert('Nombre y edad son requeridos');
        return;
    }
});
```

2. Validación en Backend (Java):

```
public synchronized boolean agregarEntrenamientoDesdeApi(Entrenamiento ent) {
    // Validar que el atleta exista
    Atleta atleta = buscarAtletaPorId(ent.getIdAtleta());
    if (atleta == null) return false;
    // Validar datos del entrenamiento
    if (ent.getValorRendimiento() < 0) {
        throw new IllegalArgumentException("Valor de rendimiento no puede ser
negativo");
    }
    atleta.agregarEntrenamiento(ent);
    return true;
}
```

3. Manejo de Excepciones:

```
try {
    LocalDate fecha = LocalDate.parse(fechaStr);
    atleta.setFechaIngreso(fecha);
} catch (DateTimeParseException e) {
    System.out.println("Formato de fecha inválido. Use YYYY-MM-DD");
    // Recovery: mantener fecha anterior
```

```
}
```

Optimizaciones y Consideraciones de Rendimiento

1. Estrategias de Caching:

```
public class SistemaOlimpico {
   private List<Atleta> atletas; // Cache en memoria
   private void cargarDatos() {
      // Carga inicial completa para evitar queries repetitivas
      if (db != null) {
         this.atletas = db.cargarAtletas();
      }
   }
}
```

2. Queries Optimizadas:

```
// Uso de Prepared Statements para reutilización
String sql = "INSERT INTO atletas VALUES(?,?,?,?,?,?,?)";
PreparedStatement ps = conn.prepareStatement(sql);
// Reutilizar el mismo PreparedStatement con diferentes parámetros
```

3. Paginación (Para futuras versiones):

```
// Actual: carga completa
// Futuro: paginación para grandes volúmenes
public List<Atleta> getAtletasPaginados(int pagina, int tamañoPagina) {
   int inicio = pagina * tamañoPagina;
   int fin = Math.min(inicio + tamañoPagina, atletas.size());
   return atletas.subList(inicio, fin);
```

}

Ciclo de Vida de los Objetos

Atleta: