

Programación Concurrente



Paradigmas de Lenguajes
3er Año LCC– Facultad CEFyN – UNSJ –

Concurrencia

Cuando la solución de **un problema** se logra ejecutando varias tareas al mismo tiempo



... Programación concurrente ...

Una aplicación hace mas de una cosa a la vez

Introducción

La concurrencia no es una idea nueva. Gran parte de la base teórica se estableció en la década de 1960, y Algol 68 incluye funciones de programación concurrente.

Sin embargo, el interés generalizado en la concurrencia es un fenómeno relativamente **reciente**; proviene de:

- la disponibilidad de máquinas multinúcleo
- multiprocesador de bajo costo,
- la proliferación de app gráficas, web y multimedia; basados en subprocesos de **control concurrentes**.

Concurrencia. Motivaciones:

- Captar la estructura lógica de un problema. Esto es estructurar un programa en bloques de código en hilos independientes.
- Explotar hardware paralelo (múltiples procesadores o múltiples núcleos dentro de un procesador), por velocidad. Para usar estos núcleos de manera efectiva, los programas generalmente deben escribirse (o reescribirse) teniendo en cuenta la concurrencia.
- Hacer frente a la distribución física. Las aplicaciones que se ejecutan en Internet o en un grupo más local de máquinas son intrínsecamente concurrentes.

Definiciones:

- Un sistema es **concurrente** si dos o más tareas pueden estar en ejecución (en algún punto) al mismo tiempo.
- Un sistema concurrente es **paralelo** si más de una tarea puede estar físicamente activa a la vez; esto requiere más de un procesador. La distinción es puramente una cuestión de implementación y rendimiento: desde un punto de vista semántico, no hay diferencia entre el verdadero paralelismo y el “cuasiparalelismo” de un sistema que cambia entre tareas en momentos impredecibles.
- Un sistema paralelo es **distribuido** si sus procesadores están asociados con personas o dispositivos que están físicamente separados unos de otros en el mundo real.

“Concurrente” se aplica a las tres motivaciones anteriores.

“Paralelo” se aplica a la segunda y tercera;

“Distribuido” se aplica solo a la tercera.”

Ejemplo:

map head [[4,1,6], [5,2,9], [9,4,7]]

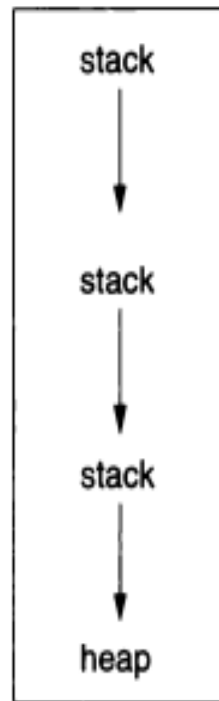
[head:[4,1,6], head:[5,2,9], head:[9,4,7]]

=> [4, 5, 9]

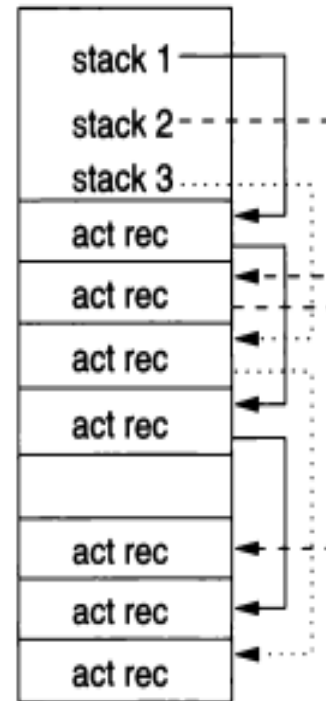
Modelos alternativos de memoria



(a) Single stack

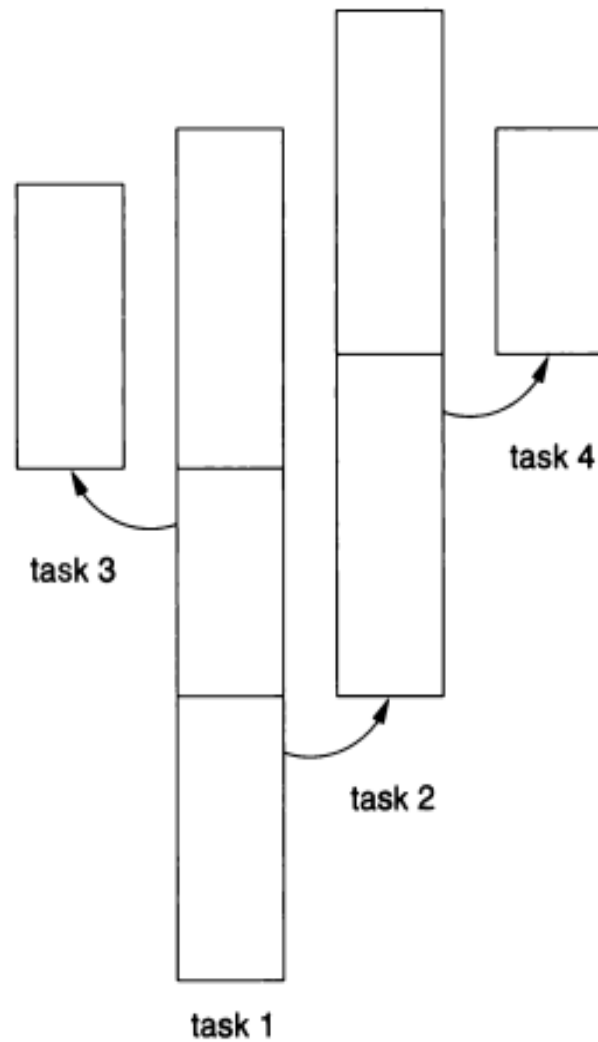


(b) Multiple stacks



(c) Single heap

Modelo Pila Cactus



Concurrencia en Java

- ➔ Java fue uno de los primeros lenguajes de programación con soporte integrado para **programación concurrente**.
- ➔ Los primeros programadores de Java estaban entusiasmados con lo fácil que era cargar imágenes en hilos background o implementar un servidor web que atendiera múltiples solicitudes al mismo tiempo.

En ese momento, la atención se centraba en **mantener ocupado** a un procesador mientras que algunas tareas se pasan el tiempo esperando a la red.

- ➔ Hoy en día, la mayoría de las computadoras tienen múltiples procesadores o núcleos, y los programadores se preocupan por mantenerlos ocupados.

JDK 5.0 incluye **API de alto nivel** en los paquetes
`java.util.concurrent`

Hilos y procesos:

Unidades básicas de ejecución en prog concurrentes:

- **Procesos:** tiene un entorno de ejecución autónomo. Un proceso generalmente tiene un conjunto completo y privado de recursos básicos en tiempo de ejecución; en particular, cada proceso tiene su propio espacio de memoria.
- **Hilos:** procesos livianos.

Ambos procesos e hilos proporcionan un entorno de ejecución, pero la creación de un hilo requiere menos recursos que la creación de un nuevo proceso. Los hilos existen dentro de un proceso; cada proceso tiene al menos uno. **Los hilos comparten los recursos** del proceso, incluida la memoria y los archivos abiertos. Esto lo convierte en una comunicación eficiente, pero potencialmente problemática.

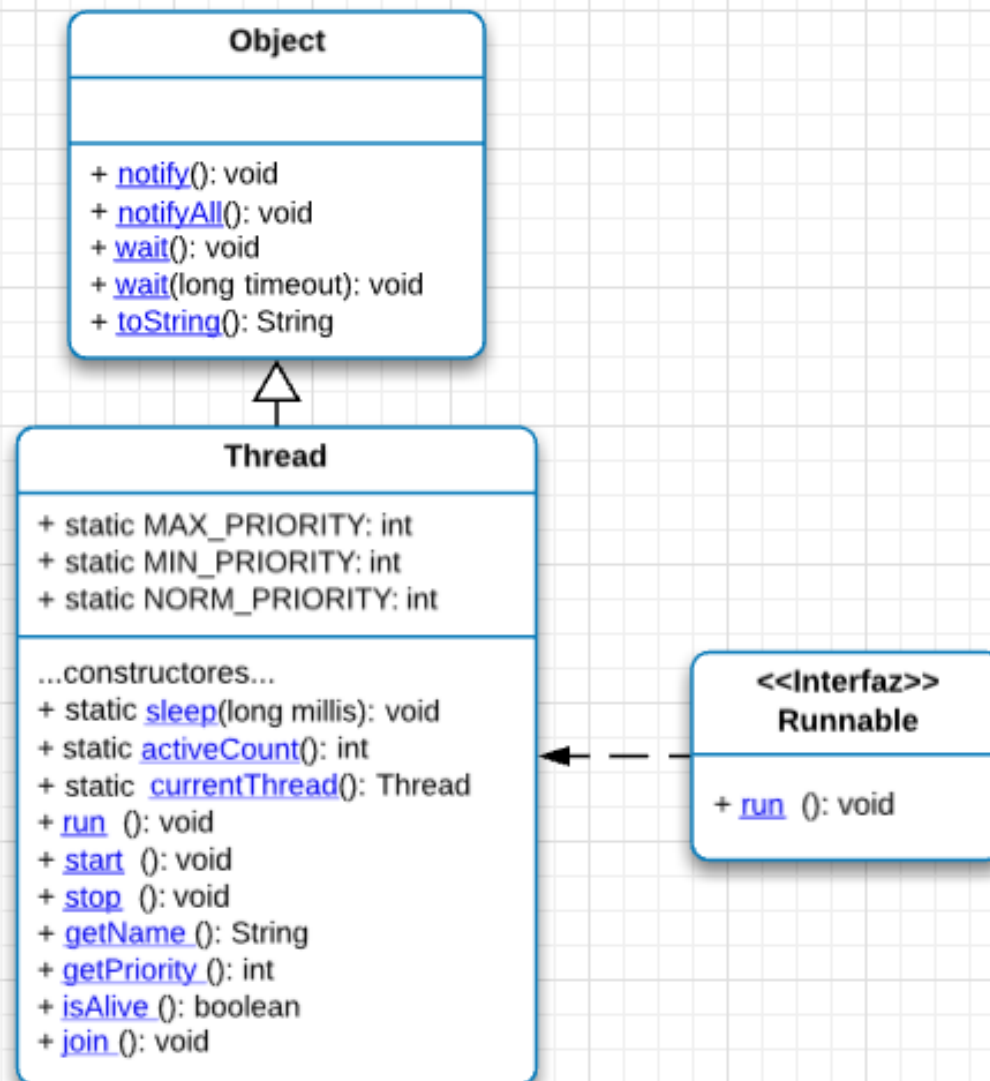
La ejecución multihilo es una característica esencial
de la plataforma Java.

Thread

Cada hilo está asociado con una instancia de la clase Thread. Hay dos estrategias básicas para usar objetos Thread:

- ➔ Para controlar directamente la creación y gestión de hilos, simplemente se crea una instancia de **Thread**.
- ➔ Para abstraer la administración de hilos del resto de la aplicación, pase las tareas a un **executor**.

API. Jerarquía



Runnable es la interfaz que se debe implementar para lograr la concurrencia:

```
1. public class HolaThread extends Thread {  
    @Override  
    public void run() {  
        System.out.println("Hola desde un hilo !!");  
    }  
    public static void main(String args[]) {  
        (new HolaThread()).start();  
    }  
}
```

```
2. public class HolaRunnable implements Runnable {  
    @Override  
    public void run() {  
        System.out.println("Hola desde un hilo !!");  
    }  
    public static void main(String args[]) {  
        (new Thread(new HolaRunnable())).start();  
    }  
}
```


Ejemplo 1:

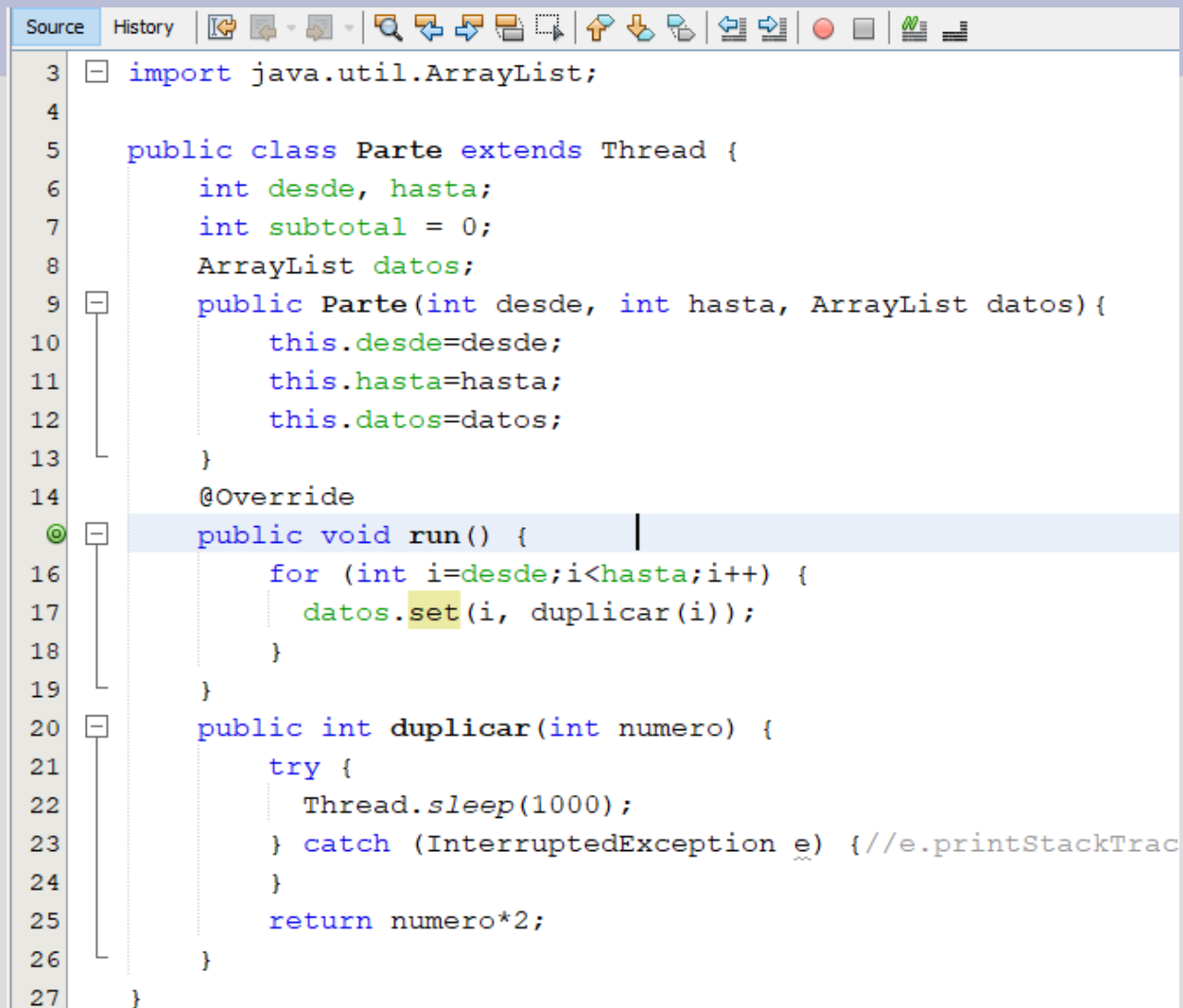
```
package hilos;
class Hola extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 1000; i++)
            System.out.println("Hola " + i);
    }
}
class Adios extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 1000; i++)
            System.out.println("Adios " + i);
    }
}
public class Hilos {
    public static void main(String args[]){
        (new Hola()).start();
        (new Adios()).start();
    }
}
```

Ejemplo 2: Un hilo en ejecución

```
4
5 public class Ejemplo2 {
6     public static void main(String[] args){
7         ArrayList datos = new ArrayList();
8         for(int i=0; i < 15; i++){
9             datos.add(i,i);
10        }
11        long numerol=System.currentTimeMillis();
12        for (int i=0;i<15;i++) {
13            duplicar(i);
14        }
15        long numero2=System.currentTimeMillis();
16
17        System.out.print("Mili Segundos = ");
18        System.out.println(numero2-numerol);
19
20        for(int i=0; i < 15; i++){
21            System.out.println(i + " = "+datos.get(i));
22        }
23    }
24    public static int duplicar(int numero) {
25        try {
26            Thread.sleep(1000);
27        } catch (InterruptedException e) {
28            //e.printStackTrace();
29        }
30        return numero*2;
31    }
```

Ejemplo 3: Varios hilos ejecutando

```
Source History 
7 public class Principal {
8     public static void main(String[] args){
9         ArrayList datos = new ArrayList();
10        for(int i=0; i < 15; i++){
11            datos.add(i,i);
12        }
13        long numero1=System.currentTimeMillis();
14        Thread h1 = new Parte(0,5,datos);
15        Thread h2 = new Parte(5,10,datos);
16        Thread h3 = new Parte(10,15,datos);
17        h1.start();
18        h2.start();
19        h3.start();
20        try {
21            h1.join();
22            h2.join();
23            h3.join();
24        } catch (InterruptedException ex) {
25            Logger.getLogger(conHilos.Principal.class.getName()).log(Level.SEVERE, nul
26        }
27        long numero2=System.currentTimeMillis();
28        System.out.print("Mili Segundos = ");
29        System.out.println(numero2-numero1);
30        for(int i=0; i < 15; i++){
```

```
3  import java.util.ArrayList;
4
5  public class Parte extends Thread {
6      int desde, hasta;
7      int subtotal = 0;
8      ArrayList datos;
9      public Parte(int desde, int hasta, ArrayList datos){
10         this.desde=desde;
11         this.hasta=hasta;
12         this.datos=datos;
13     }
14     @Override
15     public void run() {
16         for (int i=desde;i<hasta;i++) {
17             datos.set(i, duplicar(i));
18         }
19     }
20     public int duplicar(int numero) {
21         try {
22             Thread.sleep(1000);
23         } catch (InterruptedException e) {e.printStackTrace();}
24     }
25     return numero*2;
26 }
27 }
```

Sincronización:

<https://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html>

Los hilos se comunican principalmente **compartiendo el acceso** a los datos y los objetos.

Esta forma de comunicación es **extremadamente eficiente**, pero hace posible dos tipos de errores:

- interferencia de hilo
- errores de consistencia de memoria.



Sincronización



Debe controlar:

- Deadlock
- Starvation (Inanición) y
- Livelock (Bloqueo)

Ejercicio:

¿Qué ocurrirá si intento compilar y correr este código?

```
package ejercicios;

public class Preguntal implements Runnable{
    public static void main(String[] args) {
        Preguntal p=new Preguntal();
        Thread t=new Thread(p);
        t.start();
    }
    @Override
    public void run() {
        while(true) {
            Thread.currentThread().sleep(1000);
            System.out.println("bucle while");
        }
    }
}
```

- ☐ a. Error en compilación.
- ☐ b. Compila y hay una única salida "bucle while".
- ☐ c. Compila y no hay ninguna salida.
- ☐ d. Compila y repite la salida del "bucle while".

Colecciones:

“Las colecciones son fundamentales para muchas tareas de programación, permiten agrupar y procesar datos.”

Supongamos la tarea de crear un menú como una colección de platos para calcular diferentes consultas. Por ej:

- Conocer los platos vegetarianos del menú.
- Averiguar el número total de calorías para el menú.
- Seleccionar solo platos bajos en calorías.

Toda esta lógica del negocio implica operaciones similares a las de una base de datos, como agrupar los platos por categoría o encontrar el plato más caro.

```
Select nombre  
      from platos  
      where calorías < 400;
```

El motor de BD permite realizar la consulta en forma declarativa.
(Aquí se expresa **qué** quiero como resultado y **no cómo** implementar)

¿ Cómo lo resolverá el motor ???

¿ Cómo crees que se resuelve en Java ???

Streams

Son una actualización de la API de Java 8 que permiten manipular colecciones de datos de forma declarativa.

Se pueden procesar en paralelo de forma transparente, ¡sin tener que escribir código multi-hilo !!!

Ejemplo:

Listar el nombre de los platos que son bajos en calorías
ordenados por calorías

Source History

```
2 public class Plato {
3     private final String nombre;
4     private final boolean vegetariano;
5     private final int calorías;
6     private final Type tipo;
7
8     public Plato(String nombre, boolean vegetariano, int calorías, Type tipo) {
9         this.nombre = nombre;
10        this.vegetariano = vegetariano;
11        this.calorías = calorías;
12        this.tipo = tipo;
13    }
14    public String getNombre() {
15        return nombre;
16    }
17    public boolean esVegetariano() {
18        return vegetariano;
19    }
20    public int getCalorías() {
21        return calorías;
22    }
23    public Type getType() {
24        return tipo;
25    }
26    @Override
27    public String toString() {
28        return nombre;
29    }
30    public enum Type { CARNE, PESCADO, OTRO }
```

iReport output Search Results Output 1:18


```
14
15 | List<Plato> menu = Arrays.asList(
16 |     new Plato("cerdo", false, 800, Plato.Type.CARNE),
17 |     new Plato("bife", false, 700, Plato.Type.CARNE),
18 |     new Plato("pollo", false, 400, Plato.Type.CARNE),
19 |     new Plato("papas fritas", true, 530, Plato.Type.OTRO),
20 |     new Plato("arroz", true, 350, Plato.Type.OTRO),
21 |     new Plato("fruta de temporada", true, 120, Plato.Type.OTRO),
22 |     new Plato("pizza", true, 550, Plato.Type.OTRO),
23 |     new Plato("langostino", false, 300, Plato.Type.PESCADO),
24 |     new Plato("salmon", false, 450, Plato.Type.PESCADO));
25
```

Solución clásica:

Ejemplo: Listar el nombre de los platos que son bajos en calorías ordenados por calorías.

```
25 |
26 | List<Plato> bajasCalorias = new ArrayList<>();
27 |
28 | 📌 for(Plato obj: menu) {
29 |     if(obj.getCalorias() < 400) {
30 |         bajasCalorias.add(obj);
31 |     }
32 | }
33 |
34 | 📌 Collections.sort(bajasCalorias, new Comparator<Plato>() {
35 |     📌 public int compare(Plato obj1, Plato obj2) {
36 |         return Integer.compare(obj1.getCalorias(), obj2.getCalorias());
37 |     }
38 | });
39 |
40 | bajasCalorias.forEach(obj -> System.out.println(obj.getNombre()));
41 |
42 |
```

Expresión
lambda

Resolviendo las advertencias:

```
25
26     List<Plato> bajasCalorias = new ArrayList<>();
27
28     menu.stream().filter(obj -> (obj.getCalorias() < 400)).forEachOrdered(obj -> {
29         bajasCalorias.add(obj);
30     });
31
32     Collections.sort(bajasCalorias, (Plato obj1, Plato obj2) ->
33         Integer.compare(obj1.getCalorias(), obj2.getCalorias()));
34
35     bajasCalorias.forEach(obj -> System.out.println(obj.getNombre()));
36
37
38
```

Solución mejorada:

```
List<String> lista = menu.stream()
    .filter(d -> d.getCalorias() < 400) // retorn Stream<Plato>
    .sorted(comparing(Plato::getCalorias)) //Retorna Stream<Plato>
    .map(Plato::getnombre) //map(p -> p.getNombre()) //Retorna Stream<R>
    .collect(toList());
System.out.println("== 2) " + lista);
```

Salida:

```
45
46     List<String> bajasNom;
47     bajasNom = menu.stream()
48         .filter(d -> d.getCalorias() < 400)
49         .sorted(comparing(Plato::getCalorias))
50         .map(Plato::getNombre) // .map(p -> p.getNombre())
51         .collect(toList());
52
```

Output - Paralelo (run) x

```
run:
*== 1) fruta de temporada
*== 1) langostino
*== 1) arroz

*== 2) [fruta de temporada, langostino, arroz]
BUILD SUCCESSFUL (total time: 0 seconds)
```

iReport output 🔍 Search Results

Ejemplos:

```
int s = menu.stream()  
    .mapToInt(Plato::getCalorias) //convierte un Stream a un numerico stream  
    .sum(); //se aplica a numeros
```

Ejemplos:

```
int s = menu.stream()  
    .mapToInt(Plato::getCalorias) //convierte un Stream a un numerico stream  
    .sum(); //se aplica a numeros
```

```
long c = menu.stream()  
    .filter(obj -> obj.getType().equals(obj.getType().CARNE))  
    .distinct()  
    .count(); //Operación terminal
```

Ejemplos:

```
int s = menu.stream()
    .mapToInt(Plato::getCalorias) //convierte un Stream a un numerico stream
    .sum(); //se aplica a numeros
```

```
long c = menu.stream()
    .filter(obj -> obj.getType().equals(obj.getType().CARNE))
    .distinct()
    .count(); //Operación terminal
```

```
int aa = menu.stream()
    .map(obj -> obj.getCalorias()) // retorna un Stream
    .distinct()
    .reduce(0, Integer::max); //convierte cada Stream en Integer
```


Para tener en cuenta:

// forma 1

```
int aa = menu.stream()  
    .map(obj -> obj.getCalorias()) // retorna un Stream  
    .distinct()  
    .reduce(0, Integer::max); //convierte cada Stream en Integer
```

// forma 2

```
java.util.Optional<Plato> p = menu.stream()  
    .max(comparing(Plato::getCalorias));
```

// forma 3

```
java.util.OptionalInt op = menu.stream()  
    .mapToInt(Plato::getCalorias)  
    .max();
```

Ejemplos:

```
List<String> lista = menu.stream()  
    .sorted(comparing(Plato::getCalorias))  
    .map(Plato::getNombre)  
    .limit(3)  
    .collect(toList());
```

Ejemplos:

```
List<String> lista = menu.stream()
    .sorted(comparing(Plato::getCalorias))
    .map(Plato::getNombre)
    .limit(3)
    .collect(toList());
```

```
List<Plato> lista = menu.stream()
    .filter(Plato::esVegetariano)
    .collect(toList());
```

Ejemplos:

```
List<String> lista = menu.stream()
    .sorted(comparing(Plato::getCalorias))
    .map(Plato::getNombre)
    .limit(3)
    .collect(toList());
```

```
List<Plato> lista = menu.stream()
    .filter(Plato::esVegetariano)
    .collect(toList());
```

```
List<Integer> mide = menu.stream()
    .map(Plato::getNombre)
    .map(String::length)
    .collect(toList());
```

Ejemplos varios:

```
int nro = java.util.stream  
    .Stream.iterate(1, i -> i + 1)  
    .limit(3)  
    .mapToInt(d -> d.intValue())  
    .sum();
```

Ejemplos varios:

```
int nro = java.util.stream
```

```
    .Stream.iterate(1, i -> i + 1)  
    .limit(3)  
    .mapToInt(d -> d.intValue())  
    .sum();
```

```
List<Integer> nros = Arrays.asList(1, 2, 1, 3, 3, 2, 4);  
nros.stream()  
    .filter(i -> i % 2 == 0)  
    .distinct()  
    .forEach(System.out::println);
```

Ejemplos varios:

```
List<String> palabras = Arrays.asList("Hola", "mundo");
```

Ejemplos varios:

```
List<String> palabras = Arrays.asList("Hola", "mundo");
```

```
List<Integer> p = palabras.stream()  
    .map(String::length)  
    .collect(toList());
```


Ejemplos varios:

```
List<String> palabras = Arrays.asList("Hola", "mundo");
```

```
List<Integer> p = palabras.stream()  
    .map(String::length)  
    .collect(toList());
```

```
List<String> caracter = palabras.stream()  
    .map(p -> p.split("")) //convierte el arreglo en arreglo de letras  
    .flatMap(Arrays::stream) //convierte String[] a Stream  
    .distinct()  
    .collect(toList());
```

Ejemplos varios:

```
List<String> palabras = Arrays.asList("Hola", "mundo");
```

```
List<Integer> p = palabras.stream()  
    .map(String::length)  
    .collect(toList()); [4, 5]
```

```
List<String> caracter = palabras.stream()  
    .map(p -> p.split("")) //convierte el arreglo en arreglo de letras  
    .flatMap(Arrays::stream) //convierte String[] a Stream  
    .distinct()  
    .collect(toList());  
[H, o, l, a, m, u, n, d]
```

Conclusiones:

Debido a que las operaciones como **filter** (o **sorted**, **map** y **collect**) están disponibles como bloques de construcción de alto nivel, su implementación interna podría ser de un solo “hilos” o podría maximizar la arquitectura multinúcleo.

“No mas problemas con hilos y bloqueos
¡la API de Streams se encarga!”

Conclusión:

Para explotar una arquitectura multinúcleo y ejecutar este código en paralelo, solo necesita cambiar **stream ()** por **parallelStream ()**.

Por ej.:

```
long c = menu.parallelStream()  
    .filter(obj -> obj.getType().equals(obj.getType().CARNE))  
    .distinct()  
    .count();
```

Conclusión:

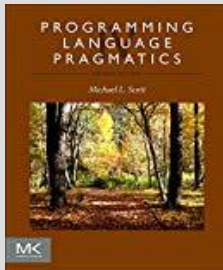
La API de Streams en Java 8 permite escribir código:

- ✓ Declarativo: más conciso y legible
- ✓ Flexible
- ✓ Paralelizable: mejor rendimiento

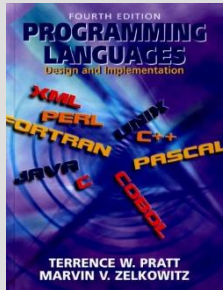
Referencias:

- ▶ Tutorial de Java. [Sitio Oficial](#)
- ▶ [Características Java 8](#)
- ▶ [Ventaja Java 9](#)
 - [Jshell](#)
- ▶ [Aspectos relevantes de Java](#)

Bibliografía:



Programming language pragmatics (4th Edition). 2016. Michael L. Scott.



Programming Languages: Design and Implementation (4th Edition). 2001. Terrence W. PRATT y Marvin V. ZELKOWITZ



Lenguajes de Programación. Diseño e Implementación (3ra. Edición). Terrence W. PRATT y Marvin V. ZELKOWITZ