Instructor:Goicoechea, Lucas

Email:lucasgoicoechea@gmail.com

Programación bajo Entornos Avanzados Introducción

Introducción al API Básica



Colecciones

• Podemos definir una colección como un grupo de

objetos.

• El framework de colecciones provee una API para

el manejo de diferentes tipos de colecciones:

• Con elementos duplicados o sin elementos duplicados

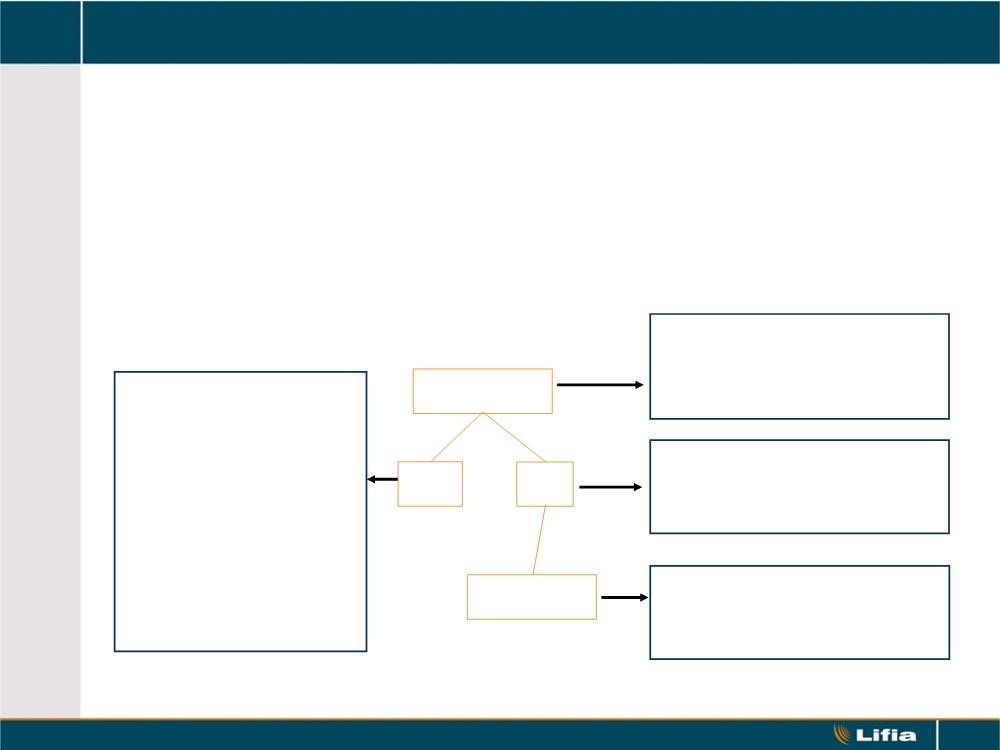
• Ordenadas o Desordenadas

• Diccionarios

• Las clases del framework encapsulan tanto los

datos como los algoritmos asociados con las

abstracciones.



Colecciones

•

El framework de colecciones define una jerarquía de

interfaces. A medida que se va bajando en la

jerarquía se va haciendo más específico el

comportamiento.

Jerarquía de Interfaces:

**Agrega la API necesaria**

Collection

**para respetar un orden**

**entre los elementos.**

**Permiten elementos**

**Define el comportamiento**

**común que deben tener todas**

**las colecciones. No hay**

**clases que la implementen.**

**No permite elementos**

**duplicados.**

**La clase Vector la**

**implementa.**

**Crece dinámicamente a**

**diferencia de los array:**

**String[]**

List

Set

SortedSet

**duplicados.**

**No tiene orden.**

**No permite elementos**

**duplicados.**

**Respeta un orden.**



Interface Collection

•

Se usa para representar cualquier grupo de objetos, o

elementos.

**Collection**

add(element:Object):boolean

addAll(collection:Collection):boolean

clear():void

contains(element:Object):boolean

remove(element:Object):boolean;

toArray():Object[]

**iterator():Iterator**

**List**

add(index:int, el:Object):boolean

addAll(index:int,

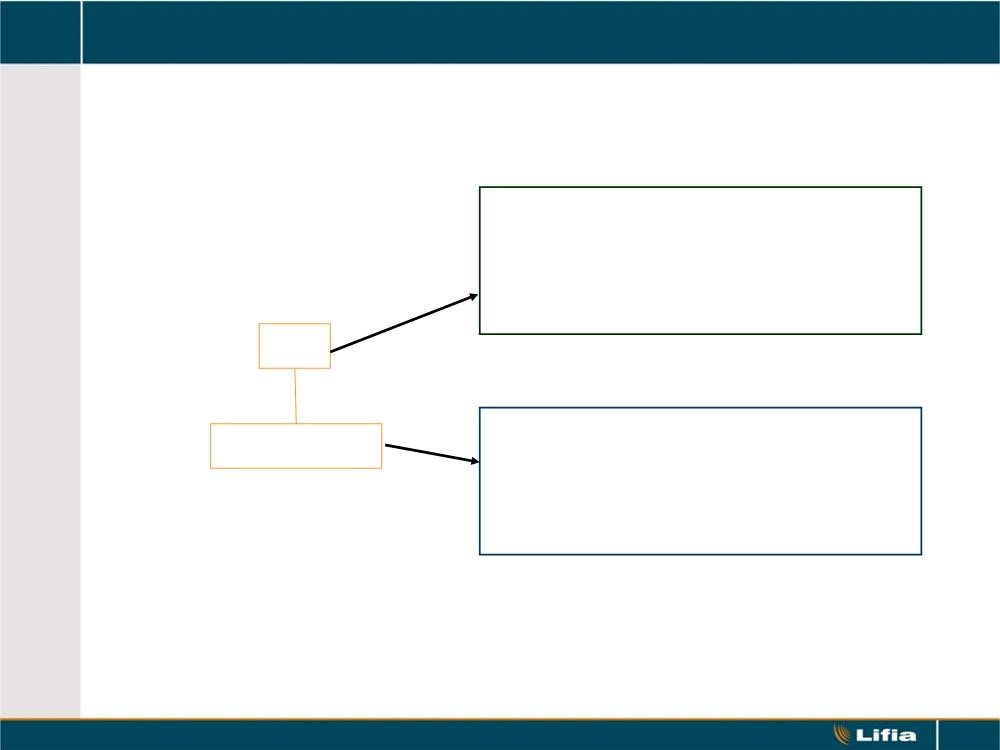
col:Collection):boolean

get(index:int):Object

indexOf(element:Object):int

remove(index:int):boolean;

subList(from:int, to:int):List



Interface Map

• Diccionarios

Define el comportamiento común que

deben tener todas las colección que

mapean claves a valores.

No permite claves repetidas.

**Map**

**SortedMap**

Agrega la API necesaria para mantener

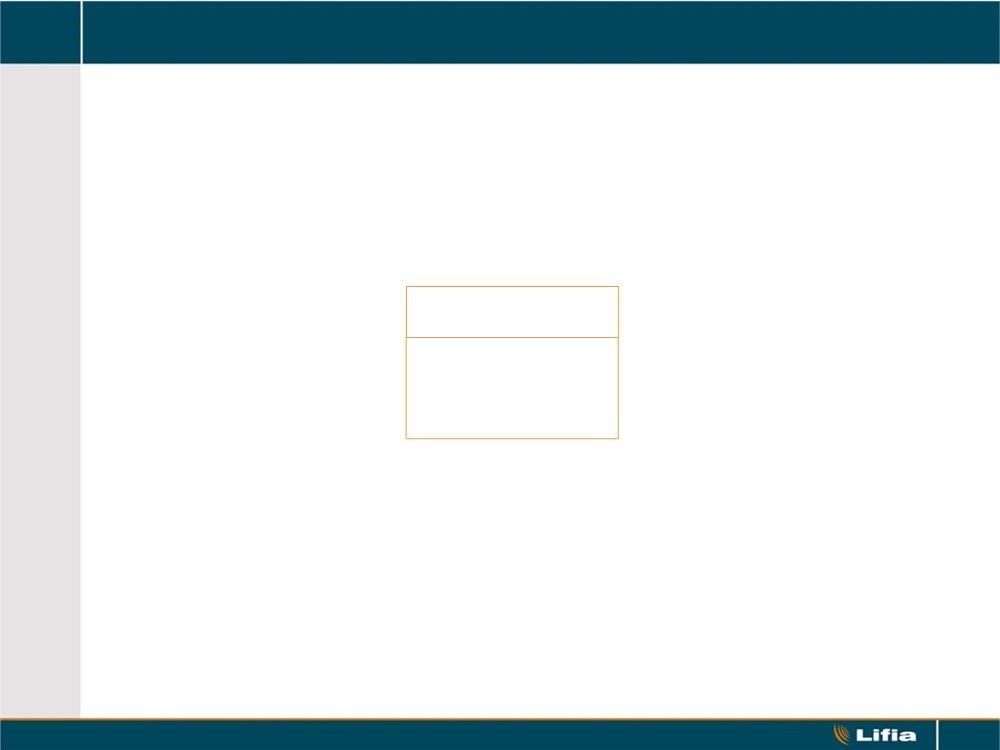
ordenada la colección según las claves.

Alguna de las clases que la implementan:

**Hashtable**

,

**HashMap**



Interface Iterator

•

Permite recorrer la colección de principio a fin.

•

El método remove() remueve el último elemento

devuelto por el método next().

Iterator

hasNext(): boolean

next(): Object

remove(): void

•

Ejemplo:

List mesas = new Vector();

// Agregar mesas.

Iterator mesasIterator = mesas.iterator();

while (mesasIterator.hasNext()) {

Mesa mesa = (Mesa) mesasIterator.next();

mesa.setCantidadPatas(4);

}



Conclusiones

•

Permite una colección de objetos de longitud variable

•

Acepta cualquier objeto

• Problema: la información de tipo se pierde al introducir un

objeto a la colección

•

No acepta tipos primitivos

•

Las implementaciones más usadas:

• Vector, funciona como un arreglo pero sin longitud fija

• Hashtable, guarda pares de objetos (clave

-

valor), permite

acceso directo por clave



Excepciones

•

Representan condiciones excepcionales que ocurren

durante la ejecución de un programa y que el

programador quiere tratar.

•

La clase Exception extiende la clase Throwable.

•

La clase Throwable provee características útiles para

tratar con excepciones

•

Específicamente:

• Provee un slot para un mensaje

• Contiene un stack trace



Excepciones

RuntimeException vs Exception

• Las RuntimeExceptions no necesita ser declaradas

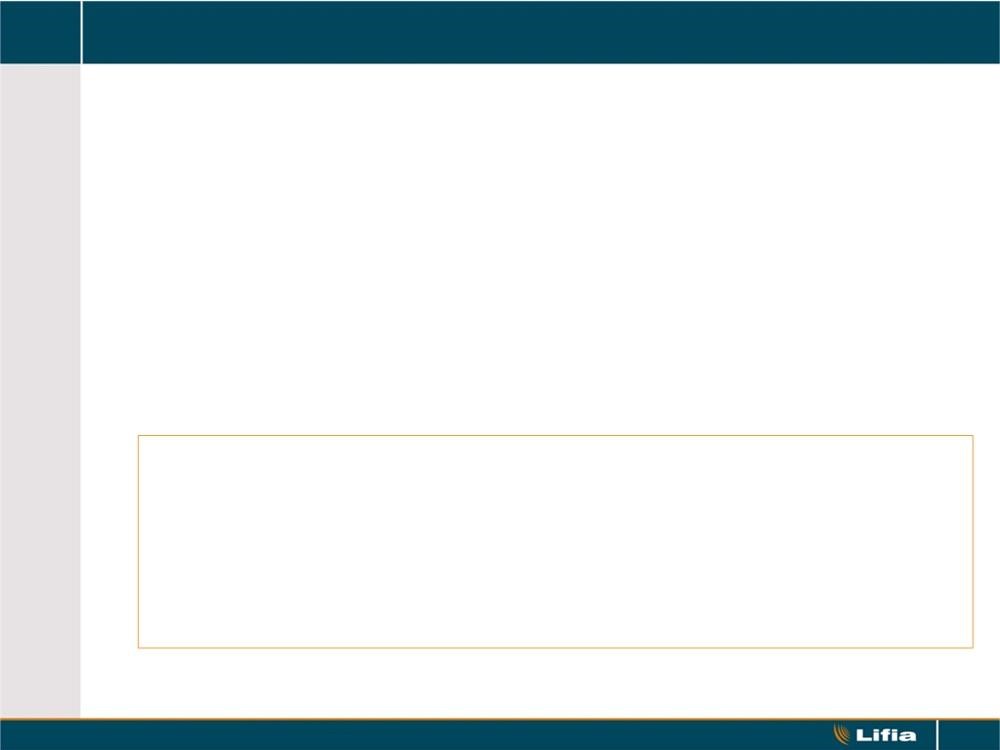
ni manejadas explícitamente. No se tiene control

sobre ellas.

• Las Exceptions tiene que ver con el dominio de la

aplicación. Deben ser declaradas y manejadas

explícitamente.



Excepciones

Declaración:

• Para crear excepciones propias generalmente se hace una

subclase de

**Exception**

.

• Por convención de nombres es una buena práctica terminar

el nombre con la palabra

**Exception.**

**package**

nombrePaquete

**;**

{

importaciones

}

[

modificadores

]

**class**

nombreException

**extends**

Exception

**{**

**}**



¿Cómo se causan excepciones?

• Implícitamente:

el programa hace algo ilegal

• Explícitamente:

ejecución de la sentencia

**throw**

.

**class**

SinNaftaException

**extends**

Exception { }

**class**

Auto {

if ( fuel < 0.1 )

**throw new**

OutOfGas();

}



Cómo se manejan?

•

¿Cómo manejar una excepción dentro del método donde

fue disparada?

**try**

block

debe haber por lo menos una de las dos opciones siguientes:

[

**catch**

(

arg) block

]

puede haber 0 ó más de esta sentencias

[

**finally**

block]

puede haber 0 ó más de esta sentencia

• block es una sentencia simple o un grupo de sentencias entre llaves



Excepciones

Ejemplo:

**public**

Object pop()

**throws**

SinElementosException

**{**

**if**

(

**this**

.getElementos().isEmpty())

**{**

**throw new**

SinElementoException();

**}**

**else**

**{**

**return**

this.getElementos().removeLast();

**}**

**}**

**try {**

elemento

**=**

pila.pop();

**}**

**catch**

(

SinElementosException e

)

**{**

e.printStackTrace();

**}**



Excepciones

Sobre el uso de excepciones:

• Una condición de error es tratada sólo donde tiene sentido

hacerlo y no en todo el nivel entre que ocurre y es tratada.

• El código puede ser escrito como si todas las operaciones

funcionaran correctamente.

• Deben ser tratadas lo más específicamente posible.

• Los bloques catch

**NO**

se deben dejar vacíos ni sólo imprimir

el

stack trace (pila de ejecución)

.



Excepciones

¿Se puede incorporar información a las excepciones?

• Es posible agregar información a un objeto excepción

respecto a la condición anormal producida.

• La cláusula

catch

permite obtener información interrogando al

objeto excepción.

• La clase Exception permite especificar mensajes de tipo

String a un objeto excepción y, recuperarlos vía el método

getMessage().



Threads

•

Un thread es un flujo de control secuencial dentro de un

programa.

•

Soportados por la máquina virtual independientemente del

SO

•

El método

**public void**

run() define el comportamiento de

un thread en ejecución.

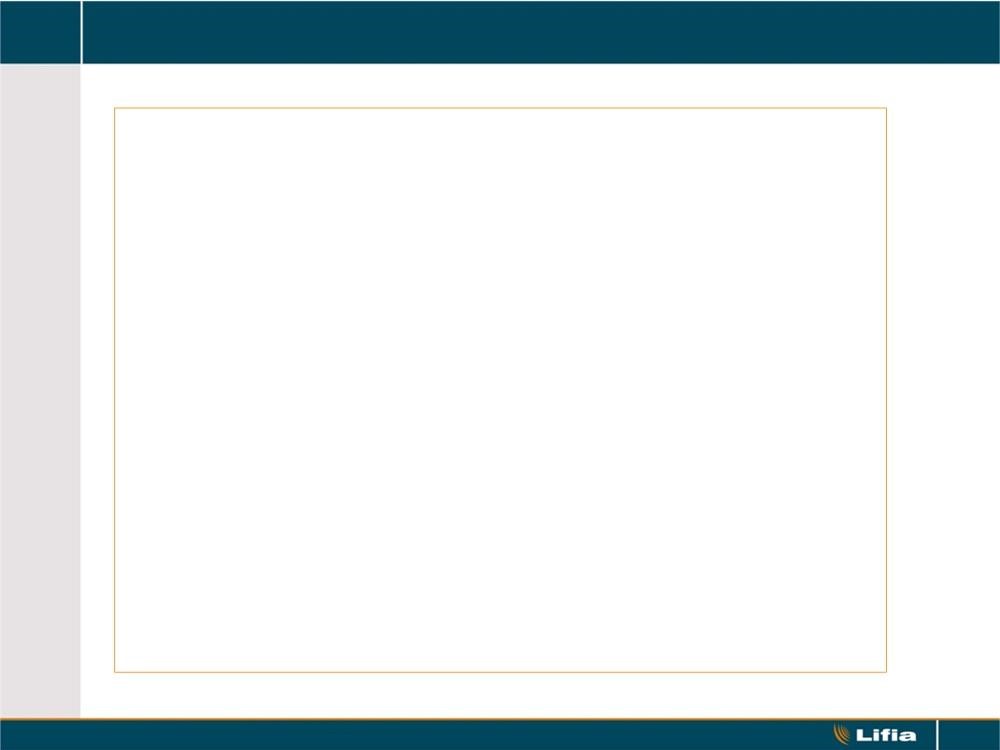
•

Existen dos mecanismos para especificar comportamiento

a un thread

• Extendiendo la clase Thread

• Implementando la interface Runnable



Subclases de Thread

**public class**

SimpleThread

**extends**

Thread {

**public**

SimpleThread(String str) {

**super**

(

str);

}

**public void**

run() {

**for**

(

**int**

i = 0; i < 10; i++) {

System.out.println(i + " " + getName());

**try**

{

sleep((

**int**

)(

Math.random() \* 1000));

}

**catch**

(

InterruptedException e)

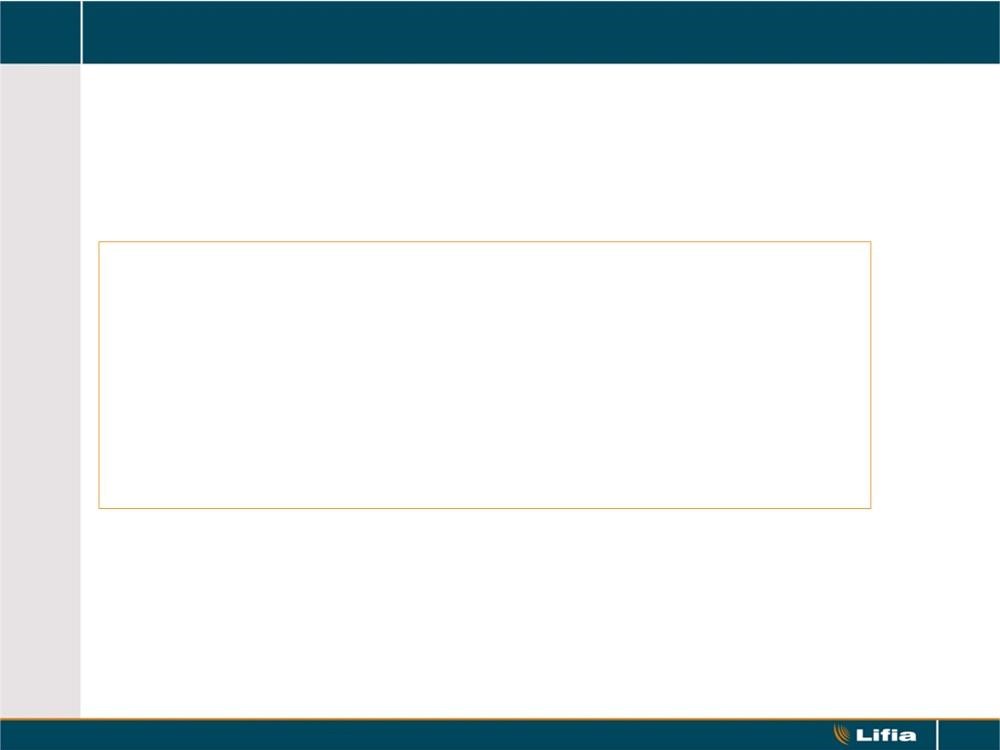
{}

}

System.out.println(getName() + “ terminó");

}

}



Subclases de Thread

Para ejecutarlo:

**public class**

DosThreads{

**public static void**

main (String[] args) {

**new**

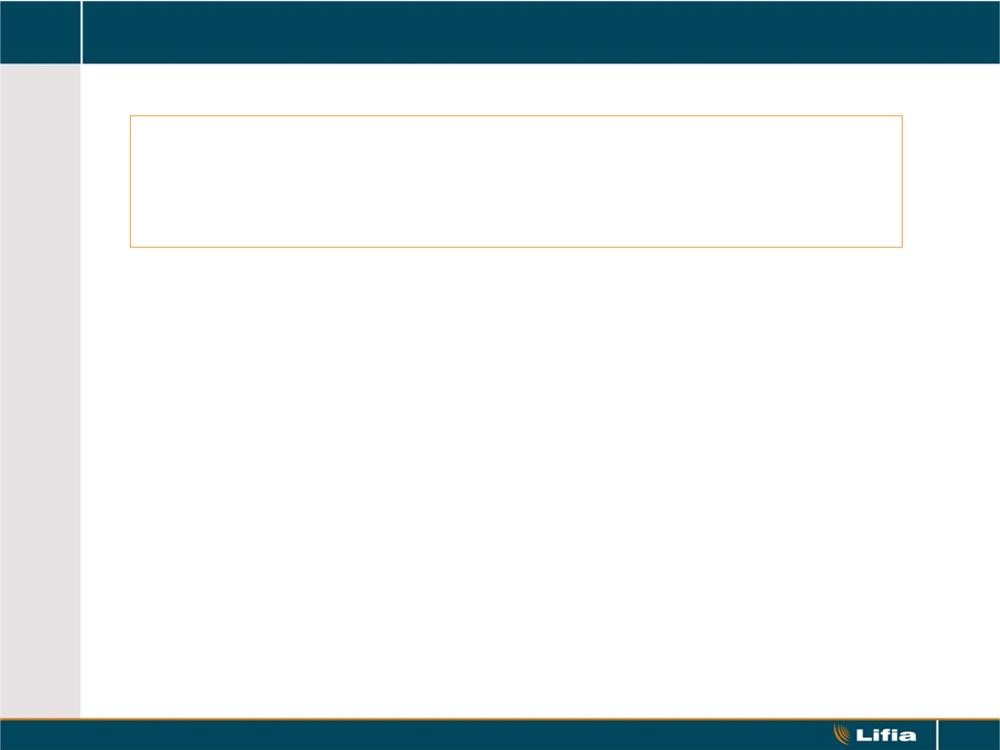
SimpleThread(“Gimnasia").start();

**new**

SimpleThread(“Estudiantes").start();

}

}



Implementar Runnable

**public interface**

Runnable{

**public void**

run();

}

•

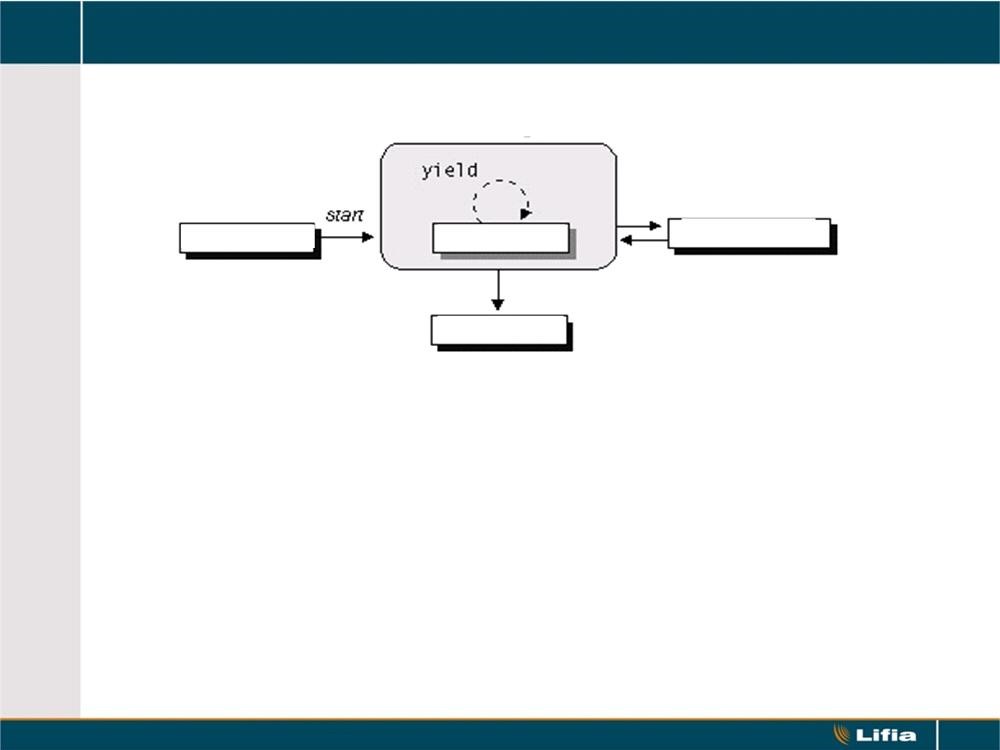
Útil cuando la clase que debe correr en un thread

separado ya es subclase de otra clase

•

Cómo modificaríamos el ejemplo anterior para usar esta

variante?



Ciclo de vida de los Threads

Ejecutándose

Inicio con el

método start

Nuevo Thread

Not Runnable

Runnable

El método run termina

Muerto

•

**Nuevo Thread**

:

cuando el thread está recién creado.

•

**Runnable**

:

cuando el thread está listo para ejecutarse o

ejecutándose.

•

**Not Runnable**

:

cuando se invocó al método

**sleep(miliseg)**

,

al

**wait()**

o el thread está esperando por I/O.

•

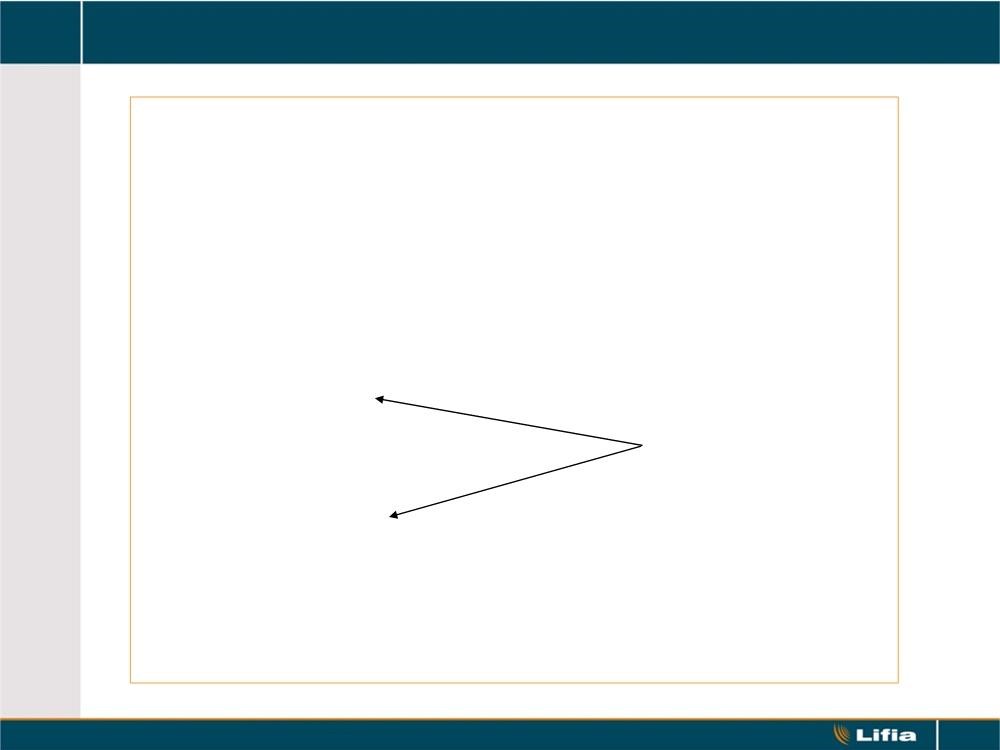
**Muerto**

:

cuando termina el método

**run()**

.



Bloqueo de Acceso de Múltiples Threads

**Qué pasaría si tenemos un contenedor que puede ser**

**accedido por distintos consumidores???**

**public class**

Contenedor {

**private int**

contenido;

**private boolean**

disponible =

**false**

;

**Garantiza**

**public synchronized int**

get() {

}

**Acceso**

**exclusivo a**

**contenido.**

**Monitor con**

**reentrada**

**public synchronized void**

put(

**int**

value) {

}

}



Sincronización de Threads

Debemos tener en cuenta:

•

El uso de prioridades (Priorities)

•

El uso de

**synchronized**

para secciones críticas

•

Los métodos wait() y notifyAll()



FIN...