Programación Concurrente en Java

Programación Concurrente – Tema 5

Miguel Ángel Rodríguez García

Carlos Grima

Lucía Serrano Luján

Universidad Rey Juan Carlos

Curso 2017 - 2018

Expresiones Lambda

Programación Concurrente en Java - Tema 5.2

Clase Thread

- Declaración en java.lang.Thread

public class Thread extends Object implements Runnable

¿Cómo crearíais un objeto Thread?

Clase Thread

¿Cómo crearías una instancia de Thread?

Clase Thread

¿Cómo crearías una instancia de Thread?

1. Forma tradicional: **creación de un objeto** que implemente la interfaz

Objeto Thread que escriba en pantalla cuando se llame a "start"

Clase Thread

- Creación TRADICIONAL: objeto

  Primero creamos una clase hilo que implemente *Runnable*:

```
public class hilo implements Runnable{ //Runnable sólo tiene el método run, de modo
    //que en el compilador va a estar implícito aunque no pusiésemos el nombre

        public void run() {
            System.out.println("Hola desde el hilo....");
        }

}
```

Clase Thread

- Creación TRADICIONAL: objeto

```
public static void main(String[] args) {
    /*-----------------------------------
    TRADICIONAL
    -----------------------------------*/
    Thread hilo = new Thread(new hilo());
    hilo.start();
```

Creamos un hilo de tipo Thread, y le pasamos una instancia de la clase *hilo*

```
public class hilo implements Runnable{ //Runnable sólo tiene el método run, de modo
    //que en el compilador va a estar implícito aunque no pusiésemos el nombre

        public void run() {
            System.out.println("Hola desde el hilo....");
        }

}
```

Clase Thread

¿Cómo crearías una instancia de Thread?

1. Forma tradicional: **creación de un objeto** que implemente la interfaz

2. **Clase anónima**: Indicamos en la declaración del thread lo que queremos que implemente en *Run*

Clase Thread

- Creación con CLASE ANÓNIMA

```
/*-------------------------------------
CON CLASE ANÓNIMA
-------------------------------------*/
//para no tener que utilizar otros archivos podríamos utilizarlo aquí mismo
Thread hilo = new Thread(new Runnable(){
    public void run() {
        System.out.println("hola otra vez");
    }
});
hilo.start();

//esto es lo que se llama CLASE ANÓNIMA porque no es necesario crear una clase llamada hilo
```

Clase Thread

¿Cómo crearías una instancia de Thread?

1. Forma tradicional: **creación de un objeto** que implemente la interfaz

2. **Clase anónima**: Indicamos en la declaración del thread lo que queremos que implemente en *Run*

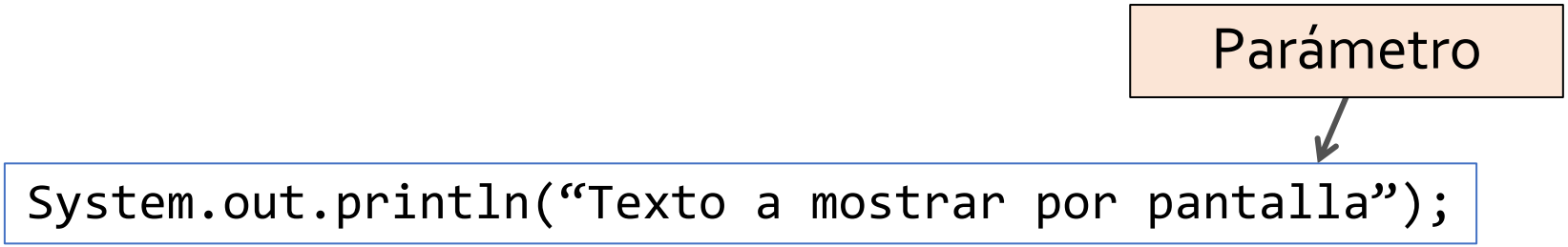
3. **Expresión lambda**: para optimizar código escribimos directamente las instrucciones, dejando al compilador deducir que estamos implementando *Run*

Clase Thread

- Creación con expresión LAMBDA

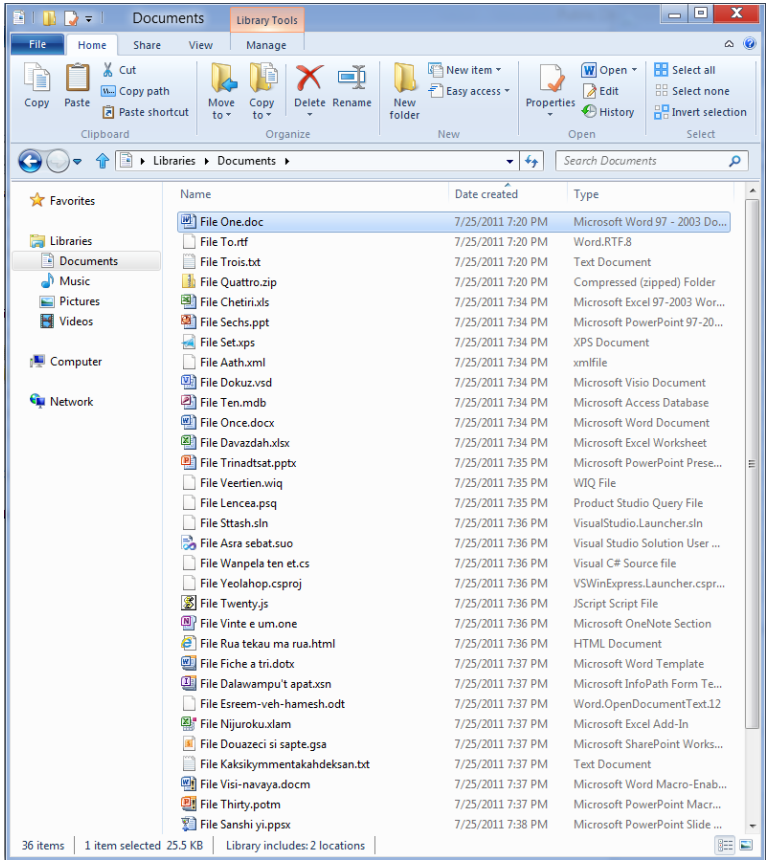
```
/*-----------------------------------
CON LAMBDA
-----------------------------------*/
//PODEMOS INCLUSO OPTIMIZARLO MÁS CON LAMBDA
//usamos lambda para el public void run
Thread hilo = new Thread(
        () -> System.out.println("Hola desde el hilo...")
);
hilo.start();
```

- Cuando se desarrolla software es habitual la **reutilización de código** en forma de métodos (o funciones)
- Esos métodos se **adaptan** para el caso concreto **mediante parámetros**
- Los parámetros suelen ser **datos**



- En ocasiones resulta útil pasar **código** como **parámetro**
- Por ejemplo:
  - Obtener la lista de ficheros dentro de un directorio que cumplen con un determinado **filtro**.
  - Ese **filtro** es un código que se ejecutará para cada fichero y devolverá **true** si el fichero pasa el filtro o **false** si el fichero no pasa el filtro.

- En ocasiones resulta útil pasar **código** como **parámetro**
- Por ejemplo:
  - Obtener la lista de ficheros dentro de un directorio que cumplen con un determinado **filtro**.
  - Ese **filtro** es un código que se ejecutará para cada fichero y devolverá **true** si el fichero pasa el filtro o **false** si el fichero no pasa el filtro.



¿¿ Filtro de ficheros .txt ??

- En la librería de Java la clase **File** permite listar el contenido de una carpeta pasando un **filtro** (**código**) como parámetro.

```
public String[] list(FilenameFilter filter)
```

- ¿Cómo se llama a este método?
- ¿Cómo se le pasa el código del filtro?

- Se puede pasar un **objeto** de una clase que implemente el interfaz **FilenameFilter**:

```
class TextFilesFilter implements FilenameFilter {

  @Override
  public boolean accept(File dir, String name) {
     return name.endsWith(".txt");
  }
}
```

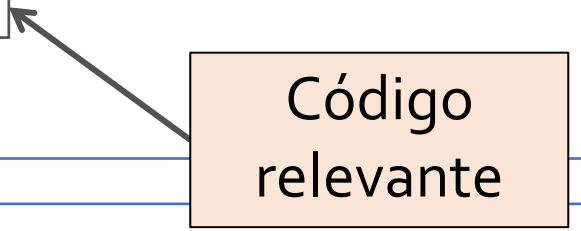
```
File currentDir = new File("");

String[] files = currentDir.list(new TextFilesFilter());
```

- Este código es **demasiado largo** para la funcionalidad que se desea

```
class TextFilesFilter implements FilenameFilter {

  @Override
  public boolean accept(File dir, String name) {
     return name.endsWith(".txt");
  }
}
```




```
File currentDir = new File("");

String[] files = currentDir.list(new TextFilesFilter());
```

- Las **expresiones lambda** son una forma **compacta** de pasar a un método **código** como **parámetro**

Expresión lambda

```
String[] files =
    currentDir.list((dir,name)->name.endsWith(".txt"));
```

- Se escribe únicamente lo **relevante**
- Disponibles desde **Java 8**

- **¿Cuándo se puede usar una expresión lambda?**
  - En todos aquellos lugares en los que se espere un objeto que implementa un **interfaz con un único método**

Interfaz con un método

```
FilenameFilter filter = (dir,name)->name.endsWith(".txt");

String[] files = currentDir.list(filter);
```

- Algunas de las interfaces en java con **un sólo método abstracto**
  - *Runnable*
  - *ActionListener*
  - *Comparator*
  - *Callable*

- **¿Cómo se escribe una expresión lambda?**
  - Si el método no tiene parámetros se ponen los paréntesis

```
Runnable runnable = () -> System.out.println();
```

  - Si sólo tiene un parámetro, se pueden omitir

```
FileFilter filter = f -> f.isDirectory();
```

- **¿Cómo se escribe una expresión lambda?**
  - Si el método tiene que devolver un valor y hay una única sentencia, se omite el **return**

```
FileFilter filter = f -> f.isDirectory();
```

  - Si son varias sentencias, se ponen entre **{ }** y se pone el **return**

```
FileFilter filter = f -> {
   int numBytes = f.length();
   return numBytes > 100;
}
```

- **¿Cómo se escribe una expresión lambda?**
  - Si lo único que vas a hacer es invocar un método en el objeto que te pasan como parámetro, puedes usar una **referencia a método**

```
FileFilter filter = f -> f.isDirectory();
```



```
FileFilter filter = File::isDirectory;
```

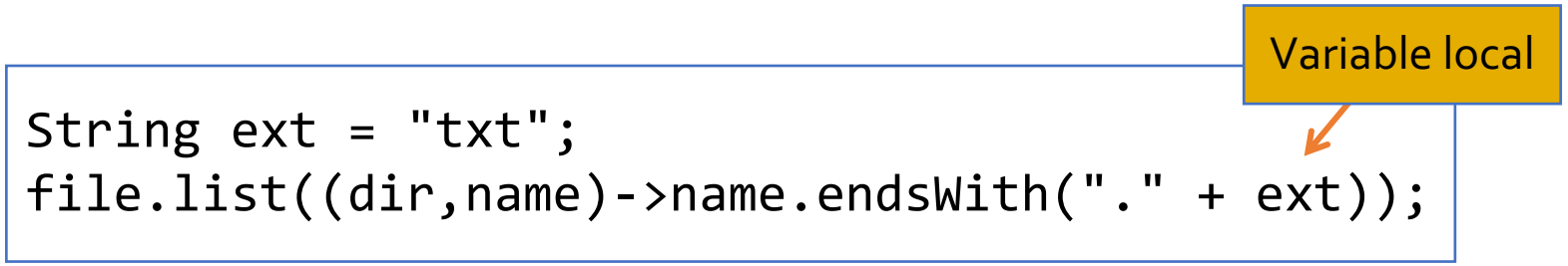
- **¿Cómo se escribe una expresión lambda?**
  - Si quieres, puedes poner el **tipo de los parámetros**, pero no es necesario

```
FileFilter filter = f -> f.isDirectory();
```



```
FileFilter filter = (File f) -> f.isDirectory();
```

- **¿Qué información puedo usar en una lambda?**
  - Puedes usar los **parámetros** de la lambda
  - Puedes usar **atributos** de la clase en la que se escribe
  - Puedes usar **this**, y se refiere al objeto en el que se escribe
  - Puedes usar **variables locales**, pero no pueden cambiar de valor



- Suma de 2 números

1. Crear una interfaz (llamada Suma) que defina un método público "sumarDosNumeros", que recibe dos enteros y devuelve la suma de ambos (tipo entero).

2. A continuación desde el *main*, crea una suma y define el método. El resultado es mostrado por pantalla.

- Suma de 2 números: clase anónima

```java
public class Main3 {

    // interfaz con un sólo método
    @FunctionalInterface //anotación opcional, a partir de Java 8
    interface Suma{
        public Integer sumarDosNumeros(Integer a, Integer b);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        Suma suma = new Suma(){
            public Integer sumarDosNumeros(Integer a, Integer b){
                return a+b;
            }
        };

        System.out.println(suma.sumarDosNumeros(10,45));
```

- Suma de 2 números

```
public class Main3 {

    // interfaz con un sólo método
    @FunctionalInterface //anotación opcional, a partir de Java 8
    interface Suma{
        public Integer sumarDosNumeros(Integer a, Integer b);
    }
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
                                                                    {
```

¿Lambda?

- Suma de 2 números: lambda

```
// interfaz con un sólo método
@FunctionalInterface //anotación opcional, a partir de Java 8
interface Suma{
    public Integer sumarDosNumeros(Integer a, Integer b);
}
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    Suma suma = (a,b) ->a+b;
    System.out.println(suma.sumarDosNumeros(10,45));
```

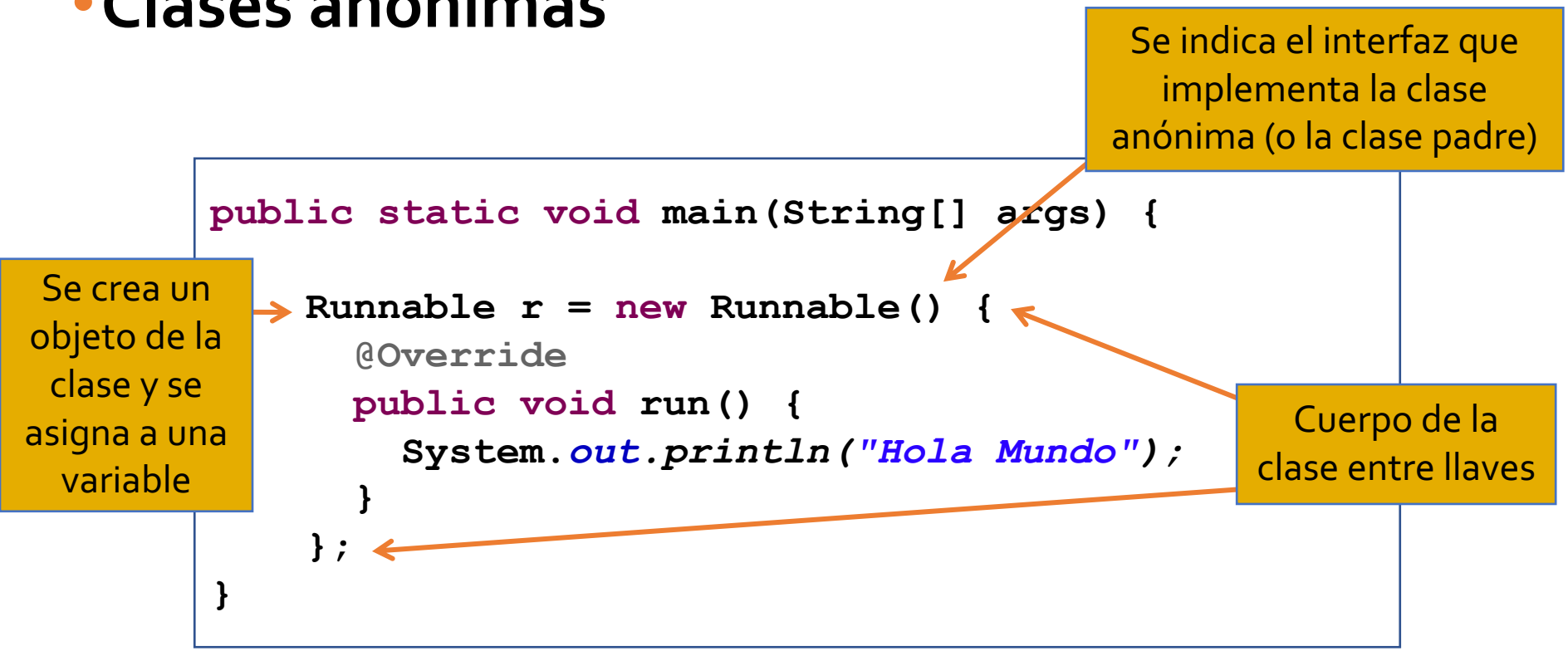
- **Clases anónimas**
  - En versiones anteriores de Java, la forma habitual de pasar código como parámetro era usar las **clases anónimas**
  - Las clases anónimas son una **forma compacta** de implementar una **clase completa** dentro de un método
  - En la misma sentencia se **declara la clase** y se **crea un objeto** de dicha clase
  - Una clase anónima **no tiene nombre**

- **Clases anónimas**

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

    Runnable r = new Runnable() {
      @Override
      public void run() {
        System.out.println("Hola mundo");
      }
    };
}
```

- **Clases anónimas**



- **Clases anónimas**
  - Las clases anónimas son muy **largas de escribir**
  - Ahora es **preferible usar una expresión lambda** siempre que sea posible en vez de una clase anónima

- **Clases anónimas**

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

    Runnable r = new Runnable() {
      @Override
      public void run() {
        System.out.println("Hola mundo");
      }
    };
}
```

```
public static void main(String[] args) {
    Runnable r = ()-> System.out.println("Hola mundo");
}
```

- **Clases anónimas vs Expresiones lambda**
  - Las expresiones lambda son más **eficientes** en tiempo de ejecución que las clases anónimas y son **más compactas** de escribir
  - Una clase anónima es una **clase completa** (puede tener clase padre, atributos, implementar varios métodos…).
  - Una lambda sólo puede usarse donde se espere un **interfaz con un método**

- **Clases anónimas vs Expresiones lambda**
  - Si se usa **this** en una clase anónima, se referencia al **objeto de la clase anónima**
  - Si se usa **this** en una expresión lambda, se referencia al **objeto donde se está declarando la lambda**

- Un lenguaje de programación se puede considerar **funcional** si permite manejar **funciones:**
  - Asignar funciones a variables
  - Pasar funciones como parámetro
  - Definir funciones anónimas en los mismo lugares que se puede poner un valor o una expresión
- Hay muchos lenguajes imperativos que permiten manejar funciones: **JavaScript, Python, Ruby, Groovy, Scala, C#, C++…**

- Ejemplo de función en **JavaScript**

```
function mySandwich(param1, param2, callback) {
    alert('Started eating. It has: '+param1+', '+param2);
    callback();
}

mySandwich('ham', 'cheese', function() {
    alert('Finished eating my sandwich.');
});
```

- Cuando se define una función usando una expresión (directamente en el código) se denomina ***lambda*** o ***clousure*** (**cerradura**)