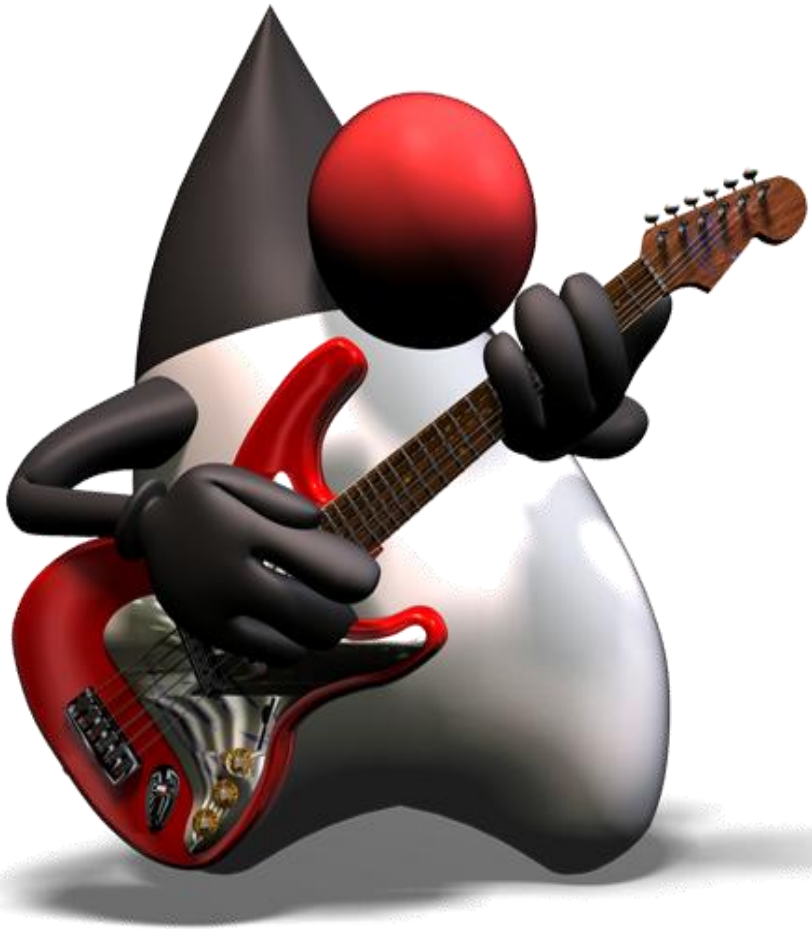


CJAVA

CJAVA
siempre para apoyarte

AUSPICIAN:

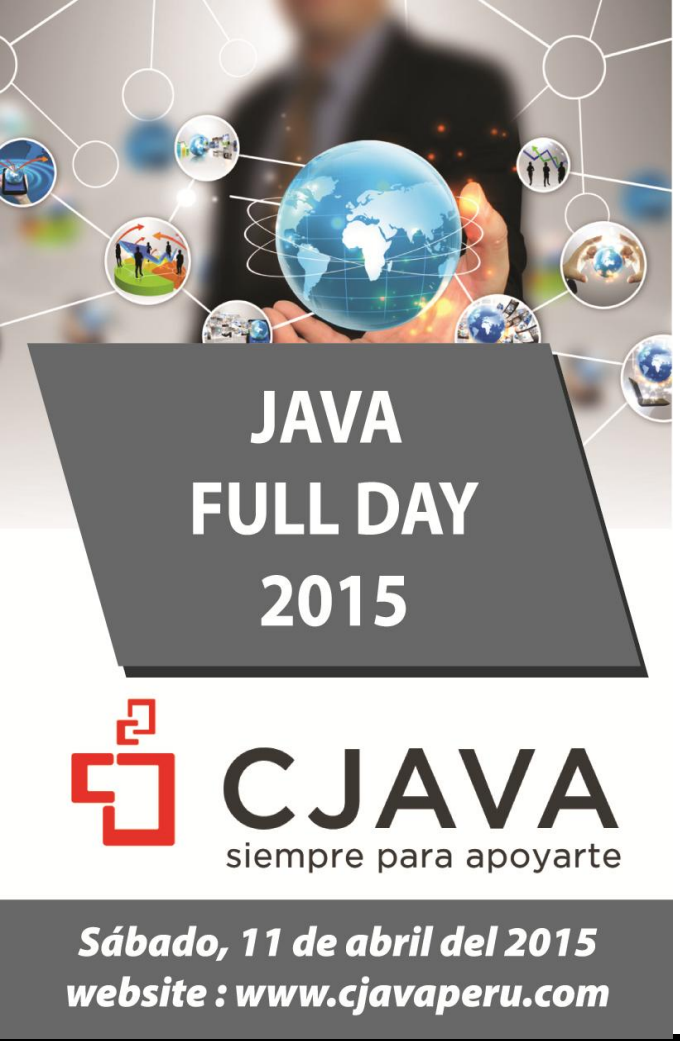
3dev BUSINESS & CONSULTING

JB Enterprise Group
Especialización - Consultoría

ELISE
Aseguramos la Calidad de Energía
que sus equipos necesitan

Universidad WIENER

TATA
TATA CONSULTANCY SERVICES



Quiénes Somos

Somos una organización orientada a desarrollar, capacitar e investigar tecnología JAVA a través de un prestigioso staff de profesionales a nivel nacional.

Tema de conferencia:

Orígenes
Precedente
Interfaces Funcionales
Expresiones Lambda
Streams

Ing. Eric Gusto Coronel Castillo
gcoronelc.blogspot.com

CJAVA
siempre para apoyarte

Orígenes

- El cálculo lambda fue desarrollado por Alonso Church en la década de los años 30 con el objeto de dar una teoría general de las funciones.
- El cálculo lambda ha sido empleado como fundamento conceptual de los lenguajes de programación, aportando:
  - Una sintaxis básica
  - Una semántica para el concepto de función como proceso de transformación de argumentos en resultados
- El cálculo lambda sirve como el modelo computacional de lenguajes que implementan la programación funcional, como Lisp, Haskell, Ocaml, etc.
- Características del cálculo lambda como expresiones lambda se han incorporado en muchos lenguajes de programación ampliamente utilizados, ahora muy recientemente Java 8.

Precedente

- **Clases anónimas**
  - Una clase anónima siempre debe ser una subclase de otra clase existente o bien debe implementar alguna interfaz.
  - La definición de la clase anónima y la creación de una instancia de la misma representan acciones inseparables que se realizan en la misma línea de código.

Precedente

```
public static void main(String[] args) {

  List<String> lista = Arrays.asList("Gustavo","Guino","Sergio","Cesar","Ernesto");
  System.out.println(lista);

  Collections.sort(lista, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
      return o1.compareTo(o2);
    }
  });

  System.out.println(lista);

}
```

```
run:
[Gustavo, Guino, Sergio, Cesar, Ernesto]
[Cesar, Ernesto, Guino, Gustavo, Sergio]
```

CJAVA
siempre para apoyarte

Interfaces Funcionales

- Son interfaces cuya característica es que tienen solo un método abstracto.
- Normalmente, su implementación es como una clase anónima, definida en la misma línea de código donde se crea el objeto de la clase.
- Son la base de las expresiones lambda.
- Ejemplos: Comparator, Runnable, ActionListener, Callable.

Interfaces Funcionales

```
@FunctionalInterface
public interface IMate {

  int opera(int n1, int n2);

}
```

```
public static void main(String[] args) {

  IMate o = new IMate() {

    @Override
    public int opera(int n1, int n2) {
      return (n1 + n2);
    }
  };

  System.out.println("5 + 8 = " + o.opera(5, 8));
}
```

CJAVA
siempre para apoyarte

Interfaces Funcionales

```
public static void main(String[] args) {

  IMate sumar = (a,b) -> (a+b);
  IMate restar = (a,b) -> (a*b);

  System.out.println("5 + 8 = " + sumar.opera(5, 8));
  System.out.println("5 * 8 = " + restar.opera(5, 8));

}
```

```
run:
5 + 8 = 13
5 * 8 = 40
```

Expresiones Lambda

- Una expresión lambda se compone de tres partes:

Lista de argumentos	Operador	Cuerpo
`( int x, int y )`	`->`	`{ x + y }`

  - Si es un solo argumento, no necesita los paréntesis.
  - Si es una sola sentencia, no necesita las llaves.

CJAVA
siempre para apoyarte

Expresiones Lambda

```
public static void main(String[] args) {

  List<String> lista = Arrays.asList("Gustavo", "Guino",
     "Sergio", "Cesar", "Ernesto");
  System.out.println(lista);

  Collections.sort(lista, (o1, o2) -> o1.compareTo(o2));
  System.out.println(lista);

}
```

```
run:
[Gustavo, Guino, Sergio, Cesar, Ernesto]
[Cesar, Ernesto, Guino, Gustavo, Sergio]
```

Expresiones Lambda

ActionListener

```
public class Prueba06 extends JFrame{

  private JButton button;

  public Prueba06() throws HeadlessException {
    super("Demo Clásico");
    setLayout(new GridLayout(1, 1));
    setSize(200, 200);
    setLocationRelativeTo(null);
    button = new JButton("Saludar");
    add(button);
    button.addActionListener( e ->
      JOptionPane.showMessageDialog(rootPane, "Hola Gustavo."));
  }

  public static void main(String[] args) {
    Prueba06 bean = new Prueba06();
    bean.setVisible(true);
  }

}
```

Streams

- Una secuencia de elementos que soportan operaciones secuenciales y paralelas.

```
public static void main(String[] args) {

  List<Integer> lista =
    Arrays.asList(34, 76, 23, 78, 15, 80, 45, 67);

  lista.stream()
      .filter( n -> n > 50)
      .sorted()
      .forEach( n -> System.out.println(n));

}
```

```
run:
67
76
78
80
```

Streams

- Soportan diferentes tipos de operaciones.

```
public static void main(String[] args) {

  List<Integer> lista =
    Arrays.asList(34, 76, 23, 78, 15, 80, 45, 67);

  int suma = lista.stream()
                  .filter( n -> n > 50)
                  .mapToInt(n -> n.intValue())
                  .sum();

  System.out.println("Suma: " + suma);

}
```

```
run:
Suma: 301
```

Streams

- Soportan diferentes tipos de operaciones.

```
public static void main(String[] args) {

  int suma = IntStream
      .of(34, 76, 23, 78, 15, 80, 45, 67)
      .filter( n -> n > 50)
      .sum();

  System.out.println("Suma: " + suma);

}
```

```
run:
Suma: 301
```

CJAVA
siempre para apoyarte

Streams

- Soportan diferentes tipos de operaciones.

```
public static void main(String[] args) {

  Stream
   .of("Gustavo","Guino","Sergio","Cesar","Ernesto")
   .map(s -> s.toUpperCase())
   .sorted()
   .forEach(System.out::println);

}
```

```
run:
CESAR
ERNESTO
GUINO
GUSTAVO
SERGIO
```

CJAVA

siempre para apoyarte

Gracias