

# Expresiones LAMBDA

# Introducción

JAVA introduce los conceptos de **interfaces funcionales**, **expresiones lambdas** y **referencias a métodos** para facilitar la **creación de funciones (objetos función)**.

Originariamente las **clases anónimas** permitieron **crear objetos función**:

Demasiada  
verborragia

```
Collections.sort(words, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return Integer.compare(s1.length(), s2.length());  
    }  
});  
System.out.println(words);
```

Ordena una lista de Strings de acuerdo a su longitud usando una **clase anónima** opera como una **función de comparación**

Las **clases anónimas** fueron una solución adecuada para implementar **patrones de diseño OO** que **requerían de objetos funciones**. Ejemplo es el **patrón Strategy**: la interface **Comparator** representa una **estrategia abstracta para ordenar** y la **clase anónima** una **estrategia concreta** para ordenar strings como el ejemplo de arriba.

# Los objetos función en JAVA

```
interface MathOperation {  
    double operation(double a, double b);  
}
```

```
class Addition implements MathOperation {  
  
    public double operation(double a, double b) {  
        return a + b;  
    }  
}
```

**Es muy largo!**  
**Creamos una clase para algo muy simple**

```
MathOperation addition = new MathOperation() {  
    public double operation(double a, double b) {  
        return a + b;  
    }  
};
```

**Clases anónimas, también demasiado detalle!**

```
MathOperation addition = (int a, int b) -> a + b;
```

**Código conciso: expresión lambda**



# Introducción

En JAVA 8 se formaliza la idea que las **interfaces con un único método** son especiales y requieren de un tratamiento especial. A estas interfaces se las denomina **interfaces funcionales**.

Es posible **crear instancias de interfaces funcionales** con **expresiones lambdas** o simplemente con **lambdas**.

Las **expresiones lambdas** cumplen una **función similar a las clases anónimas** pero son más **concisas**.

Lambda es una instancia de la interface `Comparator<String>`

```
Collections.sort(words,
    (s1, s2) -> Integer.compare(s1.length(), s2.length()));

-----
Collections.sort(words, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
});
System.out.println(words);
```

El compilador deduce los tipos por contexto usando inferencia de tipos

En lambdas no están presentes los tipos de datos:

- el tipo del lambda: `Comparator<String>`
- el tipo de los parámetros `s1` y `s2`: `String`
- el tipo del valor del retorno: `int`

El **compilador** para hacer inferencias usa la información aportada por los **tipos genéricos**.

# Introducción

El **compilador** para hacer **inferencias de tipos** usa la información aportada por los **tipos genéricos**.

Si se usan **raw types**, el **compilador no puede hacer inferencia de tipos** y es necesario explicitar los tipos de datos y hacer casting - > **versión más verbose**.

Versión concisa

```
public static void testLambdas_congenericos() {  
    List<String> words = new ArrayList<>();  
    words.add("hola");  
    words.add("chau");  
    words.add("genia");  
    words.add("ok");  
    words.add("1");  
    Collections.sort(words, (s1, s2) -> Integer.compare(s1.length(), s2.length()));  
}
```

Versión verbose

```
public static void testLambdas_singenericos() {  
    List words = new ArrayList<>();  
    words.add("hola");  
    words.add("chau");  
    words.add("genia");  
    words.add("ok");  
    words.add("1");  
    Collections.sort(words,  
        (String s1, String s2) -> Integer.compare(((String) s1).length(), ((String) s2).length()));  
}
```

# ¿Qué es y cómo se escribe un lambda?

Una **expresión Lambda** en JAVA es una **función**, toma **parámetros de entrada** y **devuelve un valor**. No tiene nombre, no pertenece a una clase, se puede pasar como parámetro y ejecutarse bajo demanda.

Con lambdas el compilador realiza **inferencia de tipos** y deduce los tipos de datos del contexto.

```
Collections.sort(words,  
    (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

**Parámetros (podría ser vacío)**



**Operador Flecha**

**Cuerpo del Lambda (implementación de la interface)**

**Corolario:** omitir los tipos de datos en los parámetros lambdas a menos que su presencia aporte claridad.

**Lambdas son similares a las clases anónimas** en cuanto a su función, sin embargo son mucho **más concisas**.





# Ejemplos

## Interfaces funcionales

```
interface MathOperation {  
    int operation(int a, int b);  
}
```

```
interface GreetingService {  
    void sayMessage(String message);  
}
```

```
GreetingService greetService1 = message  
    -> System.out.println("Hola " + message);
```

**Parámetros sin paréntesis ni declaración de tipos**

```
GreetingService greetService2 = (message)  
    -> System.out.println("Hola " + message);
```

**Parámetros con paréntesis y sin declaración de tipos**

```
MathOperation addition = (int a, int b) -> a + b;
```

**Parámetros con declaración de tipos**

```
MathOperation subtraction = (a, b) -> a - b;
```

**Parámetros sin declaración de tipos**

**Las variables del tipo de las Interfaces almacenan una implementación de las mismas.**

```
MathOperation multiplication = (int a, int b) -> {  
    return a * b;  
};
```

**Bloque de código entre llaves y con sentencia return**

```
MathOperation division = (int a, int b) -> a / b;
```

**Sin sentencia return y sin llaves**

# Ejemplos

Las **variables del tipo de las interfaces** almacenan una **implementación** de las mismas.

## Interfaces funcionales

```
interface MathOperation {  
    int operation(int a, int b);  
}
```

```
interface GreetingService {  
    void sayMessage(String message);  
}
```

```
int num1=Integer.valueOf(args[0]);  
int num2=Integer.valueOf(args[1]);  
MathOperation multiplicacion=(n1,n2)->n1*n2;  
int resultado=multiplicacion.operation(num1, num2);
```

```
MathOperation suma=(n1,n2)->{ return n1+n2;};  
int resultado2=suma.operation(num1, num2);
```

```
GreetingService servicio= mensaje->System.out.println("Hola "+mensaje);  
servicio.sayMessage("Labo 2021");  
servicio.sayMessage("Labo 2022");
```



# Referencias a métodos

```
Comparator<String> com2 = (s1, s2) ->  
s2.length() - s1.length();
```

```
Collections.sort(words, com2);
```

Expresiones Lambdas como objetos

En el caso en que lo único que hace la expresión lambda es invocar a un método con los parámetros pasados al lambda, se pueden usar **referencias a métodos** y se obtiene un código más conciso.

```
MyPrinter myPrinter = (s) -> { System.out.println(s); };
```

**Referencias a métodos**

```
MyPrinter myPrinter = System.out::println;
```

```
myPrinter.print("Hola");
```

Los dos puntos dobles le indican al compilador que es una **referencia a un método** y el método al que se hace referencia es el que viene después de ::

**Referencias a métodos**

```
Collections.sort(words, Comparator.comparingInt(String::length));
```

```
words.sort(comparingInt(String::length));
```

**Método de clase de la interface Comparator que construyen comparadores**

```
public interface MyPrinter {  
    public void print(String s);  
}
```

```

package enumerativos;
import java.util.function.DoubleBinaryOperator;
public enum Operation {
    PLUS("+", (x, y) -> x + y),
    MINUS("-", (x, y) -> x - y),
    TIMES("*", (x, y) -> x * y),
    DIVIDE("/", (x, y) -> x / y);
    private final String symbol;
    private final DoubleBinaryOperator op;

    Operation(String symbol, DoubleBinaryOperator op) {
        this.symbol = symbol;
        this.op = op;
    }

    public String toString() {
        return symbol;
    }

    public double apply(double x, double y) {
        return op.applyAsDouble(x, y);
    }
}

```

```

@FunctionalInterface
public interface DoubleBinaryOperator {
    double applyAsDouble(double left, double right);
}

```

Es una interface funcional del paquete **java.util.function** que representa a una función que toma 2 valores double como argumento y retorna un valor double como resultado.

### Tipo enumerativo basado en lambdas

Es posible implementar el **comportamiento específico de cada constante enum** pasándole al constructor una expresión Lambda con dicho comportamiento.

El constructor guarda la expresión Lambda en la variable de instancia **op**.

El método **apply()** se usa para invocar al Lambda. No están más las sobreescrituras del método **apply()**.



# Resumen

Las **expresiones lambdas** se utilizan para implementar **interfaces funcionales o interfaces con un único método abstracto**.

Los **lamdbas** son la mejor manera de representar **funciones pequeñas**.

Las expresiones lambdas **eliminan** la necesidad de usar **clases anónimas para interfaces función**.

El **tipo de la expresión Lambda**, los **tipos de los parámetros** y el **tipo del valor de retorno** no están necesariamente presentes en el código. El compilador usa **inferencia de tipos** para deducir los tipos del contexto.

Las **expresiones Lambdas** son esencialmente **objetos**.

Las **expresiones Lambdas no tienen estado**.

Una línea de código es ideal para un lambda y 3 es un máximo razonable.

# Colecciones en JAVA

# Colecciones

Una **colección o contenedor** es un objeto que agrupa múltiples elementos u objetos.

Las colecciones se usan para almacenar, recuperar, manipular y comunicar datos agregados.

Las colecciones representan elementos agrupados naturalmente, por ej. una carpeta de emails, un directorio telefónico, el conjunto de registros retornados al ejecutar una consulta a una BD, un diccionario (palabras con su significado), etc.

Un **framework de colecciones** es una arquitectura que permite representar y manipular colecciones de datos de manera estándar. Todos los *frameworks de colecciones* están compuestos por:

**Interfaces:** son **tipos de datos abstractos** que representan colecciones. Las interfaces permiten que las colecciones sean manipuladas independientemente de los detalles de implementación. Forman una jerarquía.

**Implementaciones:** son las implementaciones concretas de las interfaces. Son estructuras de datos reusables.

**Algoritmos:** son **métodos** que realizan operaciones útiles (búsquedas y ordenamientos) sobre objetos que implementan alguna de las interfaces de colecciones. Son métodos **polimórficos** es decir el mismo método se usa sobre diferentes implementaciones de las interfaces de colecciones. Son unidades funcionales reusables.

El **framework de colecciones** se incorporó en JAVA a partir de la versión 1.2



# Colecciones en JAVA

**Reduce la programación:** provee estructuras de datos y algoritmos útiles. Facilita la interoperabilidad entre APIs no relacionadas evitando escribir adaptadores o código de conversión para conectar APIs.

**Provee estructuras de datos de tamaño no-limitado:** es posible almacenar la cantidad de objetos que se desee.

**Aumenta la velocidad y calidad de los programas:** provee implementaciones de estructuras de datos y algoritmos de alta *performance y calidad*. Las diferentes implementaciones de las interfaces son intercambiables pudiendo los programas adaptarse a diferentes implementaciones.

**Permite interoperabilidad entre APIs no relacionadas:** establece un lenguaje común para pasar colecciones de elementos.

**Promueve la reusabilidad de software:** las interfaces del framework de colecciones y los algoritmos que manipulan las implementaciones de las interfaces son reusables.

El *framework* de colecciones de JAVA está formado por un conjunto de clases e interfaces ubicadas mayoritariamente en el paquete **java.util**.

El *framework* de colecciones de JAVA a partir de JAVA 5 soporta **tipos genéricos**. El **compilador inserta castings** automáticamente y realiza comprobación de tipos cuando se agregan elementos, evitando errores que sólo podían detectarse en ejecución. Como resultado los **programas son más seguros y claros**.

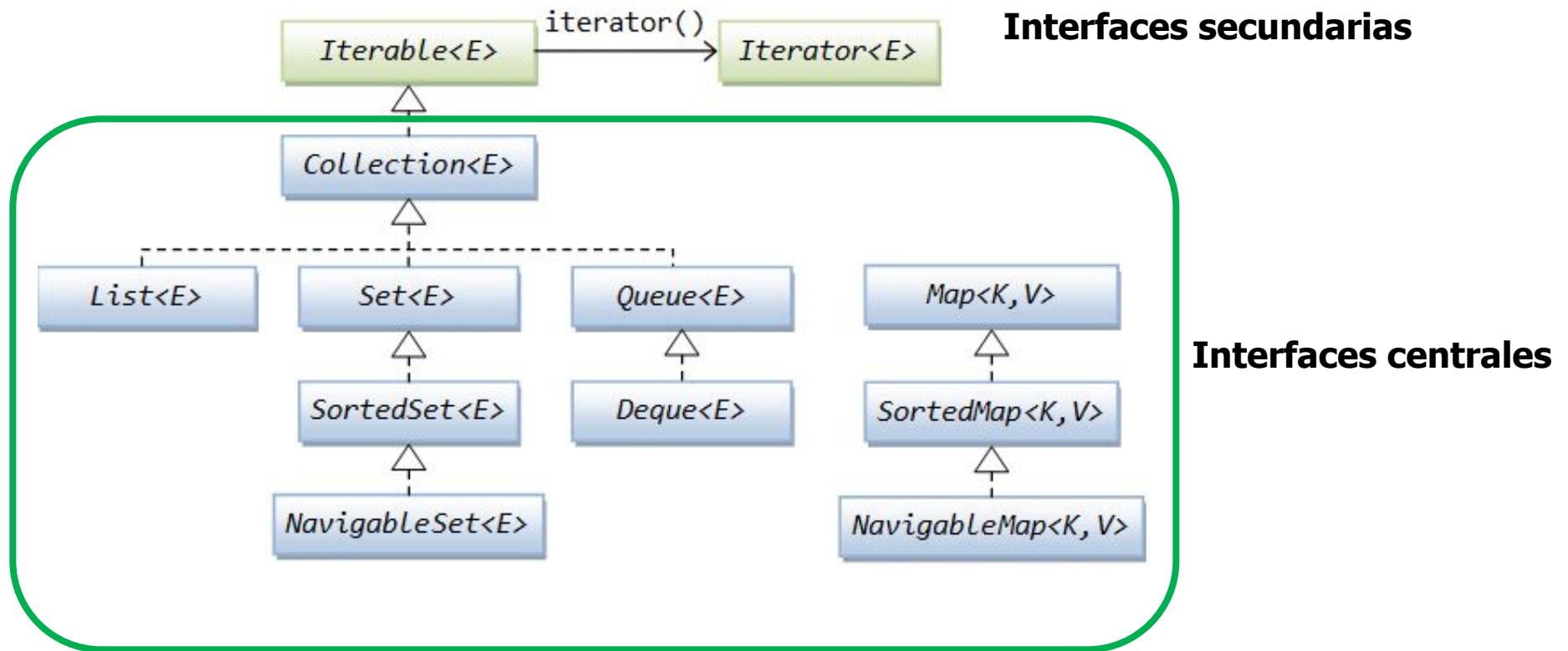
# Colecciones y Genéricos en JAVA

Los **tipos genéricos en JAVA** fueron **incorporados** fundamentalmente para **implementar colecciones genéricas**.

El **framework de colecciones no-genérico en JAVA** no ofrecía ninguna forma de colecciones homogéneas. Todas las colecciones contenían elementos de tipo **Object** y por esa razón eran de **naturaleza heterogénea**, mezcla de objetos de diferente tipo. Esto se puede observar desde la API de colecciones (jse 4): las colecciones no-genéricas aceptan objetos de cualquier tipo para insertar en la colección y retornan una referencia a un **Object** cuando un elemento es recuperado de una colección.

El **framework de colecciones genérico de JAVA** permite implementar **colecciones homogéneas**. Este framework se define a través de **interfaces genéricas y clases genéricas** que pueden ser instanciadas por una gran variedad de tipos. Por ejemplo, la interface genérica **List<E>** puede ser parametrizada como una **List<String>**, **List<Integer>**, cada una de ellas es una lista homogénea de valores strings, enteros, etc. A partir de la clase genérica **ArrayList<E>** puede obtenerse la clase parametrizada **ArrayList<String>**, **ArrayList<Double>**, etc.

# Interfaces Centrales y Secundarias



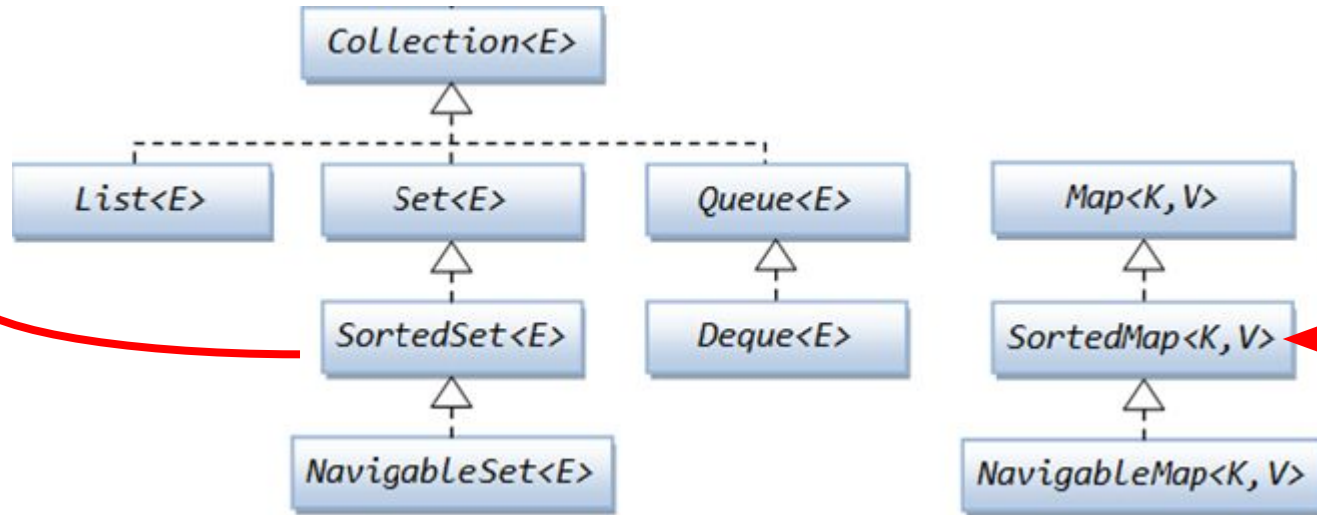
Las **interfaces centrales** especifican los **múltiples contenedores** de elementos y las **interfaces secundarias** especifican las **formas de recorrido** de las colecciones.

Las **interfaces centrales** permiten a las colecciones ser manipuladas independientemente de los detalles de implementación.

A partir de `Collection` y `Map` se definen dos jerarquías de interfaces que constituyen el fundamento del framework de colecciones de JAVA

# Interfaces Centrales

Conjunto  
especializado  
que mantiene  
sus elementos  
ordenados



Es un Map  
especializado  
que mantiene  
sus elementos  
ordenados

Todas las interfaces de colecciones son **genéricas**.

Encapsulan distintos tipos de colecciones de objetos y son el **fundamento del framework de colecciones de Java**. Pertenecen al paquete **java.util**. Las interfaces centrales son:

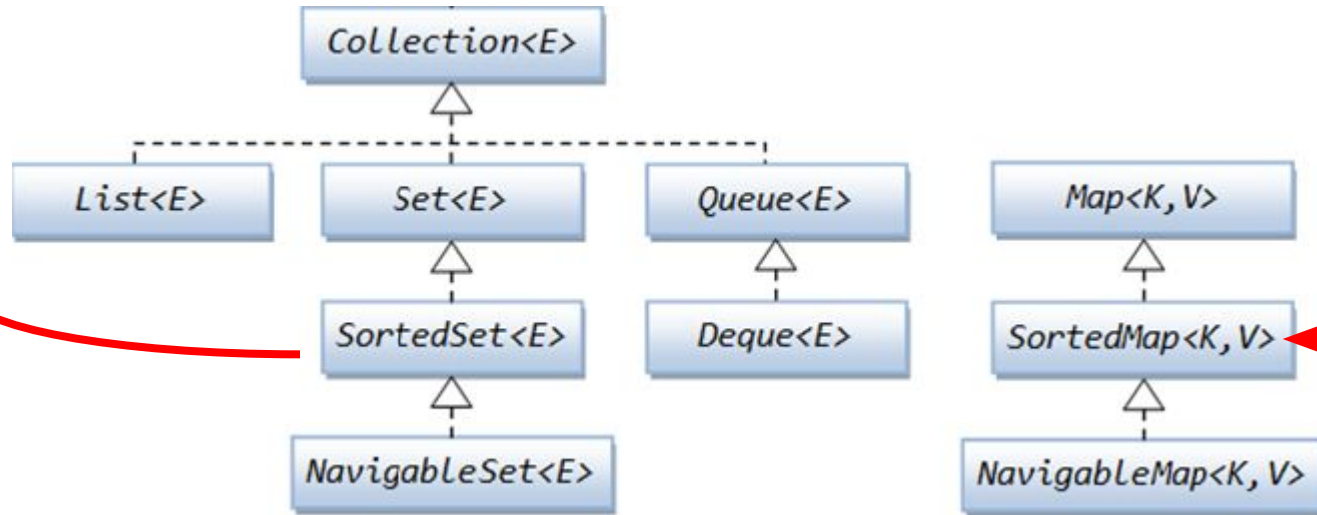
**Collection:** generaliza el concepto de grupos de objetos llamados elementos. Es la raíz de la jerarquía de colecciones. La plataforma Java no provee una implementación directa para la interface Collection pero sí para sus subinterfaces **Set**, **List** y **Queue**.

**Set:** es una colección que **no contiene duplicados**. Modela el concepto de conjunto matemático y es usado para representar conjuntos por ej. los materias que cursa un estudiante, los procesos en una computadora.

**List:** guarda los elementos en el mismo orden en que fueron insertados, permite elementos duplicados. También llamada **secuencia**. Provee acceso indexado.

# Interfaces Centrales

Conjunto especializado que mantiene sus elementos ordenados



Es un Map especializado que mantiene sus elementos ordenados

**Queue:** mantiene los elementos previamente a ser procesados. Agrega **operaciones adicionales para inserción, extracción e inspección de elementos**. Típicamente los elementos de una Queue **están ordenados usando una estrategia FIFO** (First In First Out). Existen implementaciones de colas de prioridades.

**Deque:** puede ser usada con estrategia FIFO (First In First Out) y como una LIFO (Last In First Out). Todos los elementos pueden ser insertados, recuperados y removidos de ambos extremos.

**Map:** representa asociaciones entre objetos de la forma clave- valor. **No permite claves duplicadas**. Cada clave está asociada a lo sumo con un valor. También llamada **diccionario**. Tiene similitudes con **Hashtable**.

## Versiones de ordenadas de Set y Map:

**SortedSet:** es un Set que mantiene todos los elementos ordenados en orden ascendente. Se agregan métodos adicionales para sacar ventaja del orden. Se usan para conjuntos ordenados naturalmente.

**SortedMap:** es un Map que mantiene sus asociaciones ordenadas ascendentemente por clave. Son usados para colecciones de pares clave-valor naturalmente ordenados (diccionarios, directorios telefónicos).



# Implementaciones de propósito general

## Implementación de SortedSet

Interfaces	Implementaciones				
	Tabla de Hashing	Arreglo de tamaño variable	Árbol	Lista Encadenada	Tabla de Hashing + Lista Encadenada
Set	HashSet	EnumSet	TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue			PriorityQueue		
Map	HashMap	EnumMap	TreeMap		LinkedHashMap

## Implementación de SortedMap

Las implementaciones de propósito general definen todos los métodos opcionales. Permiten elementos, claves y valores **null**. No son *thread-safe*. Todas son serializables. La regla es pensar en términos de interfaces y no de implementaciones.

**HashMap:** es una implementación de un Map basada en una tabla de hashing. Las operaciones básicas (get() y put()) tienen tiempo de ejecución constante  $O(1)$ .

**HashSet:** es una implementación de Set basada en una tabla de hash, un HashSet especializado. No hay garantías acerca del orden de ejecución en las distintas iteraciones sobre el conjunto. Las operaciones básicas (add(), remove(), size(), contains()) tienen tiempo de ejecución constante  $O(1)$ .

**ArrayList:** es una implementación de List basada en arreglos de longitud variable, es similar a Vector. Las operaciones size(), isEmpty(), get(), set(), iterator(), y listIterator() tienen tiempo de ejecución constante  $O(1)$ . La operación add() tiene tiempo de ejecución lineal  $O(n)$ .

Hay dos implementaciones de **Queue**: **PriorityQueue** y **LinkedList** (también implementa List)

# Interface Collection

`public interface Collection<E> extends Iterable<E> {` **Es una interface genérica que representa un agrupamiento de objetos de tipo E**

## //Operaciones para Agregar Elementos

`boolean add(E element);`

`boolean addAll(Collection<? extends E> c);`

## //Operaciones para Eliminar Elementos

`boolean remove(Object element);`

`void clear();`

`boolean removeAll(Collection<?> c);`

`boolean retainAll(Collection<?> c);`

## //Operaciones de Consultas

`int size();`

`boolean isEmpty();`

`boolean contains(Object element);`

`boolean containsAll(Collection<?> c);`

## //Operaciones que facilitan el Procesamiento

`Iterator iterator();`

`Object[] toArray();`

`<T> T[] toArray(T[] a);`

## //Operaciones para obtener Stream

`default Stream<E> parallelStream();`

`default Stream<E> stream();`

Cuando se declara un objeto Collection es posible establecer el tipo de dato que se almacenará en la colección y de esta manera se evita la necesidad de *castear* cuando se leen los elementos. Se evitan errores de *casting* en ejecución y se le da al compilador información sobre el tipo usado para poder hacer un chequeo fuerte de tipos.

```
import java.util.*;
public class ColeccionSimple {
    public static void main(String [] args){
        Collection<Integer> c=new ArrayList<>();
        for (int i=0; i < 10; i++){
            c.add(i);
        }
    }
}
```

## Auto-Boxing

El framework de colecciones de JAVA NO provee ninguna implementación de Collection, sin embargo es una interface muy importante pues establece un comportamiento común para todas las subinterfaces. Permite convertir el tipo de las colecciones.

# Recorrer una Colección

## 1) Usando el constructor *for-each*.

El constructor **for-each** provee una manera concisa de recorrer colecciones y arreglos. Se incorporó en la versión JSE 5. Es la manera preferida

### Autoboxing/Unboxing

Convierte automáticamente datos de tipos primitivos (como int) a objetos de clases *wrappers* (como Integer) cuando se inserta en la colección y, convierte instancias de clases *wrappers* en primitivos cuando se leen elementos de la colección.

Es una característica provista a partir de **j2se 5**

El **for-each** funciona bien con cualquier cosa que produzca un iterador. En Java 5 la interface **Collection** extiende **Iterable**, entonces cualquier implementación de **Set**, **List** y **Queue** puede usar el **for-each**

```
public Iterable<T> {  
    Iterator<T> iterator();  
}
```

```
public class DemoColeccion {  
    public static void main(String[] args) {  
        List<Integer> lista= new ArrayList<>();  
        lista.add(1);  
        lista.add(2);  
        lista.add(190);  
        lista.add(90);  
        lista.add(7);  
        for (int i: lista)  
            System.out.println(i);  
    }  
}
```

# Recorrer una Colección

## 2) Usando la interface Iterator

Un objeto **Iterator** permite recorrer secuencialmente una colección y remover elementos selectivamente si se desea. Es posible obtener un iterador para una colección invocando al método **iterator()**. Una colección es un objeto **Iterable**.

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

*Interfaces secundarias del  
framework de colecciones de  
java*

```
public interface ListIterator<E> extends Iterator<E>  
{  
    boolean hasPrevious();  
    int nextIndex();  
    E previous();  
    int previousIndex();  
    void add(E elemento); //opcional  
    void set(E elemento); //opcional  
    void remove();       //opcional  
}
```

```
public Iterable<T> {  
    Iterator<T> iterator();  
}
```

```
public class Demolterador{  
    public static void main(String[] args) {  
        List<Integer> lista= new ArrayList<>();  
        lista.add(1);  
        lista.add(2);  
        lista.add(190);  
        lista.add(90);  
        lista.add(7);  
        Iterator it= lista.iterator();  
        while (it.hasNext())  
            System.out.println(it.next());  
    }  
}
```

**NO es la manera preferida de  
escribir código para recorrer  
secuencias en JAVA 5**

# Interface List

Un objeto **List** es una **secuencia de elementos**, cada uno tiene una posición, permite duplicados y los elementos se almacenan en el mismo orden en que son insertados. La interface **List** define métodos para recuperar y agregar elementos en una posición determinada o índice.

```
public interface List<E> extends Collection<E> {
```

## // Métodos de Acceso Posicional

```
void add(int index, E element);
boolean addAll(int index, Collection<? extends E> c);
E get(int index);
E remove(int index);
E set(int index, E element);
```

## // Métodos de Búsqueda

```
int indexOf(Object o);
int lastIndexOf(Object o);
```

## // Métodos de Iteración

```
ListIterator<E> listIterator();
ListIterator<E> listIterator(int index);
```

## // Método de Rangos de Vistas

```
List<E> subList(int from, int to);
```

```
}
```

ListIterator es un iterador que aprovecha las ventajas de la naturaleza secuencial de la lista: permite recorrer la lista en cualquier dirección (hacia adelante y hacia atrás), modificarla durante el recorrido y obtener la posición del elemento actual.

```
public class Demolterador{
public static void main(String[] args) {
    List <Number> lista=new ArrayList<>();
    lista.add(10);
    lista.add(200.0f);
    lista.add(100.0);
    Set<Float> setDecimales= new HashSet<>();
    setDecimales.add(100.0F);
    setDecimales.add(200.0F);
    setDecimales.add(300.0F);
    lista.addAll(setDecimales);
    ListIterator it= lista.listIterator(lista.size());
    while (it.hasPrevious())
        System.out.println(it.previous());
    }
}
```



# Interface Set

Un objeto **Set** es una colección de objetos **sin duplicados**: no contiene 2 referencias al mismo objeto, 2 referencias a **null** o referencias a 2 objetos a y b tal que a.equals(b). El propósito general de las implementaciones de **Set** son colecciones de objetos sin orden. Sin embargo existen conjuntos con orden, SortedSet.

```
public interface Set<E> extends Collection<E> {  
    boolean add(E o);  
    boolean addAll(Collection<? extends E> c);  
    void clear( );  
    boolean contains(Object o);  
    boolean containsAll(Collection<?> c);  
    boolean equals(Object o);  
    int hashCode( );  
    boolean isEmpty( );  
    Iterator<E> iterator( );  
    boolean remove(Object o);  
    boolean removeAll(Collection<?> c);  
    boolean retainAll(Collection<?> c);  
    int size( );  
    Object[] toArray( );  
    <T> T[] toArray(T[] a);  
}
```

## Ejemplo

```
public class TestSortedSet {  
    public static void main (String[] args) {  
        SortedSet<String> s = new TreeSet<>(Arrays.asList(args));  
        // Itera: los elementos son ordenados automáticamente  
        for(String palabra : s)  
            System.out.println(palabra);  
        // Recuperate elementos especiales  
        String primero = s.first(); // Primer Elemento  
        String ultimo = s.last(); // Último Elemento  
    }  
}
```

```
java TestSortedSet hola chau adios quetal elena hola chau elena luis
```

¿Cuál es la salida?

# Interface Map

Es la única interface que no hereda de Collection. Un objeto **Map** es un conjunto de asociaciones clave-valor. Las claves son únicas y cada clave está asociada con a lo sumo un valor. Es un tipo genérico con 2 parámetros que representan tipos de datos: **K** es el tipo de las claves y **V** el tipo de los valores.

## Implementación de la interface Map

```
public interface Map <K,V> {  
// Agregar Asociaciones  
    V put(K clave, V valor);  
    void putAll(Map<? extends K, ? extends V> t);  
// Eliminar Asociaciones  
    void clear();  
    V remove(K clave);  
// Consultar el Contenido  
    V get(K clave);  
    boolean containsKey(K clave);  
    boolean containsValue(V valor);  
    int size();  
    boolean isEmpty();  
//Proveer Vistas de Claves, Valores o Asociaciones  
    Set<Map.Entry<K, V>> entrySet();  
    Set<K> keySet();  
    Collection<V> values();  
}
```

```
public class DemoMap {  
    public static void main(String[] args) {  
        Map<Integer, Alumno> tablaAlu = new HashMap<> ();  
        Alumno[] arregloAlumno =  
        { new Alumno(789, "Luis", "Gomez"),  
          new Alumno(321, "Elena", "Martinez"),  
          new Alumno(123, "Claudia", "Gutierrez"),  
          new Alumno(123, "Claudia", "Gutierrez"),  
          new Alumno(321, "Elena", "Lopez"),  
          new Alumno(456, "Manuel", "Martinez"),  
          new Alumno(789, "Luis", "Gomez") };  
        for (Alumno unAlu: arregloAlumno)  
            tablaAlu.put(unAlu.getLegajo(), unAlu);  
        Collection<Alumno> listAlu= tablaAlu.values();  
        for (Alumno unAlu: listAlu)  
            System.out.println(unAlu);  
    }  
}
```

Cambiando la instanciación por un objeto **TreeMap()** obtenemos una colección ordenada::

```
Map<String, Integer> numeros=new TreeMap<String, Integer>();
```

Aunque no es una Collection sus claves pueden ser recuperadas como un Set, sus valores como Collection y sus asociaciones como un Set de Map.Entry (es una interface anidada en Map que representa pares clave-valor)

# Interface Queue

Un objeto **Queue** es una colección diseñada para mantener elementos que esperan por procesamiento. Además de las operaciones de **Collection** provee operaciones para insertar, eliminar e inspeccionar elementos. No permite elementos nulos. La plataforma java provee una implementación de **Queue** en el paquete **java.util**, **PriorityQueue**, que es una cola con prioridades y varias en el paquete **java.util.concurrent** como **DelayQueue** y **BlockingQueue** que implementan diferentes tipos de colas, ordenadas o no, de tamaño limitado o ilimitado, etc.

PriorityQueue chequea que los objetos que se insertan sean **Comparables!!**

```
public interface Queue<E> extends Collection<E>{  
    boolean add(E e);  
    E element();  
    E remove();  
    E peek();  
    boolean offer(E e);  
    E poll();  
}
```

Recupera, pero no elimina la cabeza de la cola.

Inserta el elemento en la cola si es posible

Recupera y elimina la cabeza de la cola.

```
public class DemoQueue{  
    public static void main(String[] args) {  
        Queue<String> pQueue = new PriorityQueue<>();  
        pQueue.offer("Buenos Aires");  
        pQueue.offer("Montevideo");  
        pQueue.offer("La Paz");  
        pQueue.offer("Santigao");  
        System.out.println(pQueue.peek());  
        System.out.println(pQueue.poll());  
        System.out.println(pQueue.peek());  
    }  
}
```

¿Cuál es la salida?

# ¿Cómo las interfaces SortedSet y SortedMap mantienen el orden de sus elementos?

De acuerdo al ordenamiento natural de sus elementos (en el caso de SortedMap de sus claves) o a un comparador de ordenación provisto en el momento de la creación (**Comparator**).

Las interfaces **Comparable** y **Comparator** permiten comparar objetos y de esta manera es posible ordenarlos.

Múltiples clases de la plataforma Java implementan la interface **Comparable**, entre ellas: String, Integer, Double, Float, Boolean, Long, Byte, Character, Short, Date, etc.

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

Interfaces secundarias del  
framework de colecciones de  
java

Retorna: cero (0), un valor  
negativo o uno positivo

```
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals(Object obj)  
}
```

```
public class Alumno implements Comparable<Alumno> {  
    private int legajo;  
  
    public int compareTo(Alumno otro) {  
        int resultado=0;  
        if (this.getLegajo() < otro.getLegajo())  
            resultado = -1;  
        if (this.getLegajo() > otro.getLegajo())  
            resultado = 1;  
        return resultado;  
    }  
}
```

# Ejemplos de Set con Orden y sin Orden

```
public class DemoIterador{  
    public static void main(String[] args) {  
        Set<String> instrumentos= new HashSet<>();  
        instrumentos.add(" Piano");  
        instrumentos.add(" Saxo");  
        instrumentos.add(" Violin");  
        instrumentos.add(" Flauta");  
        instrumentos.add(" Flauta");  
        System.out.println(instrumentos.toString());  
    }  
}
```

La interface Set es útil para crear colecciones sin duplicados desde una colección **c** con duplicados:  
`Set<String> sinDup=new TreeSet<>(c);`

salida  
a **[Violin, Piano, Saxo, Flauta]**

Implementa **SortedSet**

Cambiando únicamente la instanciación por un objeto **TreeSet()** obtenemos una colección ordenada:

```
Set<String> instrumentos= new TreeSet<String>();
```

salida  
a **[Flauta, Piano, Saxo, Violin]**

En este caso el compilador chequea que los objetos que se insertan con el método add() sean **Comparables!!**



# Anotaciones & Reflection

# Anotaciones ¿Qué son?

- A partir de la versión 5, JAVA incorpora un tipo de dato nuevo llamado **anotaciones**. Las anotaciones son un **tipo especial de interface** que se utiliza para “**anotar**” **declaraciones**.
- Las anotaciones son **metadatos** que proveen a nuestros programas de información extra que es **testeada y verificada** en **compilación**. **Son metadatos del código**. Proporcionan información para describir nuestro programa que no puede expresarse en código JAVA.
- Las **anotaciones** pueden aplicarse a **paquetes, clases, interfaces, enumerativos, variables, parámetros, métodos, constructores**, en general a diferentes elementos de nuestros programas.
- El objetivo de las anotaciones es facilitar la **combinación** de **metadatos** con **código fuente (JAVA)**, en lugar de mantenerlos en archivos separados.
- Las **anotaciones** pueden ser **leídas desde el código fuente, desde archivos .class y usando el mecanismo de reflection**.
- El **código anotado no es afectado** directamente por sus anotaciones. Éstas proveen información para otros sistemas.
- El desarrollo basado en anotaciones alienta al estilo de **programación declarativa**, donde el programador dice lo qué debe hacerse y las herramientas lo hacen automáticamente (producen el



# La anotación predefinida: @Override

Una **anotación** es una **instancia de un tipo anotación** y **asocia metadatos** con un **elemento** de la aplicación. Se expresa en el código fuente con el prefijo @.

El compilador JAVA soporta los siguientes tipos de anotaciones definidas en `java.lang`:

## @Override

```
public class Subclass extends Base {  
    @Override  
    public void m() {  
        // TODO Auto-generated method stub  
        super.m();  
    }  
}
```

```
public class Subclass extends Base {  
    @Override  
    public void m() {  
        // TODO  
        super.m();  
    }  
    @Override  
    public void x() {  
        // TODO  
    }  
}
```

Error de Compilación: The method x() of type Subclass must override or implement a super type method

**Declaramos que sobrescribimos el método m() definido en la superclase, Base.** El compilador examina la superclase (Base) y garantiza que m() está definido.

```
public class Base {  
  
    public void m() {  
        // TODO Auto-generated method stub  
    }  
}
```

Las anotaciones **@Override** son útiles para indicar que un método de una subclase sobrescribe un método de la superclase y no lo sobrecarga por ejemplo.

# @Override

**@Override** es una anotación para el compilador.

La anotación **@Override** en la declaración de un método que sobrescribe una declaración de un supertipo (clase o interface), permite que el **compilador nos ayude a evitar errores**.

Las **IDEs** proveen, además, **chequeos automáticos** de código conocidos como “**inspección de código**”. Si se habilita la “inspección de código”, el **IDE generará *warnings*** si un método sobrescribe un método de la superclase y no tiene la anotación **@Override**.

Si se usa la anotación **@Override** **consistentemente**, los *warnings* de los *chequeos del IDE* nos alertarán de **sobreescrituras no intencionales**.

Los *warnings* de los **IDEs** complementan los **mensajes de error del compilador**: se garantiza que estamos sobrescribiendo los métodos en el lugar que deseamos hacerlo.



# @Deprecated

La anotación **@Deprecated** es útil para indicar que el elemento marcado será reemplazado por otro en futuras versiones. El compilador advierte (o emite un error) cuando un elemento anotado como “deprecated” es accedido por un código que está en uso. Puede aplicarse a métodos, clases y propiedades.

## @Deprecated

```
public class Base {  
    @Deprecated  
    public void s(){  
        System.out.println("Hola");  
    }  
    public void m() {  
        System.out.println("Método m()");  
    }  
}  
  
public class UseBase {  
  
    public static void main(String[] args) {  
        new Base().s();  
    }  
}
```

Indica que el elemento marcado con la anotación **@Deprecated** está **desaprobado** y se dejará de usar. El compilador genera una advertencia o un error (de acuerdo a la configuración del IDE) cada vez que un programa lo usa, el objetivo es **desalentar su uso**.

The method s() from type Base is deprecated

# @SuppressWarnings

La anotación **@SuppressWarnings** es útil para eliminar advertencias del compilador a ciertas partes del programa. Las advertencias que pueden suprimirse varían entre las diferentes IDEs, los más comunes son: "deprecation", "unchecked", "unused".

## @SuppressWarnings

```
import java.util.Date;
public class SuppressWarningsExample {
    @SuppressWarnings(value={"deprecation"})
    public static void main(String[] args) {
        Date date = new Date(2016, 9, 30);
        System.out.println("date = " + date);
    }
}
```

Si comentamos la anotación **SuppressWarnings**, obtendríamos un mensaje de advertencia del compilador:

Warning(10,25): constructor Date(int, int, int) is deprecated

**Suprime determinadas advertencias de compilación** en el elemento anotado y sus subelementos.

Advertencias "unchecked" ocurren cuando se mezcla código que usa tipos genéricos con código que no lo usa.

Advertencias "unused" ocurren cuando no se usa un método o una variable.

[Lista de advertencias que pueden suprimirse en Eclipse](#)



# @FunctionalInterface

La anotación **@FunctionalInterface** asegura que la **interface funcional** define un solo **método abstracto**. En el caso de que haya más de un método abstracto, el compilador emitirá el siguiente mensaje de error: **"Invalid @FunctionalInterface annotation"**. No es obligatorio utilizar esta anotación.

## @FunctionalInterface

```
package anotaciones;  
@FunctionalInterface  
public interface Square {  
    int calculate(int x);  
    int otro();  
}
```

La **compilación falla** y el compilador emitirá el siguiente mensaje de error:

Invalid '@FunctionalInterface' annotation; Square is not a functional interface

# Declarar Anotaciones

Declarar una anotación es similar a declarar una interface: el carácter @ precede a la palabra clave **interface** (@ = "AT" Annotation Type). Las anotaciones se compilan a archivos .class de la misma manera que las interfaces, clases y tipos enumerativos.

```
public @interface SolicitudDeMejora{
    int id();
    String resumen();
    String ingeniera() default "[no asignado]";
    String fecha() default "[no implementado]";
}
```

## Definición de la ANOTACIÓN SolicitudDeMejora

El cuerpo de la declaración de las anotaciones contiene **declaraciones de elementos** que permiten especificar valores para las anotaciones (parámetros). Los programas o herramientas usarán los valores de los elementos o parámetros para procesar la anotación.

Una vez que definimos un tipo de anotación la podemos usar para **anotar declaraciones**:

```
@ SolicitudDeMejora(
id = 2868724,
resumen= "Habilitado para viajar en el tiempo",
ingeniera = "Elisa Bachofen",
fecha = "6/10/2021" )
public static void viajarEnElTiempo(Date destino) { }
```

**El método viajarEnElTiempo()  
está anotado con la anotación  
SolicitudDeMejora**

El código precedente no hace nada por sí mismo, el compilador verifica la existencia del tipo de anotación @SolicitudDeMejora.

Las anotaciones consisten de un @ seguido por un tipo de anotación y una lista entre paréntesis de pares elemento-valor.

Los valores de los elementos deben ser constantes predefinidas en compilación.



# Anotaciones con un único elemento

Para las anotaciones que definen un único elemento podemos usar el nombre **value** y de esta manera cuando anotamos un elemento omitimos el nombre del elemento seguido del signo igual (=).

```
/*Asocia un copyright al elemento anotado*/  
public @interface Copyright {  
    String value();  
}
```

## Definición de la ANOTACIÓN Copyright

El elemento de nombre **value** es el único que no requiere el uso de la sintaxis elemento-valor para anotar declaraciones, alcanza con especificar el valor entre paréntesis:

```
@Copyright("2022 Sistema de Auditoria de Juegos de Azar")  
public class MaquinasDeJuego{ }
```

## La clase

**MaquinasDeJuego está  
anotada con la  
anotación Copyright**

Esta sintaxis abreviada sólo puede usarse cuando la anotación define un único elemento y de nombre **value**.

El nombre **value** puede aplicarse a cualquier elemento y de esta manera cuando anotamos una declaración omitimos el nombre del elemento seguido del signo igual (=).



# Anotaciones *Marker*

Las Anotaciones *Markers* NO contienen elementos, son útiles para marcar elementos de una aplicación con algún propósito.

## Definición de las anotaciones *markers* “InProgress” y “Test”:

```
/*Indica que el elemento anotado está sujeto a cambios, es una versión preliminar */  
public @interface InProgress { }
```

La anotación **@Test** la usaremos para realizar **testeos** que se ejecutan **automáticamente**. Cuando un método falla se disparan excepciones.

```
import java.lang.annotation.*;  
/**  
 * Indica que el método anotado es un método de testeo.  
 * Se usa sólo en métodos estáticos y sin argumentos.  
 */  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Test {  
}
```

↓  
**¿Puede el compilador garantizarlo?**

# Anotaciones *Marker*

## Uso de las anotaciones @InProgress y @Test:

```
@InProgress public class ViajeEnElTiempo{ }
```

```
public class Ejemplo {  
    @Test public static void m1() { }  
    public static void m2() { }  
    @Test public static void m3() { throw new RuntimeException("Boom");}  
    public static void m4() { }  
    @Test public void m5() { }  
    public static void m6() { }  
    @Test public static void m7() {throw new RuntimeException("Crash");}  
    public static void m8() { }  
}
```

Las anotaciones *marker* simplemente “marcan” el elemento anotado.

Si el programador escribe mal **Test** o la aplica a un elemento que no es la declaración de un método, el programa no compilará.

Se puede omitir el paréntesis cuando usamos una anotación *marker*.

La anotación **Test** no tiene efecto directo sobre la semántica de la clase **Ejemplo**, sólo sirve para proveer información a herramientas de testing.

# Anotaciones

## ¿Para qué se usan?

Las anotaciones no contienen ningún tipo de lógica y no afectan el código que anotan. Por ejemplo las anotaciones **@Inprogress** y **@Test** no tienen efecto directo sobre la semántica de las clases que las usan como **ViajeEnElTiempo** y **Ejemplo**.

Las anotaciones son usadas por los **procesadores o consumidores de anotaciones o parsers de anotaciones**, que son aplicaciones o sistemas que hacen uso del código anotado y ejecutan diferentes acciones dependiendo de la información suministrada.

**Ejemplos de procesadores de anotaciones:** la herramienta **JUnit**, que lee y analiza las clases de testeo anotadas y decide por ejemplo en qué orden serán ejecutadas las unidades de testeo. **Hibernate** usa anotaciones para mapear objetos a tablas de la BD.

Las anotaciones pueden ser procesadas en compilación por herramientas de pre-compilación y luego descartadas, o **en ejecución usando *reflection***.

Los procesadores de anotaciones usan ***reflection*** para leer y analizar el código anotado en ejecución.

Las anotaciones son compiladas a **.class** y recuperadas en ejecución y usadas por los procesadores de anotaciones que hacen uso de la información suministrada.

En el caso de la anotación **@Test** podríamos pensar en un **testeador “de juguete”** que analiza el contenido de una clase dada y **ejecuta** todos los métodos anotado con **@Test**. Lo vamos a llamar **RunTests**.





# Testeador: RunTests

```
package anotaciones;
import java.lang.reflect.*;
public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class <?> testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " falló: " + exc);
                } catch (Exception exc) {
                    System.out.println("INVALIDO @Test: " + m);
                }
            }
        }
        System.out.printf("Pasó: %d, Falló: %d\n", passed, tests - passed);
    }
}
```

## ¿Qué es Reflection?

Es la capacidad de **inspeccionar, analizar y modificar** código en ejecución.

Indica a RunTests qué método ejecutar

## ¿Cuál es el resultado?

**java RunTests Ejemplo**

# Anotaciones & Reflection

La facilidad de reflection: el paquete **java.lang.reflect** ofrece **acceso** programático a información de las **clases cargadas en ejecución**.

El **punto de entrada** de reflection es el objeto **Class**: contiene métodos para recuperar información de constructores, métodos, variables, etc. Es posible crear instancias, invocar métodos, acceder a variables de instancia.

La **API de Reflection** fue **extendida en JAVA 5** para **soporte** de lectura de código de **anotaciones en ejecución**:

- la interface **java.lang.reflect.AnnotatedElement** implementada por las clases: **Class**, **Constructor**, **Field**, **Method** y **Package**, provee acceso a anotaciones retenidas en ejecución mediante los métodos: **getAnnotation()**, **getAnnotations()** y **isAnnotationPresent()**.

# La Anotación @ExceptionTest

```
package anotaciones;
import java.lang.annotation.*;
/**
 * Indica que el método anotado es un método de testeo
 * que se espera dispare la excepción consignada como parámetro
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Throwable> value();
}
```

```
package anotaciones;
public class Ejemplo2 {
    @ExceptionTest(ArithmeticException.class)
    public static void m1() {
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m2() {
        int[] a = new int[0];
        int i = a[1];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { }
```

El tipo del elemento **value()** es un objeto **Class** para cualquier **subclase de Exception**

Agregamos soporte para que las **pruebas que tengan éxito** sean las que arrojan una **excepción particular**, por eso esta anotación tiene parámetros.

# Testeador de Excepciones: RunTests2

```
package anotaciones;
import java.lang.reflect.*;
public class RunTests2 {
public static void main(String[] args) throws Exception {
    int tests = 0;
    int passed = 0;
    Class<?> testClass = Class.forName(args[0]);
    for (Method m : testClass.getDeclaredMethods()) {
        if (m.isAnnotationPresent(ExceptionTest.class)) {
            tests++;
            try {
                m.invoke(null);
                System.out.printf("Test %s Falló: no hay excepciones%n", m);
            } catch (InvocationTargetException wrappedEx) {
                Throwable exc = wrappedEx.getCause();
                Class<? extends Throwable> excType = m.getAnnotation(ExceptionTest.class).value();
                if (excType.isInstance(exc)) {
                    passed++;
                } else {
                    System.out.printf( "Test %s Falló: se esperaba %s, ocurrió %s%n", m, excType.getName(), exc);
                }
            } catch (Exception exc) {
                System.out.println("INVALIDO @Test: " + m);
            }
        }
    }
    System.out.printf("Pasó: %d, Falló: %d%n", passed, tests - passed);
}
```

Se obtiene el  
valor del  
parámetro de la  
anotación  
**ExceptionTest**

¿Cuál es el resultado?

**java RunTest2 anotaciones.Ejemplo2**



# @ExceptionTest extendido

Consideremos que las pruebas que pasan son las que disparan al menos una excepción de un conjunto de excepciones.

¿Cómo haríamos?

```
package anotaciones;
import java.lang.annotation.*;
/**
 * Indica que el método anotado es un método de testeo
 * que se espera dispare la excepción consignada como parámetro
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Throwable> [] value();
}
```

El tipo del elemento **value()** es un arreglo de objetos **Class** para cualquier **subclase de Exception**

```
package anotaciones;
public class Ejemplo3 {
    @ExceptionTest({ IndexOutOfBoundsException.class, NullPointerException.class})
    public static void doublyBad() {
        List<String> list = new ArrayList<>();
        list.addAll(5, null);
    }
}
```

# Las meta-annotaciones

La declaración de anotaciones requiere de **meta-annotaciones** que indican **cómo será usada la anotación**.

Declaración de la anotación **@CasoDeUso**.

```
package anotaciones;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface CasoDeUso {
    public int id ( ) ;
    public String descripcion ( ) default "no hay descripción";
}
```

Metaanotaciones

**@Target:** indica **dónde se aplica** la anotación (métodos, clases, variables de instancia, variables locales, paquetes, constructores, etc).

**@Retention:** indica **dónde están disponibles** las anotaciones (código fuente, archivos .class o en ejecución).

Es el tiempo de vida de las anotaciones. **Dónde se usan.**

En nuestro ej. la anotación se aplica a **métodos** y el

**RetentionPolicy.RUNTIME** indica que los declaraciones anotadas con **CasoDeUso** son retenidas por la JVM y se pueden leer vía *reflection* en ejecución.

La declaración de los **elementos** permite declarar valores **predeterminados**.

Las declaraciones de **elementos** **no tienen parámetros ni cláusulas *throws***.

Las anotaciones que **no contienen elementos** se llaman **markers**.

**Un programa o una herramienta usa los valores de los elementos para procesar las anotaciones.**



# Uso de @CasoDeUso

```
package anotaciones;  
import java.util.List;
```

## 3 métodos anotados

```
public class UtilitarioPassw {  
    @CasoDeUso(id = 47, descripcion = "Passw deben contener al menos un número")  
    public boolean validarPassw(String password) {  
        return (password.matches("\\w*\\d\\w*"));  
    }  
}
```

Los valores de las anotaciones son expresados entre paréntesis como pares elemento-valor después de la declaración de @CasoDeUso.

```
    @CasoDeUso(id = 48)  
    public String encriptarPass(String password) {  
        return new StringBuilder (password).reverse().toString() ;  
    }  
}
```

Para la anotación del método encriptarPass() se omite el valor del elemento *descripcion*. Se usará el valor definido en @interface CasoDeUso

```
    @CasoDeUso(id = 49, descripcion = "Nuevas passw no pueden ser iguales a otras usadas ")  
    public boolean chequearPorNuevasPassw(List<String> prevPasswords, String password){  
        return prevPasswords.contains(password);  
    }  
}
```

Los métodos **validarPassw()**, **encriptarPass()** y **chequearPorNuevasPassw()** están anotados con **@CasoDeUso**. Las anotaciones se usan en combinación con otros modificadores como public, static, final. Por convención los preceden.

# RastreadorDeCasosDeUso

**Lista los casos de uso completados y localiza los faltantes**

```
package anotaciones;
import java.lang.reflect.*;
import java.util.*;

public class RastreadorDeCasosDeUso {
    public static void rastrearCasosDeUso(List<Integer> casosDeUso, Class<?> cl) {
        for (Method m : cl.getDeclaredMethods() ) {
            CasoDeUso uc = m.getAnnotation(CasoDeUso.class);
            if (uc != null) {
                System.out.println("Caso de Uso encontrado:" + uc.id() + " "+ uc.descripcion());
                casosDeUso.remove(new Integer(uc.id()));
            }
        }
        for (int i : casosDeUso)
            System.out.println("Advertencia: Falta el caso de uso-" + i);
    }

    public static void main(String[] args) {
        List<Integer> casosDeUso = new ArrayList<>();
        Collections.addAll(casosDeUso, 47, 48, 49, 50);
        rastrearCasosDeUso(casosDeUso, anotaciones.UtilitarioPassw.class);
    }
}
```

**Rastreador de casos de uso de un proyecto:**

- 1) los programadores anotan los métodos que cumplen los requerimientos de cada caso de uso.
- 2) el líder del proyecto usa el rastreador para conocer el grado de avance del proyecto contando la cantidad de casos de uso implementados.
- 3) los desarrolladores que mantienen el proyecto fácilmente pueden encontrar los casos de uso para actualizar o depurar reglas de negocio.

Caso de Uso encontrado:47 Passw deben contener al menos un número

Caso de Uso encontrado:48 no hay descripción

Caso de Uso encontrado:49 Nuevas passw no pueden ser iguales a otras ya usadas

Advertencia: Falta el caso de uso-50

**¿Cuál es la salida?**



# Meta-Anotaciones

Java provee 4 meta-annotaciones. Las meta-annotaciones se usan para anotar anotaciones

<b>@Target</b>	<p>Indica a qué elementos se le aplica la anotación. Los valores son los definidos en el enumerativo <b>ElementType</b>:</p> <p><b>ElementType.ANNOTATION_TYPE</b>: se aplica solamente a anotaciones</p> <p><b>ElementType.TYPE</b>: se aplica a la declaración de clases, interfaces, anotaciones y enumerativos.</p> <p><b>ElementType.PACKAGE</b>: se aplica a la declaración de paquetes.</p> <p><b>ElementType.CONSTRUCTOR</b>: se aplica a un constructor</p> <p><b>ElementType.FIELD</b>: se aplica a la declaración de un propiedad (incluye constantes <b>enum</b>)</p> <p><b>ElementType.LOCAL_VARIABLE</b>: se aplica a la declaración de variables locales</p> <p><b>ElementType.METHOD</b>: se aplica a la declaración de métodos</p> <p><b>ElementType.PARAMETER</b>: se aplica a los parámetros de un método.</p>
<b>@Retention</b>	<p>Indica dónde y cuánto tiempo se mantiene la anotación. Los valores son los definidos en el enumerativo <b>RetentionPolicy</b>:</p> <p><b>RetentionPolicy. SOURCE</b>: son retenidas en el código fuente y descartadas por el compilador. No están en los bycodes. Ej.: <b>@Override</b>, <b>@SuppressWarnings</b></p> <p><b>RetentionPolicy. CLASS</b>: son retenidas en compilación e ignoradas por la JVM. Se desechan durante la carga de la clase. Es el valor de defecto. Útil para procesar bytecodes.</p> <p><b>RetentionPolicy. RUNTIME</b>: son retenidas por la JVM en ejecución y pueden ser leídas mediante el mecanismo de reflexión. No son descartadas.</p>
<b>@Documented</b>	<p>Indica que la anotación se incluye en la documentación generada por el <b>javadoc</b>. Por defecto, las anotaciones no se incluyen en el javadoc.</p>
<b>@Inherited</b>	<p>Indica que la anotación es heredada automáticamente por todas las subclases de la clase anotada. Por defecto, las anotaciones no son heredadas por las subclases.</p>

# Meta-Anotación: @Retention

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
```

```
@Retention(RetentionPolicy.SOURCE)
```

```
public @interface SuppressWarnings {  
    String[] value();  
}
```

```
@Target(ElementType.METHOD)
```

```
@Retention(RetentionPolicy.SOURCE)
```

```
public @interface Override {  
}
```

En estos ejemplos el compilador (javac) es la herramienta que procesa la anotación. La anotaciones luego de ser procesadas son descartadas por el compilador. Aparecen sólo en el código fuente.

```
public class Subclase extends Base {  
    @Override  
    public void m() {  
        // TODO Auto-generated method stub  
        super.m();  
    }  
}
```

```
import java.util.Date;  
public class SuppressWarningsExample {  
    @SuppressWarnings(value={"deprecation"})  
    public static void main(String[] args) {  
        Date date = new Date(2008, 9, 30);  
        System.out.println("date = " + date);  
    }  
}
```

# Meta-Anotación: @Documented

```
package anotaciones;
import java.lang.annotation.Documented;
@Documented
public @interface Preambulo {
    String autor();
    int version() default 1;
    String fechaUltimaRevision();
    String[] revisores();
}
```

```
package anotaciones;
import java.lang.annotation.Documented;
@Documented
public @interface InProgress { }
```

```
import java.util.EnumMap;
import anotaciones.InProgress;
import anotaciones.Preambulo;
@InProgress
@Preambulo (autor = "Claudia",
fechaUltimaRevision = "6/10/2021",    revisores =
{ "Isa, Diego" })
public class TestEnumHash {
    //Código de la clase TestEnumHash
}
```

Las anotaciones pueden ser usadas para documentar.

Para que la información especificada en **@Preambulo** y **@InProgress** aparezca en la documentación generada por el **javadoc**, es preciso anotar la definición de estas anotaciones con la anotación **@Documented**

MODULE PACKAGE **CLASS** USE TREE INDEX HELP  
SUMMARY: NESTED | FIELD | CONSTR | METHOD    DETAIL: FIELD | CONSTR | METHOD

## Class TestEnumHash

java.lang.Object<sup>Ⓔ</sup>  
anotaciones.TestEnumHash

**@InProgress**  
**@Preambulo**(autor="Claudia",  
              fechaUltimaRevision="6/10/2021",  
              revisores={"Isa","Diego"})  
public class TestEnumHash  
extends Object<sup>Ⓔ</sup>

## Constructor Summary

### Constructors

Constructor	Description
TestEnumHash()	

## Method Summary

### Methods inherited from class java.lang.Object<sup>Ⓔ</sup>

equals<sup>Ⓔ</sup>, getClass<sup>Ⓔ</sup>, hashCode<sup>Ⓔ</sup>, notify<sup>Ⓔ</sup>, notifyAll<sup>Ⓔ</sup>, toString<sup>Ⓔ</sup>, wait<sup>Ⓔ</sup>, wait<sup>Ⓔ</sup>, wait<sup>Ⓔ</sup>

Ejecutamos el **javadoc**  
con **TestEnumHash.class**  
y obtenemos el siguiente  
archivo html

# Meta-Anotaciones: @Inherited

Por defecto las anotaciones no se heredan.

Si una anotación tiene la meta-anotación **@Inherited** entonces una clase anotada con dicha anotación causará que la anotación sea heredada por sus subclases.

```
@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface InProgress { }
```

```
@InProgress
@TODO("Calcula el interés mensual")
public class CuentaBase{
    public void calcularIntereses(float amount, float rate) {
        // Sin terminar}
    }
```

```
public class CajaDeAhorro extends CuentaBase{
    //TODO
}
```

Las anotaciones meta-anotadas con **@Inherited** son heredadas por subclases de la clase anotada. Las clases no heredan anotaciones de las interfaces que ellas implementan y los métodos no heredan anotaciones de los métodos que ellos sobrescriben

La anotación **@InProgress** se propaga en las subclases de **CuentaBase**.

# Los elementos de las Anotaciones

Los **tipos permitidos** para los elementos de las anotaciones son:

- Todos los tipos primitivos (int, long, byte, char, boolean, float, double)
- String
- Class
- Enumerativos
- Anotaciones
- Arreglos de cualquiera de los tipos mencionados

```
package anotaciones;  
import java.lang.annotation.*;  
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface SimulandoNull{  
    public int id ( ) default -1;  
    public String descripcion ( ) default "";  
}
```

El **compilador** aplica ciertas **restricciones** sobre los **valores** predeterminados de los elementos de las anotaciones:

- Ningún elemento puede no especificar valores: los elementos tienen valores predeterminados o valores provistos.
- Ninguno de los elementos (de tipo primitivo o no-primitivo) puede tomar el valor **null**.

Es importante tener esto en cuenta cuando escribimos un procesador de anotaciones y necesitamos **detectar la ausencia o presencia de un elemento**, dado que todos los elementos están presentes en una anotación.

Definir valores predeterminados como números negativos o strings vacíos nos permitirá **simular la ausencia de elementos**.



# Ejemplos de Anotaciones complejas

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)

public @interface ExceptionTest2 {
    Class<? extends Exception>[] value();
}
```

¿Cómo las uso?

```
@ExceptionTest({ IndexOutOfBoundsException.class, NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<String>();
    // El método de testeo podrá disparar alguna de estas 2 excepciones
    // IndexOutOfBoundsException or NullPointerException
    list.addAll(5, null);
}
```

# Ejemplos de Anotaciones más complejas

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface Revisiones {
    Revision[] value();
}
public @interface Revision {
    enum Concepto { EXCELENTE, SATISFACTORIO, INSATISFACTORIO };
    Concepto concepto();
    String revisor();
    String comentario() default "";
}
```

¿Cómo las uso?

```
@Revisiones(
    {@Revision(concepto=Revision.Concepto.EXCELENTE, revisor="df"),
     @Revision (concepto=Revision.Concepto.INSATISFACTORIO, revisor="eg",
                comentario="Este método necesita la anotación @Override ")
    }
)
```

# Generación de Archivos Externos

Las anotaciones son especialmente útiles cuando trabajamos con frameworks Java que requieren de cierta información adicional que acompaña al código fuente.

Tecnologías como **web services**, **librerías** de custom tags y herramientas **mapeadoras objeto/relacional** como **Hibernate** requieren de archivos descriptores XML que son externos al código Java. Trabajar con un archivo descriptor separado, requiere mantener 2 fuentes de información separadas sobre una clase y es frecuente que aparezcan problemas de sincronización entre ambas. El programador además de saber Java, debe saber cómo editar el archivo descriptor.

Consideremos el siguiente ejemplo: proveer un soporte básico de mapeo objeto-relacional para automatizar la creación de una tabla de la BD y guardarla en una clase Java. Usando anotaciones podemos mantener toda la información en el archivo fuente Java ->

**necesitamos anotaciones para definir el nombre de la tabla asociada con la clase, las columnas y los tipos SQL que mapean con las propiedades de la clase**



# **Java Networking**

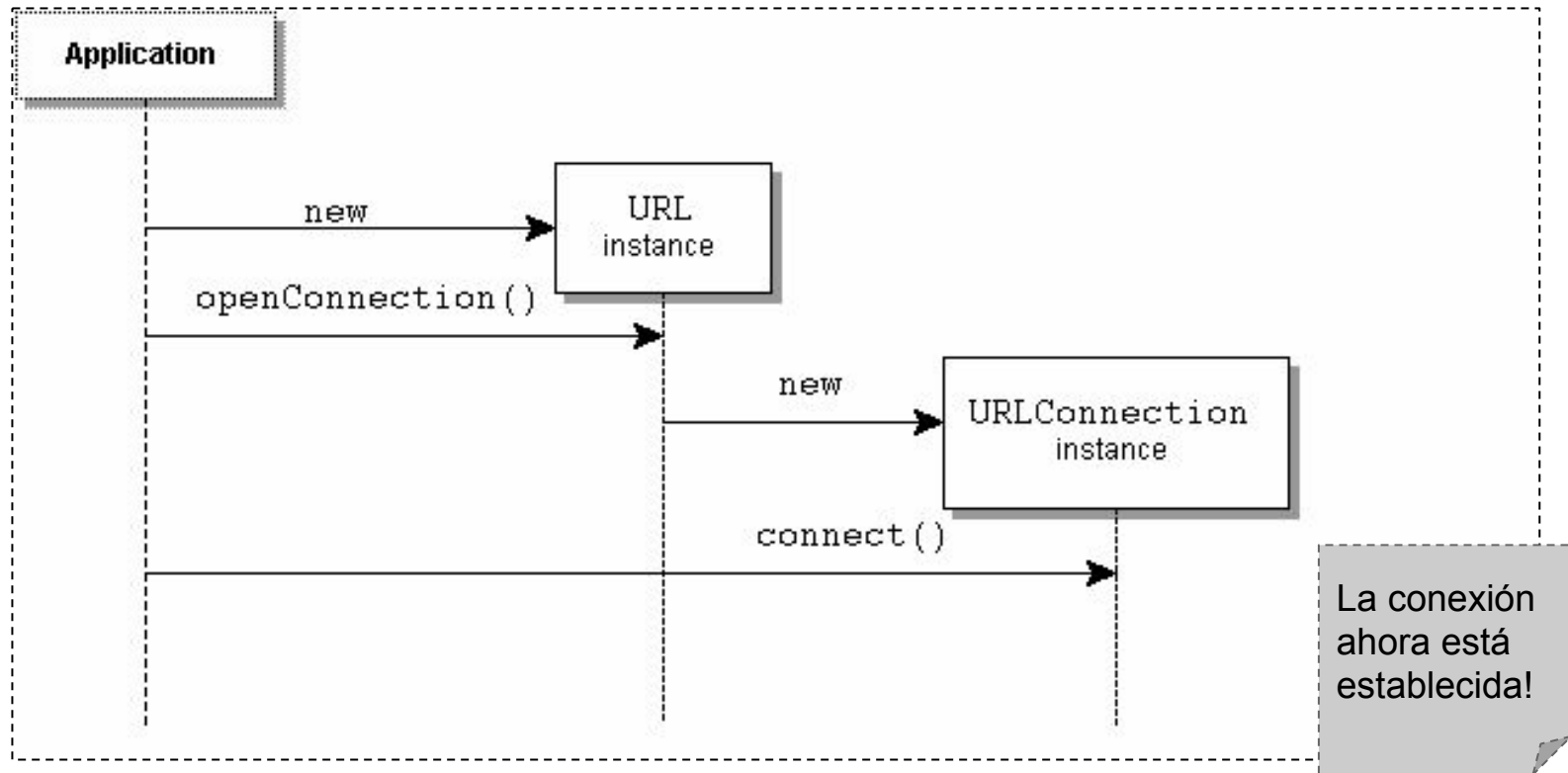
# Paquetes de la API

- java.net
- javax.net
- javax.net.ssl
- com.sun.net.httpserver
- com.sun.net.httpserver.spi

# Ubicación/Identificación de recursos de red

- URI
- **URL**
- URLClassLoader
- **URLConnection**
- URLStreamHandler
- **HttpURLConnection**
- JarURLConnection

# Ubicación/Identificación de recursos de red





# **Ejemplo de cliente HTTP**

# Uso de HttpURLConnection (redirect)

```
public class HttpRedirectExample {  
    public static void main(String[] args) {  
        try {  
            String url = "http://www.twitter.com";  
            URL obj = new URL(url);  
            HttpURLConnection conn = (HttpURLConnection) obj.openConnection();  
            conn.setReadTimeout(5000);  
            conn.addRequestProperty("Accept-Language", "en-US,en;q=0.8");  
            conn.addRequestProperty("User-Agent", "Mozilla");  
            conn.addRequestProperty("Referer", "google.com");  
  
            System.out.println("Request URL ... " + url);  
        }  
    }  
}
```

# Uso de HttpURLConnection (redirect)

```
boolean redirect = false;
// 3xx es redirect
int status = conn.getResponseCode();
if (status != HttpURLConnection.HTTP_OK) {
    if (status == HttpURLConnection.HTTP_MOVED_TEMP
        || status == HttpURLConnection.HTTP_MOVED_PERM
        || status == HttpURLConnection.HTTP_SEE_OTHER)
        redirect = true;
}

System.out.println("Response Code ... " + status);
```

# Uso de HttpURLConnection (redirect)

```
if (redirect) {  
  
    // se quiere redireccionar al header field "location"  
    String newUrl = conn.getHeaderField("Location");  
  
    // guardar las cookies y volver a enviarlas ..por las dudas se necesiten  
    String cookies = conn.getHeaderField("Set-Cookie");  
  
    // abrir una nueva conexión  
    conn = (HttpURLConnection) new URL(newUrl).openConnection();  
    conn.setRequestProperty("Cookie", cookies);  
    conn.addRequestProperty("Accept-Language", "en-US,en;q=0.8");  
    conn.addRequestProperty("User-Agent", "Mozilla");  
    conn.addRequestProperty("Referer", "google.com");  
  
    System.out.println("Redirect to URL : " + newUrl);  
  
}
```

# Uso de HttpURLConnection (redirect)

```
BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStream()));
String inputLine;
StringBuffer html = new StringBuffer();

while ((inputLine = in.readLine()) != null) {
    html.append(inputLine);
}
in.close();

System.out.println("Contenido URL... \n" + html.toString());
System.out.println("Hecho");

} catch (Exception e) {
    e.printStackTrace();
}

}
}
```

# **Ejemplo de servidor HTTP**

# paquete com.sun.net.httpserver

## clase **HttpServer**

```
HttpServer server = HttpServer.create(new InetSocketAddress(8000),0);
```

```
server.createContext("/applications/myapp", new MyHandler());
```

```
server.setExecutor(null);
```

```
server.start();
```

crea un default  
executor, que  
toma de a 1 los  
requerimientos

se pueden crear  
varios context cada  
uno administrado por  
un handler distinto

contiene el  
código de  
atención del  
requerimiento



# paquete com.sun.net.httpserver

## interfaz **HttpHandler**

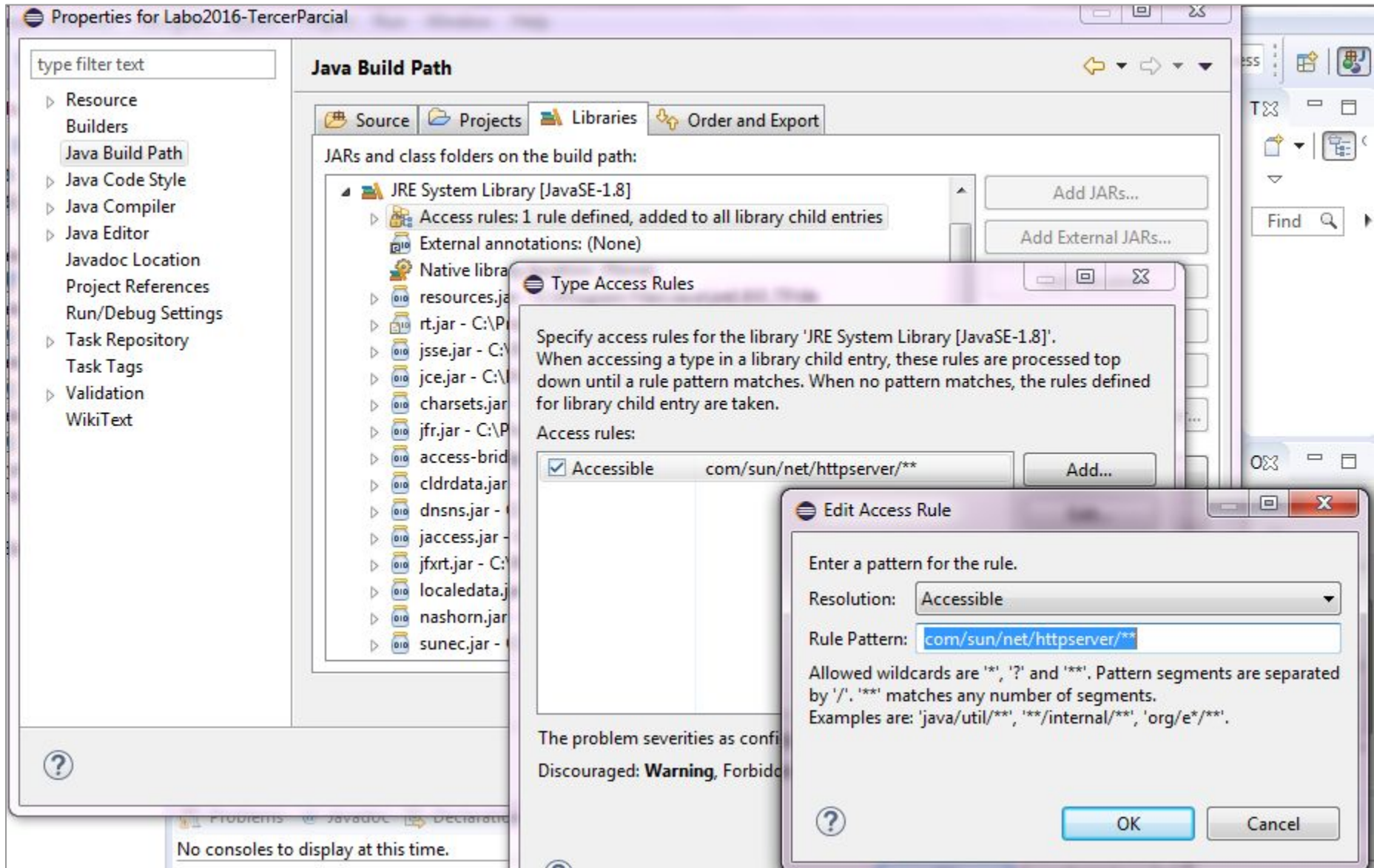
encapsula el  
requerimiento y la  
respuesta HTTP

```
class MyHandler implements HttpHandler {
```

```
    public void handle(HttpExchange t) throws IOException {  
        InputStream is = t.getRequestBody();  
        is.read(); // .. lee el request body  
        String response = "Esta es la respuesta";  
        t.sendResponseHeaders(200, response.length());  
        OutputStream os = t.getResponseBody();  
        os.write(response.getBytes());  
        os.close();  
    }
```

```
}
```

# Restricciones de Acceso



# Excepciones

# Excepciones

- Una **excepción** es un evento o problema que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de ejecución de instrucciones. Una **excepción** interrumpe el procesamiento normal porque no cuenta con la información necesaria para resolver el problema en el contexto en que sucedió. Todo lo que se puede hacer es abandonar dicho contexto y pasar el problema a un contexto de más alto nivel.
- **Java** usa excepciones para proveer de **manejo de errores** a sus programas. Ej.: acceso a posiciones inválidas de un arreglo, falta de memoria en el sistema, abrir un archivo inexistente en el *file system*, ejecutar una *query* sobre una tabla inexistente de una bd, hacer un *casting* a un tipo de dato inapropiado, etc.
- En Java cuando ocurre un error en un método, se llevan a cabo los siguientes pasos: 1) se crea un **objeto excepción** en la **heap** con el operador **new**, como cualquier otro objeto Java, 2) luego se **lanza la excepción**: se interrumpe la ejecución del método y el **objeto excepción** es expulsado del contexto actual. En este punto comienza a funcionar el **mecanismo de manejo de errores**: buscar un lugar apropiado donde continuar la ejecución del programa; el lugar apropiado es el **manejador de excepciones**, cuya función es recuperar el problema.

# Excepciones

En **Java** las excepciones se clasifican en:

- **Checked Exception o Verificables en Compilación:** representan un **problema con la posibilidad de recuperación**; las aplicaciones bien escritas podrían **anticipar y recuperar**; son errores que el compilador verifica que se contemplen y que pueden recuperarse. JAVA obliga a los métodos que disparan este tipo de excepciones a que **capturen y manejen el error o que las propaguen**. Por ejemplo al intentar abrir un archivo en el *file system* podría dispararse una excepción, dado que el archivo podría no existir, en ese caso una solución posible es pedirle al usuario que ingrese un nuevo nombre o propagar la excepción; otro error posible es intentar ejecutar una sentencia sql errónea.
- **Runtime Exception:** son **errores internos** de la aplicación que **no se pueden anticipar ni recuperar**. Estas excepciones en general son **bugs del programa** y se producen por **errores de lógica** o por el **mal uso de la API JAVA**. Por ejemplo las excepciones aritméticas (división por cero), excepciones por referencias nulas (acceso a un objeto mediante un puntero nulo), excepciones de indexación (acceso a un elemento de un arreglo con un índice muy chico ó demasiado grande) y error de *casting*. JAVA no obliga a que estas excepciones sean especificadas ni capturadas para su manejo. Conviene solucionar el error que produce el *bug*.
- **Error:** son **errores externos** a la aplicación, relacionadas al hardware, a la falta de memoria y que la aplicación no puede anticipar ni recuperar.

**NO Verificables en Compilación**

# Ejemplo

```
import java.io.*;
public class InputFile {
    private FileReader in;
                                throws FileNotFoundException
    public InputFile (String filename) {
        in=new FileReader(filename);
    }
                                throws IOException
    public String getWord() {
        int c;
        StringBuffer buf=new StringBuffer();
        do {
            c=in.read();
            if (Character.isWhitespace((char)c))
                return buf.toString();
            else
                buf.append((char)c);
        } while (c!=-1)
        return buf.toString();
    }
}
```

Si compilamos la clase **InputFile**, el compilador dispara mensajes de error similares a estos:

InputFile.java: 11: Warning: Exception **java.io.FileNotFoundException** must be caught, or it must be declared in throws clause of this method.

**in=new FileReader(filename);**

InputFile.java: 19: Warning: Exception **java.io.IOException** must be caught, or it must be declared in throws clause of this method.

**c=in.read();**

El compilador detecta que tanto el **constructor** de la clase **InputFile** como el **método *getWord()*** no **especifican ni capturan las excepciones** que se generan dentro de su alcance, por lo tanto la **compilación falla**.

# Ejemplo

<b>in=new FileReader(filename);</b>	<b>c=in.read();</b>
El nombre pasado como parámetro al constructor de la clase <b>FileReader</b> podría no existir en el <i>file system</i> , por tanto el constructor dispara la excepción: <b>java.io.FileNotFoundException.</b>	El método <b>getWord()</b> de la clase <b>InputFile</b> lee del objeto <b>FileReader</b> creado en el constructor de la clase usando el método <b>read()</b> . Este método dispara la excepción: <b>java.io.IOException</b> si por algún motivo no se puede leer.

- Al disparar estas excepciones, el **constructor** y el método **read()** de la **clase FileReader** permiten que los métodos que los invocan **capturen dicho error y lo recuperen** de una manera apropiada.
- La versión original de la clase **InputFile** **ignora** que el **constructor** y método **read()** de la clase **FileReader** disparan excepciones. Sin embargo **el compilador JAVA obliga a que toda excepción *checked* sea capturada o especificada**. Por lo tanto la **clase InputFile no compila**.  
En este punto tenemos dos opciones:
  - Ajustar el **constructor** de la clase **InputFile** y el método **getWord()** para que **capturen y recuperen** el error, o
  - **Ignorar los errores** y darle la oportunidad a los métodos que invoquen al **constructor** y al método **getWord()** de **InputFile** a que recuperen los errores usando la cláusula *throws*.



# Búsqueda del Manejador de Excepciones

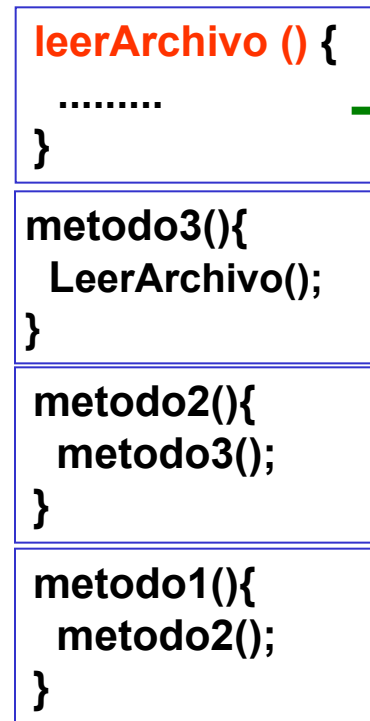
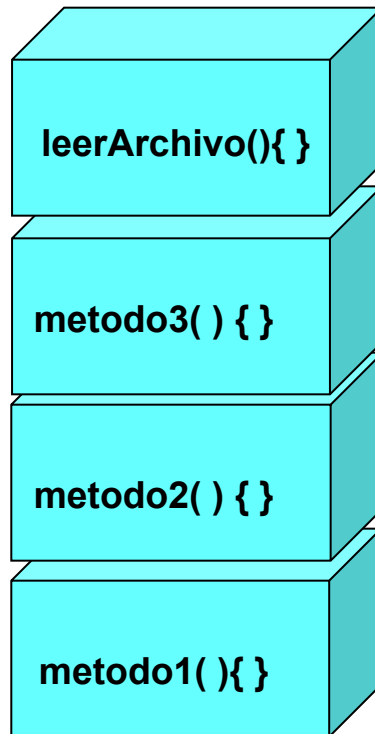
- Cuando un **método dispara una excepción** crea un objeto **Throwable** en la **heap** (la clase raíz de todas las excepciones), retorna dicho objeto y comienza a funcionar el **mecanismo de manejo de errores**. El sistema de ejecución de JAVA comienza a buscar en la **pila de ejecución de métodos invocados** aquel que contenga un **manejador de excepciones adecuado para dicho error**.
- Un **manejador de excepción es adecuado** si el tipo de la excepción disparada coincide con la manejada.

## Pila de ejecución

### leerArchivo()

- dispara una excepción:
- crea un objeto Exception en la Heap;
- interrumpe su flujo normal de ejecución y,
- el objeto excepción es lanzado del contexto actual y entregado a la VM.

El sistema de ejecución de Java comienza a buscar dónde continuar la ejecución: busca un **manejador de la excepción** apropiado en la pila de ejecución para recuperar el problema, comenzando por el método **leerArchivo()**. Podría retornar a un punto de la pila de ejecución bastante lejos del lugar dónde se produjo el error.



→ **dispara una excepción**

- Si el sistema de ejecución no encuentra un manejador apropiado en la pila de ejecución, la excepción será atendida por un manejador de *default* que finaliza la ejecución del programa.

# Excepciones: Separar el Código

Las excepciones permiten **SEPARAR** el **código regular** del programa del **código que maneja errores**.

En JAVA es posible escribir el flujo principal del código y tratar los casos excepcionales en otro lugar.

Cada cláusula **catch** es un **manejador de excepciones**, es similar a un método con un argumento de un tipo particular. Los identificadores **e1**, **e2**, **e3**, pueden usarse dentro del bloque de código del manejador, de la misma manera que los argumentos dentro del cuerpo de un método.

```
leerArchivo(){  
    try {  
        abrir archivo;  
        leer archivo;  
        cerrar archivo;  
    } catch (FileNotFoundException e1) {  
        hacerAlgo1(); El archivo no se puede abrir  
    } catch (ReadFailedException e2) {  
        hacerAlgo2(); Falla la lectura  
    } catch (FileCloseFailedException e3) {  
        hacerAlgo3(); El archivo no se puede cerrar  
    }  
    continuar();  
}
```

**Flujo normal: permite concentrarse en el problema que se está resolviendo**

**Manejo de Excepciones: permite tratar los errores del código precedente**

# Propagar y Capturar Errores

```
metodo1() {
```

```
  try {
```

```
    metodo2();
```

```
  } catch (Exception e) {
```

```
    procesarError();
```

```
  }
```

```
}
```

Flujo Normal

**metodo1()** es el único método interesado en recuperar el error que podría ocurrir en **leerArchivo()**. **Captura la excepción.**

**metodo2()** y **metodo3()** propagan la excepción que podría ocurrir en **leerArchivo()**. Se especifica en la cláusula **throws** del método.

```
metodo2() throws Exception {
```

```
  metodo3();
```

```
  //código
```

```
  JAVA
```

```
}
```

Si **metodo3()** dispara una excepción, se interrumpe la ejecución de **metodo2()** y **se propaga el error ocurrido.**

```
metodo3() throws Exception {
```

```
  leerArchivo();
```

```
  //código JAVA
```

```
}
```

Si **leerArchivo()** dispara una excepción, se interrumpe el flujo normal de ejecución de **metodo3()** y **se propaga el error.**

```
leerArchivo() throws Exception {
```

```
  Punto de creación del ERROR!!!
```

```
  //código JAVA
```

```
}
```

Se crea un objeto excepción con información sobre el error ocurrido, se interrumpe la ejecución de **leerArchivo()** y **se lanza la excepción** en busca de un manejador de la excepción.

# Jerarquía de Clases de Excepciones

**Throwable** es la clase base de todos los errores y excepciones en JAVA. Solamente los objetos que son instancias de **Throwable** o de alguna de sus subclases pueden ser disparados por la JVM o por la sentencia **throw**. A su vez el tipo del argumento de la cláusula **catch** solamente puede ser **Throwable** o de alguna de sus subclases.

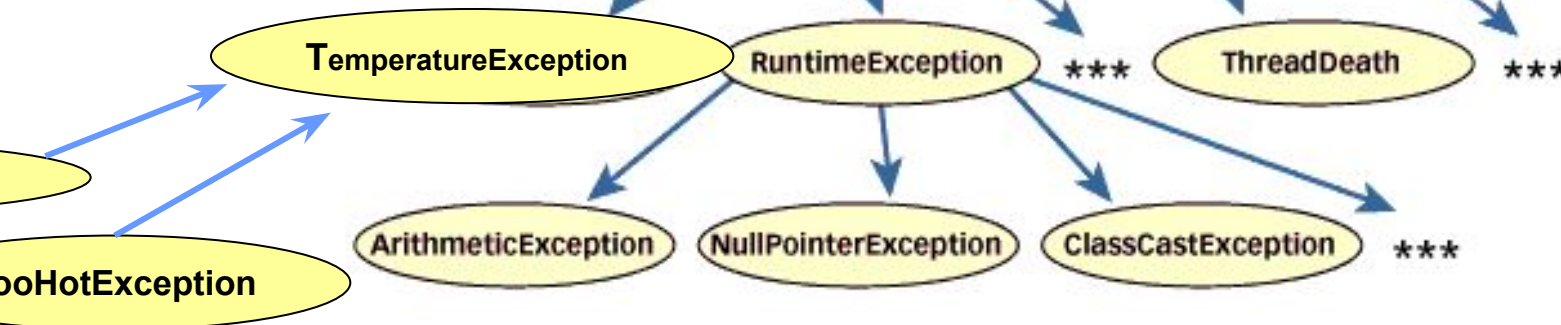
**Exception** es el tipo base de todos los objetos que pueden dispararse desde cualquier método de la API JAVA o desde nuestros propios métodos cuando ocurren condiciones anormales en la aplicación. En algunos casos pueden preverse y recuperarse mediante un código específico.

**Error** representa problemas serios, relacionados con la computadora, la memoria o el procesador. Los errores son disparados por la JVM y los programadores no pueden hacer nada.

**TemperatureException:** es una excepción *customizada* que representa los errores que ocurren cuando se toma una taza de café. Podrían desagregarse en: café muy frío o café muy caliente, etc.

Existen dos tipos de objetos **Throwable**: **Error** y **Exception**  
Ambas clases están en `java.lang`

*Excepciones customizadas*



*Excepciones específicas*

El nombre de la excepción representa el problema que ocurre y la idea es que sea lo más autoexplicativo posible. Existen clases de excepciones en diferentes paquetes: `java.util`, `java.net`, `java.io`, etc

# RuntimeException

- Representan **errores de lógica de programación** que el programador no puede anticipar, ni necesitan recuperarse ni identificarse. Ejemplos: excepciones aritméticas como división por cero; excepciones de punteros al intentar acceder a un objeto a través una referencia nula; excepciones de índices al intentar acceder a una posición fuera del rango de un arreglo; excepciones de *casting*, etc.
- Este tipo de excepciones son subclase de **RuntimeException**. Estas excepciones pueden ocurrir en cualquier lugar de un programa y típicamente podrían ser numerosas, es por ello que son ***no-verificables en compilación*** o ***unchecked exceptions***, el compilador no fuerza a especificarlas, no puede detectarlas estáticamente.
- Son disparadas automáticamente por la JVM. Por ejemplo: **NullPointerException, ClassCastException, ArrayIndexOutOfBoundsException, etc**
- Este tipo de excepciones ayudan al proceso de *debugging* del código. Los errores deben ser corregidos.

# Componentes de un Manejador de Excepciones

## El bloque try

```
try {  
    sentencias JAVA  
}
```

Las sentencias JAVA que pueden disparar excepciones deben estar encerradas dentro de un bloque **try**.

Es posible:

- a. Encerrar individualmente cada una de las sentencias JAVA que pueden disparar excepciones en un bloque *try* propio y proveer manejadores de excepciones individuales
- o
- b. Agrupar las sentencias que pueden disparar excepciones en un único bloque *try* y asociarle múltiples manejadores.

```
PrintWriter out=null;  
try{  
    out=new PrintWriter(new FileWriter("outFile.txt"));  
  
    for (int i=0; i< CANT; i++)  
        out.println("Valor en: "+ i +" = "+v.elementAt(i));  
}
```

El constructor de **FileWriter** dispara una **IOException** si no puede abrir el archivo

El método **elementAt()** de **vector** dispara una **ArrayIndexOutOfBoundsException** si el índice es muy chico (número negativo) ó muy grande

**RunTimeException**

# Componentes de un Manejador de Excepciones

## El bloque `catch` (opcional)

- Son los **manejadores de excepciones**.
- La forma de asociar manejadores de excepciones con un bloque ***try*** es proveyendo uno o más bloques ***catch*** inmediatamente después del bloque ***try***.

```
try{  
    //sentencias que pueden disparar excepciones  
} catch (SQLException e) {  
    System.err.println("Excepción capturada..." + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Excepción capturada..." + e.getMessage());  
}
```

Manejadores de  
Excepciones  
Especializados

- Si se dispara una excepción dentro del bloque ***try***, el mecanismo de **manejo de excepciones comienza a buscar el primer manejador de excepciones** con un argumento que coincida con el tipo de excepción disparada. La coincidencia entre el tipo de la excepción disparada y la de su manejador puede no ser exacta. El **tipo de las excepciones del manejador** puede ser cualquier **superclase** de la excepción disparada.
- Luego, se ejecuta el bloque ***catch*** y la excepción se considera **manejada/recuperada**. Solamente se ejecuta el bloque ***catch*** que coincide con la excepción disparada. Luego continúa la ejecución normal.
- Si adentro del bloque ***try*** la invocación a diferentes métodos dispara el mismo tipo de excepción, solamente necesitamos un único manejador de excepciones.

*¿Dónde continúa la ejecución?*



# Componentes de un Manejador de Excepciones

## El bloque catch (Continuación)

```
class Molestia extends Exception {}  
class Estornudo extends Molestia {}
```

```
public class SerHumano{  
    public static void main(String[] args) {  
        try {  
            throw new Estornudo();  
        } catch(Estornudo s) {  
            System.err.println("Manejador de Estornudo");  
        } catch(Molestia a) {  
            System.err.println("Manejador de Molestia");  
        }  
    }  
} // Fin de SerHumano
```

Podríamos eliminar el primer **catch** y dejar solamente el segundo:

```
catch(Molestia a) {  
    System.err.println("Manejador de Molestia");  
}
```

Captura las excepciones de tipo Molestia y todas las derivadas de Molestia

*¿Qué ocurre si se invierte el orden de los manejadores?*



# Componentes de un Manejador de Excepciones

## El bloque catch (Continuación)

```
catch (FileNotFoundException e) {  
    //código del manejador  
}
```

Manejador de excepciones específico

Capturar un error basado en su grupo ó tipo general especificando alguna de las superclases de excepciones:

```
catch (IOException e) {  
    //código del manejador  
}
```

Se puede averiguar la excepción específica usando el parámetro **e**

Se puede establecer un manejador muy general que capture cualquier tipo de excepciones. Debe ubicarse al final de la lista de manejadores:

```
catch (Exception e) {  
    //código del manejador  
}
```

Los manejadores de excepciones muy generales hacen el código propenso a errores pues capturan y manejan excepciones que no fueron previstas. No son útiles para recuperación de errores

# Componentes de un Manejador de Excepciones

## El bloque catch (Continuación)

La clase **Throwable** superclase de **Exception** provee un conjunto de métodos útiles para obtener información de la excepción disparada:

**String getMessage():** devuelve un mensaje detallado de la excepción.

**String getLocalizedMessage():** idem getMessage(), pero adaptado a la región.

**String toString():** devuelve una descripción corta del Throwable incluyendo el mensaje (si existe).

**void printStackTrace()**

**void printStackTrace(PrintStream)**

**void printStackTrace(java.io.PrintWriter)**

Imprimen el **Throwable** (error) y el *stack-trace* del **Throwable**. El *stack-trace* muestra la secuencia de métodos invocados que condujo al punto donde se disparó la excepción. La primera versión, imprime en la salida de error estándar y en la segunda y tercera es posible especificar dónde queremos imprimir.

# Ejemplo

```
public class TesteaExcepciones {
```

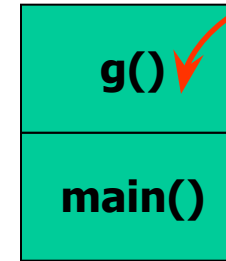
```
    public static void f() throws MiExcepcion {  
        System.err.println( "Origen de MiExcepcion desde f()" );  
        throw new MiExcepcion();  
    }
```

```
    public static void g() throws MiExcepcion {  
        System.err.println( "Origen de MiExcepcion desde g()" );  
        throw new MiExcepcion();  
    }
```

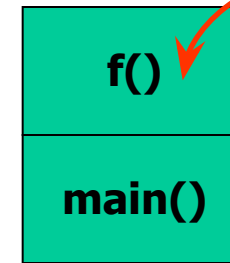
```
    public static void main(String[] args) {  
        try {  
            f();  
        } catch (MiExcepcion e) {  
            e.printStackTrace();  
        }  
  
        try {  
            g();  
        } catch (MiExcepcion e) {  
            e.printStackTrace();  
        }  
    }
```

```
}// Fin de la clase TesteaExcepciones
```

Dispara la MiExcepcion



Dispara la MiExcepcion



Origen de MiExcepcion desde f()

MiExcepcion

at TesteaExcepciones.f(TesteaExcepciones.java:6)

at TesteaExcepciones.main(TesteaExcepciones.java:14)

Origen de MiExcepcion desde g()

MiExcepcion

at TesteaExcepciones.g(TesteaExcepciones.java:10)

at TesteaExcepciones.main(TesteaExcepciones.java:20)

# Componentes de un Manejador de Excepciones

## El bloque finally

El último paso para definir un manejador de excepciones es liberar recursos antes que el control sea pasado a otra parte del programa. Esto se hace dentro del bloque **finally** escribiendo el código necesario para liberar recursos.

**El sistema de ejecución de JAVA siempre ejecuta las sentencias del bloque *finally* independientemente de lo que sucedió en el bloque *try*.**

### finally {

```
if (out != null){  
    System.out.println("Cerrando PrintWriter");  
    out.close();  
} else {  
    System.out.println("PrintWriter no fue abierto");  
}  
}
```

```
PrintWriter out=null;
```

### try{

```
out=new PrintWriter(new FileWriter("OutFile.txt"));  
for (int i=0; i< CANT; i++)  
    out.println("Valor en: "+ i + " = "+v.elementAt(i));  
}
```

Si no está el bloque **finally**: *¿Cómo se cierra el `PrintWriter` si no se provee un manejador de excepciones para `ArrayIndexOutOfBoundsException`?*

# Ejemplo Completo

```
public void writeList() {
    PrintWriter out=null;
    try {
        out=new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i=0; i< CANT; i++)
            out.println("Valor en: "+ i +" = "+ v.elementAt(i));
    } catch (IOException e) {
        System.err.println("Excepción capturada..." + e.getMessage());
    } finally {
        if (out !=null){
            System.out.println("Cerrando PrintWriter");
            out.close();
        } else {
            System.out.println("PrintWriter no fue abierto");
        }
    }
}
```

**v** es una variable de instancia privada de tipo **Vector**.

**CANT** es una constante de clase inicializada en 20

El bloque **try** tiene tres formas posibles de terminar:


- El **constructor de FileWriter** falla y dispara una **IOException** por ej. si el usr no tiene permiso de escritura, el disco está lleno, etc.
- La sentencia **v.elementAt(i)** falla y dispara una **ArrayIndexOutOfBoundsException**.
- No sucede ninguna falla y el bloque **try** termina exitosamente.

# Especificación de Excepciones

- JAVA fuerza a usar una sintaxis que permite informarle al programador las excepciones que podrían disparar los métodos que usa y de esta manera puede anticiparse a los errores y manejarlos. **Especificación de Excepciones**.
- La **Especificación de Excepciones** es parte de la **declaración del método** y se escribe después de la lista de argumentos. Es parte de la **interface pública del método**. Se usa la palabra clave **throws** seguida por una **lista de tipos de excepciones** que podrían dispararse en el alcance de dicho método.
- Si un método **NO captura** ni maneja las **excepciones checked** disparadas dentro de su alcance, el compilador JAVA fuerza al método a especificarlas en su declaración, **propagarlas**.
- En algunas situaciones es mejor que un método propague las excepciones, por ejemplo si se está implementando una librería, es posible que no se puedan prever las necesidades de todos los usuarios de la librería. En este caso es mejor no capturar las excepciones y permitirle a los métodos que usan las clases que manejen las excepciones que podrían dispararse.

```
public void writeList() throws IOException {  
    PrintWriter out=null;  
    out=new PrintWriter(new FileWriter("OutFile.txt"));  
  
    for (int i=0; i<CANT; i++)  
        out.println("Valor en: "+ i + " = "+v.elementAt(i));  
    out.close();  
}
```

No es necesario tratar ni propagar (especificar) la excepción **ArrayIndexOutOfBoundsException** porque es de tipo **RuntimeException**



# El bloque finally

```
public void writeList() throws IOException {  
    PrintWriter out=null;  
    try{  
        out=new PrintWriter(new FileWriter("OutFile.txt"));  
  
        for (int i=0; i<CANT; i++)  
            out.println("Valor en: "+ i +" = "+v.elementAt(i));  
    }  
    finally{  
        if (out !=null){  
            System.out.println("Cerrando PrintWriter");  
            out.close();  
        } else {  
            System.out.println("PrintWriter no fue abierto");  
        }  
    }  
}
```

- El bloque **finally** debe tener un bloque **try**, el **catch** es opcional.
- Lo relevante del bloque **finally** es liberar los recursos que podrían haberse alocado en el bloque **try** independientemente de si se disparó o no una excepción.

# ¿Cómo disparar Excepciones ?

La palabra clave **throw** es usada por todos los métodos que crean objetos `Exception` y requiere como único argumento un objeto **Throwable**.

El método `pop()` usa la cláusula **throws** para declarar que dentro de su alcance se puede disparar una **EmptyStackException**

```
public Object pop() throws EmptyStackException {  
    Object obj;  
    if (size == 0)  
        throw new EmptyStackException();  
    obj = objectAt(size - 1);  
    setObjectAt(size - 1, null);  
    size--;  
    return obj;  
}
```

- Se crea un objeto en la *heap* que representa el error y la referencia la tiene la cláusula **throw**.
- El objeto **EmptyStackException** es retornado por el método `pop()`.

El método `pop()` chequea si hay algún elemento en la pila. Si está vacía, instancia un objeto **EmptyStackException** y lo lanza. Algún manejador en un contexto superior manejará el error.



# Re-disparar una Excepción

```
try{  
    // código del bloque try  
} catch (FileNotFoundException e) {  
    System.out.println("Excepción de archivo no encontrado");  
    logger.log(e);  
    throw e; ← Re-dispara la excepción capturada  
} catch (IOException e) {  
    //código del manejador de excepciones  
}
```

- Re-disparar una excepción causa que la excepción busque un manejador de excepciones en un contexto de más alto nivel. En este caso, el bloque catch maneja parcialmente la excepción ocurrida y la re-lanza para que un manejador de más alto nivel finalice su manejo.
- Las cláusulas **catch** del mismo bloque **try** son ignoradas.
- Se conserva todo acerca del objeto excepción, de manera que el manejador del contexto de más alto nivel que capture la excepción, pueda extraer información.
- Si la **excepción que se re-dispara es la actual** (como en el ejemplo), la información que se imprime con el método **printStackTrace()** pertenece al origen de la excepción, no al lugar dónde se re-disparó.
- **Es posible re-disparar una excepción diferente a la capturada:** la información del lugar de origen de la excepción se pierde y se tiene la información perteneciente al nuevo **throw**.

# Re-disparar una Excepción

```
try{  
    // código del bloque try  
} catch (IOException ioe)  
{  
    throw new ReportCreationException(ioe);  
}
```

- **Traslación de excepciones:** consiste en wrappear **excepciones de bajo nivel en excepciones de alto nivel**. Es decir, explicar excepciones (errores) en términos de abstracciones de más alto nivel.
- Se produce un error, sin embargo el verdadero **manejo ocurre en un nivel superior**. En el lugar donde se produce el error se añade información útil que el manejador de nivel superior no conoce, en este caso la verdadera causa del error, que puede recuperarse con el método **getCause()**.

Un método dispara una **IOException** como resultado de intentar **crear un reporte**. La firma del método establece que se dispara un **ReportCreationException**. La instancia de **IOException** está wrapeada (abstraída) en una **ReportCreationException**.

# Re-disparar la misma Excepción

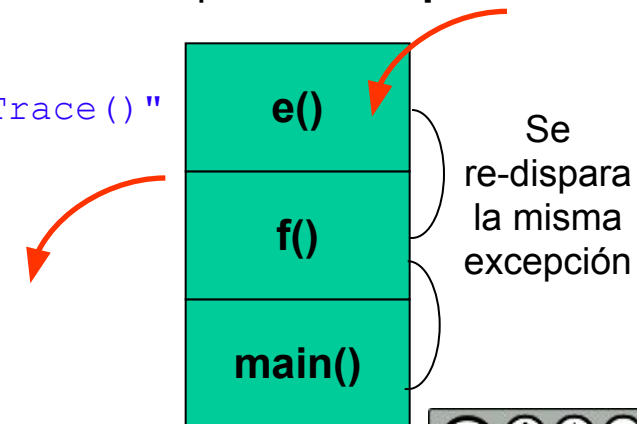
```
public class ReDisparar{
    public static void e() throws Exception{
        System.err.println( "Origen de la excepción en e()" );
        throw new Exception( "disparada en e()" );
    }
    public static void f() throws Exception{
        try {
            e();
        } catch (Exception e) {
            System.err.println( "Adentro de f(),
                                e.printStackTrace()" );

            e.printStackTrace();
            throw e;
        }
    }
    public static void main(String[] args) {
        try {
            f();
        } catch (Exception e) {
            System.err.println( "Capturada en el main(), e.printStackTrace()" );
            e.printStackTrace();
        }
    }
} // Fin de ReDisparar
```

Origen de la excepción en e()  
Adentro de f(), e.printStackTrace()  
java.lang.Exception: disparada en e()  
at ReDisparar.e(ReDisparar.java:18)  
at ReDisparar.f(ReDisparar.java:22)  
at ReDisparar.main(ReDisparar.java:31)

Capturada en el main(), e.printStackTrace()  
java.lang.Exception: disparada en e()  
at ReDisparar.e(ReDisparar.java:18)  
at ReDisparar.f(ReDisparar.java:22)  
at ReDisparar.main(ReDisparar.java:31)

Se dispara la excepción, **Exception**



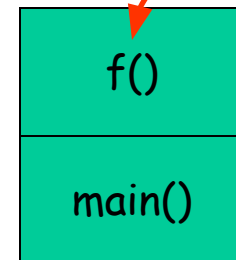
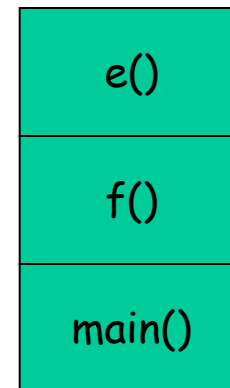
# Re-disparar otra Excepción

```
public class ReDisparar{
    public static void e() throws Exception{
        System.err.println( "Origen de la excepción en e()" );
        throw new Exception("disparada en e()");
    }
    public static void f() throws Exception{
        try {
            e();
        } catch (Exception e) {
            System.err.println( "Adentro de f(),
            e.printStackTrace()" );
            e.printStackTrace();
            throw new MiExcepcion( "MiExcepcion()" );
        }
    }
    public static void main(String[] args) {
        try {
            f();
        } catch (Exception e) {
            System.err.println( "Capturada en el main(),
            e.printStackTrace()" );
            e.printStackTrace();
        }
    }
} // Fin de ReDisparar
```

Origen de la excepción en e()  
Adentro de f(), e.printStackTrace()  
java.lang.Exception: disparada en e()  
at ReDisparar.e(ReDisparar.java:18)  
at ReDisparar.f(ReDisparar.java:22)  
at ReDisparar.main(ReDisparar.java:32)  
Capturada en el main(), e.printStackTrace()  
MiExcepcion: MiExcepcion()  
at ReDisparar.f(ReDisparar.java:27)  
at ReDisparar.main(ReDisparar.java:32)

Se dispara la excepción, **Exception**

Se re-dispara **MiExcepcion**



**De la excepción MiExcepcion solamente sabe que se originó en f() y no en e()**

# Restricciones en Excepciones

## Sobreescritura de métodos

- Cuando se sobreescribe un método solamente se pueden disparar las excepciones especificadas en la versión de la clase base del método. La utilidad de esta restricción es que el código que funciona correctamente para un objeto de la clase base, seguirá funcionando para un objeto de la clase derivada (principio fundamental de la OO)
- La **interface de especificación de excepciones** de un método puede reducirse y sobreescribirse en la herencia, pero nunca ampliarse. Es exactamente opuesto a lo que ocurre en la herencia con los especificadores de acceso de una clase.

```
public class A{
    public void f() throws AException{
        //código de f()
    }
    public void g() throws BException, CException{
        //código de g()
    }
}
```

```
public class B extends A{
    public void f(){
        //código de f()
    }
    public void g() throws DException{
        //código de g()
    }
}
```

```
class AException extends Exception {}
class BException extends AException {}
class CException extends AException {}
class DException extends BException {}
```

```
public class Test{
    public static void main(String[] args){
        try {
            A x= new A();    A x=new B();
            x.f();
            x.g();
        } catch (BException e) {}
        catch (CException e) {}
        catch (AException e) {}
    }
}
```

*¿Está bien?*

# Restricciones en Excepciones

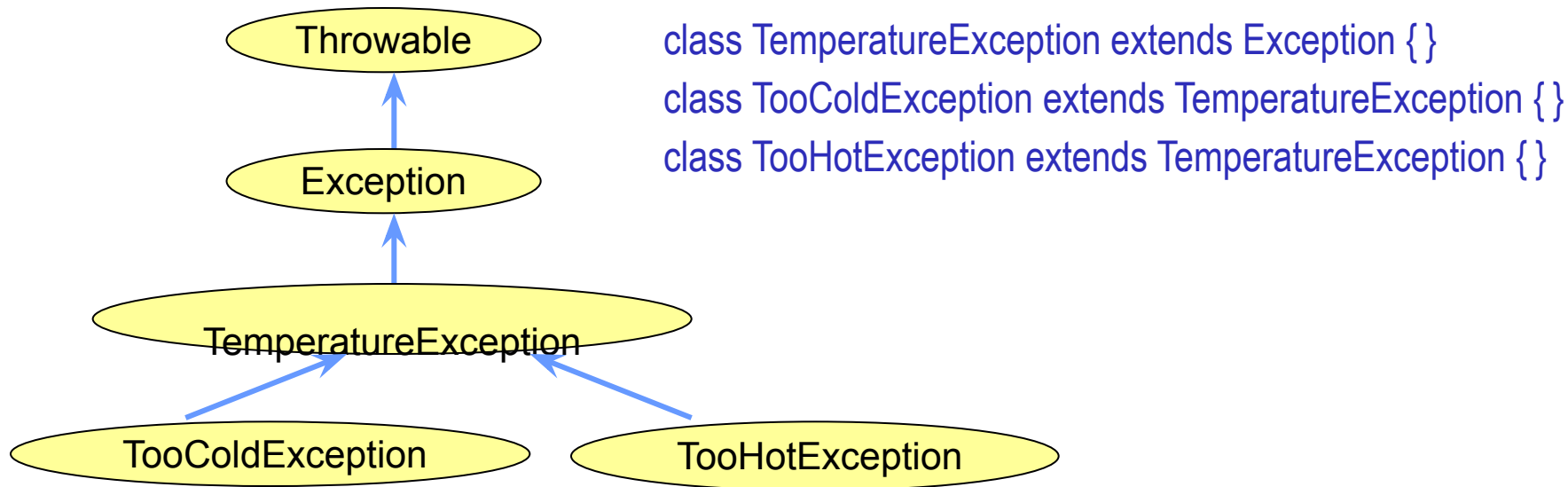
## Constructores

- Los constructores no se sobrescriben.
- Los constructores de una subclase pueden disparar excepciones diferentes a las excepciones disparadas por el constructor de la superclase.
- Hay que ser cuidadoso de dejar el objeto que se intenta construir y no se puede, en un estado seguro.

# Crear Excepciones Propias

## Ejemplo: Café Virtual

Cuando se diseña una librería las clase deben interactuar bien y sus interfaces deben ser fáciles de entender y de usar. Para ello es bueno diseñar clases de excepciones. Las condiciones excepcionales que pueden ocurrir cuando el cliente toma una taza de café en el café virtual son: el café está muy frío o muy caliente.



**Convención de nombres:** es una buena práctica agregar el texto Exception a todos los nombres de clases que heredan directa ó indirectamente de la clase `Exception`.

# Ejemplo: Café Virtual

```
class VirtualPerson {  
    private static final int tooCold = 65;  
    private static final int tooHot = 85;  
  
    public void drinkCoffee(CoffeeCup cup) throws  
        TooColdException, TooHotException {  
  
        int temperature = cup.getTemperature();  
  
        if (temperature <= tooCold) {  
            throw new TooColdException();  
        }  
  
        else if (temperature >= tooHot) {  
            throw new TooHotException();  
        }  
    }  
}
```

Se declaran las excepciones que puede disparar el método **drinkCoffee()**



Se crea un  
objeto  
excepción y  
se dispara

```
class CoffeeCup {  
    // 75 grados Celsius: es la temperatura ideal del café  
    private int temperature = 75;  
  
    public void setTemperature(int val){  
        temperature = val;  
    }  
  
    public int getTemperature() {  
        return temperature;  
    }  
}
```



# Ejemplo: Café Virtual

```
class VirtualCafe {  
    public static void serveCustomer(VirtualPerson cust, CoffeeCup cup) {  
        try {  
            cust.drinkCoffee(cup);  
            System.out.println("El Café está OK.");  
  
        } catch (TooColdException e) {  
            System.out.println("El Café está muy frío.");  
        } catch (TooHotException e) {  
            System.out.println("El Café está muy caliente.");  
        }  
    }  
}
```

El método drinkCoffee() puede disparar las excepciones: **TooHotException** ó **TooColdException**

```
catch (TemperatureException e) {  
    System.out.println("El Café no está OK");  
}
```

- Se recomienda el uso de manejadores de excepciones especializados.
- Los manejadores genéricos (que agrupan muchos tipos de excepciones) no son útiles para recuperación de errores, dado que el manejador tiene que determinar qué tipo de excepción ocurrió para elegir la mejor estrategia para recuperar el error.
- Los manejadores genéricos pueden hacer que el código sea más propenso a errores, dado que se capturan y manejan excepciones que pueden no haber sido previstas por el programador.

# Incorporar información a las Excepciones

- Las excepciones además de transferir el control desde una parte del programa a otra, permiten transferir información.
- Es posible agregar información a un objeto excepción acerca de la condición anormal que se produjo
- La cláusula *catch* permite obtener información interrogando directamente al objeto excepción.
- La clase *Exception* permite especificar mensajes de tipo String a un objeto excepción y, recuperarlos vía el método ***getMessage()*** (sobre el objeto excepción).
- Es posible agregar a un objeto *Exception* información de un tipo distinto que String. Para ello, es necesario agregar a la subclase de *Exception* datos y métodos de acceso a los mismos.

# Incorporar información a las Excepciones

```
class UnusualTasteException extends Exception {  
    UnusualTasteException() {}  
    UnusualTasteException(String msg) {  
        super(msg);  
    }  
}
```


Dos constructores para  
**UnusualTasteException**

Un programa que dispara una excepción de tipo *UnusualTasteException* puede hacerlo de las dos formas siguiente:

- a) **throw new UnusualTasteException()**
- b) **throw new UnusualTasteException("El Café parece Té")**

```
try {  
    //código JAVA que dispara excepciones  
} catch (UnusualTasteException e) {  
    String s = e.getMessage();  
    System.out.println(s);  
}
```

Se obtiene  
información del  
objeto excepción



# Incorporar información a las Excepciones

```
abstract class TemperatureException extends Exception {  
    private int temperature;  
    public TemperatureException(int temperature) {  
        this.temperature = temperature;  
    }  
    public int getTemperature() {  
        return temperature;  
    }  
}
```

Datos y métodos de acceso a la información asociada a la excepción

```
class TooColdException extends TemperatureException {  
    public TooColdException(int temperature) {  
        super(temperature);  
    }  
}
```

```
class TooHotException extends TemperatureException {  
    public TooHotException(int temperature) {  
        super(temperature);  
    }  
}
```

# Ejemplo - Café Virtual

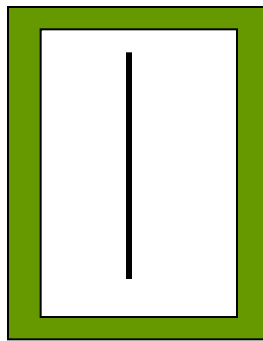
```
class VirtualPerson {  
  
    private static final int tooCold = 65;  
    private static final int tooHot = 85;  
  
    public void drinkCoffee(CoffeeCup cup) throws  
        TooColdException, TooHotException {  
  
        int temperature = cup.getTemperature();  
        if (temperature <= tooCold) {  
            throw new TooColdException(temperature);  
        }  
        else if (temperature >= tooHot) {  
            throw new TooHotException(temperature);  
        }  
        //...  
    }  
    //...  
}
```

```
class VirtualCafe {  
  
    public static void serveCustomer(VirtualPerson cust,  
        CoffeeCup cup) {  
        try {  
            cust.drinkCoffee(cup);  
        } catch (TooColdException e) {  
            int temp = e.getTemperature();  
            if (temp > 55 && temp <= 65) {  
            } else if (temp > 0 && temp <= 55) {  
            } else if (temp <= 0) {  
                //código JAVA  
            }  
        } catch (TooHotException e) {  
            int temp = e.getTemperature();  
            if (temp >= 85 && temp < 100) {  
            } else if (temp >= 100 && temp < 2000) {  
            } else if (temp >= 2000) {  
            }  
        }  
    }  
}
```

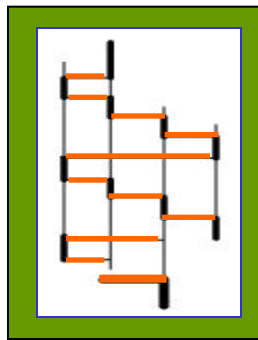
# Concurrencia Threads

# Threads

- Un **thread** es un flujo de control secuencial dentro de un proceso. A los threads también se los conoce como **procesos livianos** (requiere menos recursos crear un thread nuevo que un proceso nuevo) ó **contextos de ejecución**.
- Un **thread** es similar a un programa secuencial: tiene un comienzo, una secuencia de ejecución, un final y en un instante de tiempo dado hay un único punto de ejecución. Sin embargo, un thread no es un programa. Un **thread** se ejecuta adentro de un programa.
- Lo novedoso en **threads** es el uso de múltiples threads adentro de un mismo programa, ejecutándose simultáneamente y realizando tareas diferentes:



Programa *singleThread*



Programa *multiThread*

Thread en ejecución  
Transferencia del control  
Thread *bloqueado*



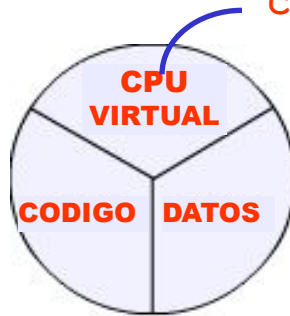
En un programa *multithread*, más de un thread se ejecuta en forma concurrente. El control de ejecución es transferido entre los diferentes threads, cada uno de los cuáles es responsable de distintas tareas.

- En el modelo de **multithreading** la CPU asigna a cada thread un tiempo para que se ejecute; cada thread “tiene la percepción” que dispone de la CPU constantemente, sin embargo el tiempo de CPU está dividido entre todos los threads.

# Threads

- Un **thread** se ejecuta dentro del contexto de un programa o proceso y comparte los recursos asignados al programa. A pesar de esto, los **threads** toman algunos recursos del ambiente de ejecución del programa como propios: tienen su propia pila de ejecución, contador de programa, código y datos. Como un **thread** solamente se ejecuta dentro de un contexto, a un thread también se lo llama **contexto de ejecución**.
- La plataforma JAVA soporta programas **multithreading** a través del lenguaje, de librerías y del sistema de ejecución. A partir de la versión 5.0, la plataforma JAVA incluye librerías de concurrencia de más alto nivel

Múltiples-threads comparten el mismo código, cuando se ejecutan a partir de instancias de la misma clase.



CPU virtual que ejecuta código y utiliza datos

Múltiples-threads comparten datos, cuando acceden a objetos comunes (podría ser a partir de códigos diferentes).

- La **clase Thread** forma parte del paquete **java.lang** y provee una implementación de threads independiente del sistema de ejecución. Hay dos estrategias para usar objetos **Threads**:
  - **Directamente controlar la creación y el gerenciamiento** instanciando un Thread cada vez que la aplicación requiere iniciar una tarea concurrente.
  - **Abstraer el gerenciamiento** de threads pasando la tarea concurrente a un **ejecutor** para que la administre y ejecute.



# Creación y Gerenciamiento de Threads

- La **clase Thread** provee el comportamiento genérico de los threads JAVA: arranque, ejecución, interrupción, asignación de prioridades, etc.
- El método **run()** es el más importante de la clase **Thread**, implementa la funcionalidad del thread, es el código que se ejecutará **simultáneamente** con otros threads del programa. El método **run()** predeterminado provisto por la clase Thread no hace nada.
- La plataforma JAVA es **multithread**: siempre hay un thread ejecutándose junto con las aplicaciones de los usuarios, por ejemplo el **garbage collector** es un thread que se ejecuta en background; las GUI's **dibujan las componentes** en la pantallas y **recolectan los eventos** generados por el usuario en threads separados, etc.
- Una aplicación JAVA siempre se ejecuta en un **thread**, llamado **main thread**. Este **thread** ejecuta secuencialmente las sentencias del cuerpo del método main() de la clase. En otros programas JAVA como **servlets**, que no tienen método main(), la ejecución del **main thread** comienza con el método main() de su contenedor, que es el encargado de invocar los métodos del ciclo de vida de dichas componentes.

# El método run() de la clase Thread

Es un método estándar de la clase **Thread**. Es el lugar donde el **thread** comienza su ejecución.

```
public class SimpleThread extends Thread {
    private int contador=10;
    public SimpleThread(int nro) {
        super("" +nro); — Invoca al constructor de Thread con
                        un argumento String que es el nombre
                        del thread
    }
    public String toString() {
        return "#" + getName() + ":"
+contador--; Se recupera el nombre del
                    thread con el método
                    getName() de la clase Thread
    }
    public void run() {
        for (int i = contador; i > 0; i --)
            System.out.println(this);

        System.out.println("Termino!" + this );
        Un thread termina cuando finaliza
                    el método run()
    }
}
```

```
public class TestCincoThread {
    private static int nroThread=0;
    public static void main(String[] args) {
        for (int i=0; i<5;i++)
            new SimpleThread(++ nroThread).start();
    }
}
```

Inicializa el objeto Thread e invoca al método run(). El thread pasa a estado "vivo"

Cuando el método start() retorna, hay 2 threads ejecutándose en paralelo: el thread que invocó al start(), en nuestro caso el main thread y el thread que está ejecutando el método run().

Tenemos 5 tareas concurrentes, cada una de ellas imprime en pantalla 10 veces su nombre. Además tenemos el **main thread**.

## ¿Cuál es la salida del programa TestCincoThread?

La salida de una ejecución del programa podría ser diferente a la salida de otra ejecución del mismo programa, dado que el mecanismo de *scheduling* de threads no es determinístico.

# Métodos de la clase Thread

## Sleep

**Suspende** temporariamente la ejecución del **thread** que se está ejecutando. Afecta solamente al **thread** que ejecuta el **sleep()**, no es posible decirle a otro thread que "se duerma". Es un método de clase. El tiempo de suspensión se expresa en milisegundos. El método **sleep()** está encerrado en un bloque

```
import java.util.concurrent.TimeUnit;
public class SimpleThread extends Thread {
    private int contador=10;
    public SimpleThread(int nro) {
        super("" +nro);
    }
    public String toString() {
        return "#" + getName()+ ":" +contador--;
    }
    public void run() {
        for (int i = contador; i > 0; i --){
            System.out.println(this);
            try {
                //Antes de JSE 5:
                //sleep(100);
                //Estilo JSE 5:
                TimeUnit.MILLISECONDS.sleep(100);
            } catch (InterruptedException e){
                throw new RuntimeException();
            }
            System.out.println("Termino!" + this );
        }
    }
}
```

try/catch, dado que el thread podría recibir una solicitud de interrupción antes que el tiempo se agote (se invoca al método **interrupt()** sobre el objeto thread).

- sleep()** es un método sobrecargado, que permite especificar el tiempo de espera en milisegundos y en nanosegundos. En la mayoría de las implementaciones de la JVM, este tiempo se redondea a la cantidad de milisegundos más próxima (en general un múltiplo de 20 milseg o 50 milseg).
- Los threads se ejecutan en cualquier orden. El método **sleep()** no permite controlar el orden de ejecución de los threads; suspende la ejecución del thread por un tiempo dado.
- En nuestro ejemplo, la única garantía que se tiene es que el thread suspenderá su ejecución por al menos 100 milisegundos, pero podría tomar más tiempo antes de retomar la ejecución.

## Join

# Métodos de la clase Thread

El método **join()** permite que un **thread** espere a que otro termine de ejecutarse. El objetivo del método **join()** es esperar por un evento específico: la terminación de un **thread**. El **thread** que invoca al **join()** sobre otro **thread** se bloquea hasta que dicho **thread** termine su método **run()**. Una vez que el **thread** completa el **run()**, el método **join()** retorna inmediatamente.

```
public class SimpleThreadTest2 {
    public static void main(String args[]) {
        SimpleThread t=new SimpleThread(1);
        t.start();
        while (t.isAlive()) {
            System.out.println("esperando...");
            try {
                t.join();
            } catch (InterruptedException e) {
                System.out.println(getName() + "join
interrumpido");
            }
            System.out.println(getName() + " join
completado");
        }
    } // fin del while
}
```

El main thread se suspende hasta que el thread t termine (isAlive() devuelve false)

```
thrd.start();
while (thrd.isAlive()){
    try{
        thrd.join(2000);
        System.out.print(".");
    } catch (InterruptedException e) { }
} System.out.println(" Listo!");
```

Este segmento de código inicia al thread thrd y cada 2 segundos imprime un "." mientras thrd continúa ejecutándose

- En este caso el **main thread** se bloquea en espera que el **thread t** termine de ejecutarse.
- El método **join()** es sobrecargado, permite especificar el tiempo de espera. Sin embargo, de la misma manera que el **sleep()**, no se puede asumir que este tiempo sea preciso. Como el método **sleep()**, el **join()** responde a una interrupción terminando con una **InterruptedException**

# Métodos de la clase Thread

## Yield

Permite indicarle al mecanismo de scheduling que el thread ya hizo suficiente trabajo y que podría cederle tiempo de CPU a otro thread. Su efecto es dependiente del SO sobre el que se ejecuta la JVM. Permite implementar **multithreading cooperativo**. Es una pista para el scheduling.

```
public class SimpleThread extends Thread {
    private int contador=10;
    public SimpleThread(int nro) {
        super("" +nro);
    }
    public String toString() {
        return "#" + getName()+ ":" +contador--;
    }
    public void run() {
        for (int i = contador; i > 0; i --){
            System.out.println(this);
            Thread.yield();
        }
        System.out.println("Termino!" + this );
    }
}
```

SimpleThread de esta manera realizaría un procesamiento mejor distribuido entre varias tareas SimpleThread.

# La interface Runnable

- Es posible escribir **threads** implementando la interface **Runnable**.
- La interface **Runnable** solamente especifica que se debe implementar el método **run()**.

```
package java.lang;

public interface Runnable {
    public void run();
}
```

```
package java.lang;

public class Thread implements Runnable {
    //código JAVA
}
```

- Si una clase implementa la **interface Runnable** simplemente significa que tiene un método **run()**, pero NO tiene ninguna habilidad de **threading**. Para crear un thread a partir de un objeto **Runnable** es necesario crear un **objeto Thread** y pasarle el objeto **Runnable** en el constructor. Luego, se invoca al método **start()** sobre el **thread** creado, NO sobre el objeto **Runnable**.

Identifica el thread ejecutándose

```
public class SimpleThread implements Runnable{
    private int contador=10;
    public String toString() {
        return "#" + Thread.currentThread().getName() + ":" + contador--;
    }
    public void run(){
        for (int i = contador; i > 0; i --)
            System.out.println(this);
        System.out.println("Termino!" + this );
    }
}
```

Se obtiene una  
referencia al thread en  
ejecución

```
public class TestCincoThread {
    private static int nroThread=0;
    public static void main(String[] args) {
        for (int i=0; i<5;i++)
            new Thread(new SimpleThread(), ""+i).start();
    }
}
```

# Métodos de la clase Thread

## Interrupt

Es un **pedido de interrupción**. El thread que lo recibe se interrumpe a si mismo de una manera conveniente. El pedido causa que los métodos de bloqueo (**sleep()**, **join()**, **wait()**) disparen la excepción `InterruptedException` y además setea un *flag* en el **thread** que indica que al thread se le ha pedido que se interrumpa. Se usa el método **isInterrupted()** para preguntar por este *flag*.

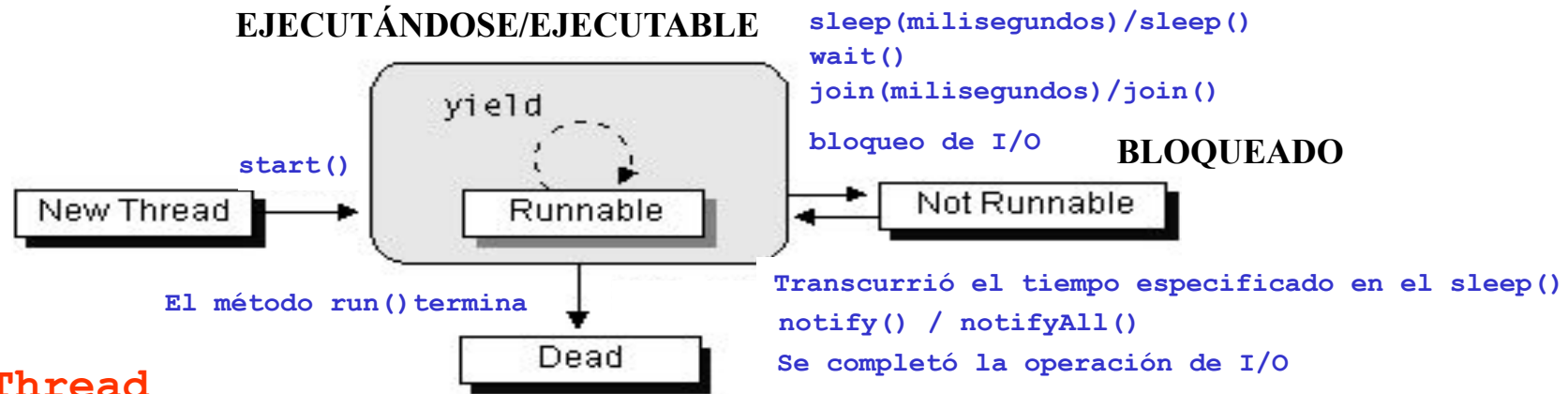
```
public class SimpleThreadInterrupt {
    private int contador = 10;
    private class Mensaje implements Runnable {
        public void run() {
            for (int i = contador; i > 0; i--) {
                System.out.println( i);
                try {
                    TimeUnit.MILLISECONDS.sleep(4000);
                } catch (InterruptedException e) {
                    System.out.println("El thread " + this + " no puede terminar");
                }
            }
            System.out.println("Termino!" + this);
        }
    }
    // Fin de la clase Mensaje
    public static void main(String[] args) throws InterruptedException {
        SimpleThreadInterrupt s=new SimpleThreadInterrupt();
        Thread t = new Thread(s.new Mensaje());
        t.start();
        while (t.isAlive()) {
            System.out.println("Esperando.....");
            t.join(1000);
            if (((System.currentTimeMillis()) > 1000 * 60 * 60) && t.isAlive()) {
                System.out.println("Cansado de esperar!");
                t.interrupt();
                t.join();
            }
        }
        System.out.println("Fin!");
    }
    // Fin de la clase SimpleThreadInterrupt
```

# La interface Runnable

- Una ventaja de implementar la **interface Runnable** es que todo el código pertenece a la misma clase y de esta manera es simple combinar la clase base con otras interfaces. Es posible acceder a cualquier objeto y métodos de la clase evitándose mantener referencias en objeto separados.
- Los objetos **Runnable** pueden extender a otras clases en lugar de Thread.
- La **interface Runnable** permite **separar** la **implementación de una tarea** del **thread** que la ejecuta. Es más flexible.
- También es importante considerar que JAVA provee un conjunto de clases que gerencian **multithreading** (por ej. pool de threads): la **interface Runnable** es aplicable a APIs para gerenciamiento de threads.



# Ciclo de vida de un Thread



## Estado New Thread

Inmediatamente después que un thread es creado pasa a estado **New Thread**, pero aún no ha sido iniciado, por lo tanto no puede ejecutarse. Se debe invocar al método **start()**.

## Estado Running (Ejecutándose)/Runnable (Ejecutable)

Después de ejecutarse el método **start()** el thread pasa al estado **Runnable o Ejecutable**. Un thread arrancado con **start()** podría o no comenzar a ejecutarse. No hay nada que evite que el thread se ejecute. La JVM implementa una estrategia (scheduling) que permite compartir la CPU entre todos los threads en estado Runnable.

## Estado Not Runnable o Blocked (Bloqueado)

Un thread pasa a estado **Not Runnable o Bloqueado** cuando ocurren algunos de los siguientes eventos: se invoca al método **sleep()**, al **wait()**, **join()** ó **el thread está bloqueado en espera de una operación de I/O, el thread invoca a un método synchronized sobre un objeto y el lock del objeto no está disponible**. Cada entrada al estado **Not Runnable** tiene una forma de salida correspondiente. Cuando un thread está en estado bloqueado, el *scheduler* lo saltea y no le da ningún *slice* de CPU para ejecutarse.

## Estado Dead

Los **threads** definen su finalización implementando un **run()** que termine naturalmente.

# Ejemplo

```
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;

public class Reloj extends Applet implements Runnable {
    private Thread relojThread = null;

    public void start() {
        if (relojThread == null) {
            relojThread = new Thread(this, "Reloj");
            relojThread.start();
        }
    }

    public void run() {
        Thread miThread = Thread.currentThread();
        while (relojThread == miThread) {
            repaint();
            try {
                TimeUnit.MILLISECONDS.sleep(1000);
            } catch (InterruptedException e) {}
        }
    }

    public void paint(Graphics g) {
        Calendar cal = Calendar.getInstance();
        Date fecha = cal.getTime();
        DateFormat fechaFormateada = DateFormat.getTimeInstance();
        g.drawString(fechaFormateada.format(fecha), 5, 10);
    }

    public void stop() {
        relojThread = null;
    }
} // Fin de la clase Reloj
```

Se crea un objeto **Thread** y se le provee de un objeto **Runnable** en el constructor. Este objeto es el que implementará el método **run()**

Implementación de la **interface Runnable**. Esto indica que se implementa el método **run()**, no se hereda ninguna habilidad de **threading**

Se crea una instancia de **Thread**, **relojThread**. Estado **NEW THREAD**

Crea los recursos para ejecutar el thread, organiza la ejecución del thread e **invoca al método run()**. Estado **RUNNABLE**

Durante un segundo el thread está en estado **NOT RUNNABLE**

Esta asignación hace que la condición de continuación del **run()** deje de cumplirse y de esta manera el thread finaliza. Pasa a estado **DEAD**

# Prioridades en Threads

- En las configuraciones de computadoras en las que se dispone de una única CPU, los threads se ejecutarán de a uno a la vez simulando concurrencia. Uno de los principales beneficios del modelo de **threading** es que permite abstraernos de la configuración de procesadores.
- Cuando múltiples threads quieren ejecutarse, es el **SO el que determina a cuál de ellos le asignará CPU**. Los **programas JAVA pueden influir**, sin embargo la **decisión final es del SO**.
- Se llama **scheduling** a la estrategia que determina el orden de ejecución de múltiples threads sobre una única CPU.
- La **JVM soporta** un algoritmo de **scheduling simple** llamado **scheduling de prioridad fija**, que consiste en determinar el orden en que se ejecutarán los threads de acuerdo a la prioridad que ellos tienen.
- La prioridad de un thread le indica al **scheduler** cuán importante es.
- Cuando se crea un thread, éste hereda la prioridad del thread que lo creó (NORM\_PRIORITY ). Es posible modificar la prioridad de un thread después de su creación usando el método **setPriority(int)**. Las prioridades de los threads son números enteros que varían entre las constantes MIN\_PRIORITY y MAX\_PRIORITY (definidas en la clase Thread).

# Prioridades en Threads

- El sistema de ejecución de JAVA elige para ejecutar entre los threads que están en estado **Runnable** aquel que tiene prioridad más alta. Cuando éste thread finaliza, cede el procesador o pasa a estado **Bloqueado**, comienza a ejecutarse un thread de más baja prioridad.
- El **scheduler** usa una estrategia **round-robin** para elegir entre dos threads de igual prioridad que están esperando por la CPU. El thread elegido se ejecuta hasta que un thread de más alta prioridad pase a estado **Runnable**, ceda la CPU a otro thread, finalice el método run() ó, expire el tiempo de CPU asignado (time-slicing). Luego, el segundo thread tiene la posibilidad de ejecutarse.
- El algoritmo de **scheduling** también es **preemptive**: cada vez que un thread con mayor prioridad que todos los threads que están en estado **Runnable** pasa a estado **Runnable**, el sistema de ejecución elige el nuevo thread de mayor prioridad para ejecutarse.

# Ejecutores

Los **Ejecutores** simplifican la programación concurrente. Se incorporaron en JSE 5.

- Los **EJECUTORES** proveen una capa de indirección entre un cliente y la ejecución de una tarea. Es un objeto intermedio que ejecuta la tarea, desligando al cliente de la ejecución de la misma.
- Los **EJECUTORES** son objetos que encapsulan la creación y administración de **threads**, permitiendo **desacoplar** la tarea concurrente del mecanismo de ejecución. Entre sus responsabilidades están la creación, el uso y el *scheduling* de threads.
- Los **EJECUTORES** permiten modelar programas como una serie de tareas concurrentes asincrónicas, evitando los detalles asociados con **threads**: simplemente se crea una tarea que se pasa al ejecutor apropiado para que la ejecute.
- Un **EJECUTOR** es normalmente usado en vez de crear explícitamente **threads**:

**Con threads creados por el programador:**

```
Runnable r= new RunnableTask();  
new Thread(r).start();
```

**Con EJECUTORES:**

```
Executor e= unEjecutor;  
Runnable r= new RunnableTask();  
e.execute(r);
```

- Un **EJECUTOR** es un objeto que implementa la interface **Executor**.

```
package java.util.concurrent;  
public interface Executor {  
    public void execute();  
}
```

Construye el  
contexto apropiado  
para ejecutar objetos  
Runnable

# Ejecutores

## Con threads creados por el programador:

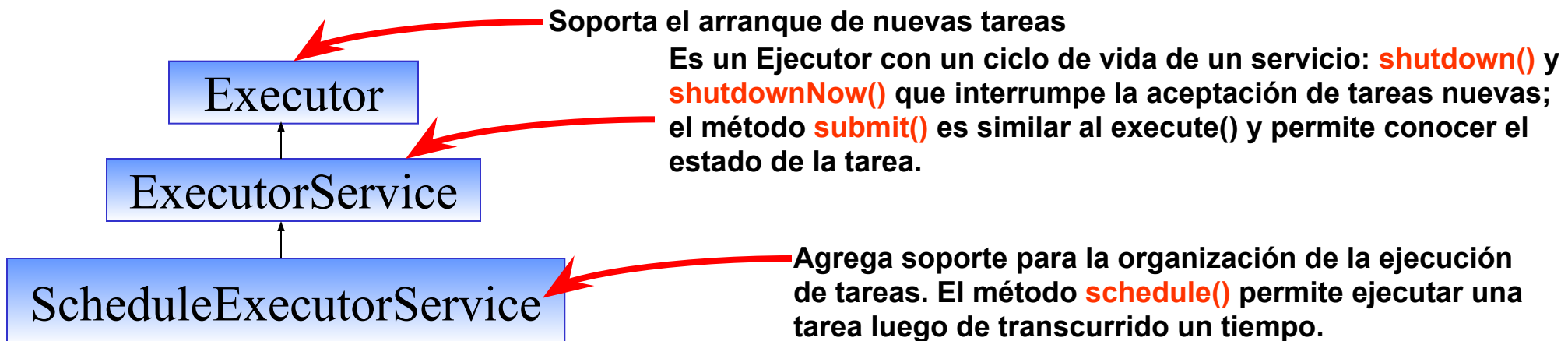
```
Runnable r= new RunnableTask();  
new Thread(r).start();
```

## Con EJECUTORES:

```
Executor e= unEjecutor;  
Runnable r= new RunnableTask();  
e.execute(r);
```

El comportamiento del método **execute()** es menos específico que el usado con **Threads**, siendo los **threads** creados y lanzamos inmediatamente. Dependiendo de la implementación del **Executor** el método **execute()** podría hacer lo mismo, o usar un **thread** existente disponible para ejecutar la tarea **r** o encolar **r** hasta que haya un **thread** disponible para ejecutar la tarea.

El paquete **java.util.concurrent** define tres interfaces Executor:



# Ejecutores & Pool de Threads

Típicamente las implementaciones de **EJECUTORES** del paquete **java.util.concurrent** usan *pool de threads*. Estos threads existen independientemente de las tareas Runnablees que ejecutan y generalmente ejecutan múltiples tareas.

El pool de threads minimiza la sobrecarga causada por la creación de nuevos threads=> **reuso de threads.**

Aumenta la *performance* de aplicaciones que ejecutan muchos **threads** simultáneamente. El pool adquiere un rol crucial en configuraciones donde se tienen más threads que CPUs => **programas más rápidos y eficientes.**

Para crear un **EJECUTOR** que **use una pool de threads** se puede invocar a los siguientes métodos de clase de la clase **java.util.concurrent.Executors**:

```
ExecutorService exec = Executors.newFixedThreadPool(int nThreads);  
ExecutorService exec = Executors.newFixedThreadPool(int nThreads, ThreadFactory threadFact);
```

Crea un pool de **threads** que reusa un conjunto finito de **threads**. En el 2do método se usa el objeto **ThreadFactory** para crear los **threads** necesarios.

```
ExecutorService exec = Executors.newCachedThreadPool();  
ExecutorService exec = Executors.newCachedThreadPool(ThreadFactory threadFact);
```

Crea un pool de **threads** que crea **threads** nuevos a medida que los necesita y reusa los construidos que están disponibles. El 2do método usa el objeto **ThreadFactory** para crear los nuevos **threads**

# Ejemplo 1

```
package concurrentes;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class CachedThreadPool {
    public static void main(String[ ] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for ( int i = 0; i < 5; i++)
            exec.execute(new SimpleThread()); Tarea Específica
        exec.shutdown();
    }
}
```

**ExecutorService exec = Executors.newFixedThreadPool(10);**



# Ejemplo 2

```
package concurrentes;
import java.util.concurrent.*;

public class SingleThreadExecutor {
    public static void main(String[] args){
        ExecutorService exec = Executors.newSingleThreadExecutor();
        for ( int i = 0; i < 5; i++)
            exec.execute(new SimpleThread());
        exec.shutdown();
    }
}
```

En este ejemplo una tarea se completa en el mismo orden en que es recibida y antes de comenzar una nueva.

Un **SingleThreadExecutor** es similar a **FixedThreadPool** con un pool de un único thread. Las tareas se encolan, cada tarea se ejecuta una vez que finaliza la previa. Todas usan el mismo thread. Este ejecutor **serializa** las tareas. Es útil para tareas que necesitan ejecutarse continuamente, por ej: escuchan conexiones entrantes, tareas que actualizan logs remotos o locales o que hacen *dispatching* de eventos. Otro ej: tareas que usan el



# Compartir Recursos

## Condición de Carrera

```
public class Counter
```

```
{
```

```
    private int c = 0;  
    public void increment()  
    {
```

```
        c++;
```

```
    }
```

```
    public void decrement()  
    {
```

```
        c- -;
```

```
    }
```

```
    public int value()  
    {
```

```
        return c;
```

```
    }
```

```
}
```

Si un mismo objeto **Counter** es **referenciado** por múltiples threads (por ejemplo A y B), la interferencia entre estos threads provocaría que el comportamiento de los métodos **increment()** y **decrement()** NO sea el esperado

Recuperar el valor actual de c.  
Incrementarlo/Decrementarlo en 1.  
Guardar en c el nuevo valor

¿Qué pasa si el thread A invoca al increment() al mismo tiempo que el thread B invoca decrement() sobre la misma instancia de Counter?

Si el valor inicial de c es 0, podría ocurrir lo siguiente:

**Thread A:** Recupera c. (c=0)

**Thread B:** Recupera c. (c=0)

**Thread A:** Incrementa el valor recuperado; resultado es c=1.

**Thread B:** Decrementa el valor recuperado; resultado es c=-1 (lo hace antes que A guarde el valor).

**Thread A:** Guarda el resultado en c; c=1

**Thread B:** Guarda el resultado en c; c=-1

**Condición de Carrera**

# Compartir Recursos

- Hasta ahora vimos ejemplos de **threads asincrónicos** que no comparten datos ni necesitan coordinar sus actividades.
- Con multithreading hay situaciones en que dos o más threads intentan acceder a los mismos recursos en el mismo momento. Se debe evitar este tipo de colisión sobre los recursos compartidos (durante períodos críticos): acceder a la misma cuenta bancaria en el mismo momento, imprimir en la misma impresora, etc. Ejemplo de la clase Counter
- Para resolver el problema de colisiones, todos los esquemas de multithreading establecen *un orden para acceder al recurso compartido*. En general se lleva a cabo usando una cláusula que *bloquea* (lock) el código que accede al recurso compartido y así solamente de a un thread a la vez se accede al recurso. Esta cláusula implementa **exclusión mutua**.
- Java provee soporte para **exclusión mutua** mediante la palabra clave **synchronized**.

# Compartir Recursos

- Cada objeto contiene un **lock** único llamado **monitor**. Cuando invocamos a un **método synchronized**, el objeto es “bloqueado” (locked) y ningún otro método **synchronized** sobre el mismo objeto puede ejecutarse hasta que el primer método termine y libere el **lock del objeto**.
- El **lock** del objeto es único y compartido por todos los métodos y **bloques synchronized** del mismo objeto. Este **lock** evita que el recurso común sea modificado por más de thread a la vez.

```
public class Recurso {  
    public synchronized int f() {}  
    public synchronized void g() {}  
}
```

Si el método f() es invocado sobre un objeto Recurso, el método g() no puede ejecutarse sobre el mismo objeto, hasta que f() termine y libere el lock.

- Es posible definir un bloque **synchronized**: **synchronized (unObjeto) {}**
- Un thread puede adquirir el **lock** de un objeto múltiples veces. Esto ocurre si un método invoca a un segundo método **synchronized** sobre el mismo objeto, quién a su vez invoca a otro método **synchronized** sobre el mismo objeto, etc. La JVM mantiene un contador con el número de veces que el objeto fue bloqueado (lock). Cuando el objeto es desbloqueado, el contador toma el valor cero. Cada vez que un thread adquiere el lock sobre el mismo objeto, el contador se incrementa en uno y cada vez que abandona un método **synchronized** el contador se decrementa en uno, hasta que el contador llegue a cero, liberando el lock para que lo usen otros threads. La adquisición del lock múltiples veces sólo es permitida para el thread que lo adquirió en el primer método **synchronized** que invocó.

# Compartir Recursos

## La cláusula synchronized

```
public class SynchronizedCounter
{
    private int c = 0;
    public synchronized void increment()
    {
        c++;
    }
    public synchronized void decrement()
    {
        c--;
    }
    public synchronized int value()
    {
        return c;
    }
}
```

# Ejemplo

## Producer/Consumidor

```
public class Productor extends Thread {  
    private Bolsa bolsa;  
    private int numero;  
  
    public Productor(Bolsa c, int numero) {  
        bolsa = c;  
        this.numero = numero;  
    }  
  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            bolsa.put(i);  
            System.out.println("Productor #" + this.numero  
                + " escribió: " + i);  
            try {  
                TimeUnit.MILLISECONDS.sleep((int)(Math.random() * 100));  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```

Objeto Compartido

```
public class Consumidor extends Thread {  
    private Bolsa bolsa;  
    private int numero;  
  
    public Consumidor(Bolsa c, int numero) {  
        bolsa = c;  
        this.numero = numero;  
    }  
  
    public void run() {  
        int valor = 0;  
        for (int i = 0; i < 10; i++) {  
            valor = bolsa.get();  
            System.out.println("Consumidor#" +  
                this.numero + " leyó: " + valor);  
        }  
    }  
}
```

# Ejemplo

## Productor/Consumidor

Para evitar que el Productor y el Consumidor colisionen sobre el objeto compartido Bolsa, esto es, que intenten guardar y leer simultáneamente dejando al objeto en un estado inconsistente, los métodos `get()` y `put()` se declaran **synchronized**

```
public class Bolsa {  
    private int contenido;  
    private boolean disponible = false;  
    public synchronized int get() {  
        //El objeto bolsa fue bloqueada por el Consumidor  
  
        .....  
        //el objeto bolsa fue desbloqueada por el Consumidor  
    }  
  
    public synchronized void put(int value) {  
        // El objeto bolsa fue bloqueada por el Productor  
  
        .....  
        //el objeto bolsa fue desbloqueado por el Productor  
    }  
}
```

Secciones críticas

# Productor/Consumidor

¿Qué sucede si el Productor es más rápido que el Consumidor y genera dos números antes que el consumidor pueda consumir el primero de ellos?

.....

Consumidor #1 leyó: 3

Productor #1 escribió: 4

Productor #1 escribió: 5

Consumidor #1 leyó: 5

← El Consumidor perdió el 4

¿Qué sucede si el Consumidor es más rápido que el Productor y consume dos veces el mismo valor?

.....

Productor #1 escribió: 4

Consumidor #1 leyó: 4

Consumidor #1 leyó: 4

Productor #1 escribió: 5

← El Consumidor obtiene el 4 dos veces

En ambos casos el resultado es erróneo dado que el Consumidor debe leer cada uno de los números producidos por el Productor exactamente una vez.



# Cooperación entre Threads

¿Cómo podemos hacer para que el **Productor** y el **Consumidor** cooperen entre ellos?

El **Productor** debe indicarle al **Consumidor** de una manera sencilla que el valor está listo para ser leído y el **Consumidor** debe tener alguna forma de indicarle al **Productor** que el valor ya fue leído.

Además, si no hay nada para leer, el **Consumidor** debe esperar a que el **Productor** escriba un nuevo valor y, el **Productor** debe esperar a que el **Consumidor** lea antes de escribir un valor nuevo.

Para este propósito la clase **Object** provee los siguientes métodos: **wait()**, **wait(milisegundos)**, **notify()** y **notifyAll()**.

Los métodos **wait()**, **wait(milisegundos)**, **notify()** y **notifyAll()** deben usarse adentro de un método o bloque **synchronized**.

**wait()**  
**wait(milisegundos)**  
**notify()**  
**notifyAll()**

El método **wait()** suspende la ejecución del thread y libera el *lock* del objeto, y así permite que otros métodos **synchronized** sobre el mismo objeto puedan ejecutarse.

El método **notifyAll()** “despierta” a todos los threads esperando (**wait()**), compiten por el *lock* y el que lo obtiene retoma la ejecución. El método **notify()** despierta a un thread.

# Productor/Consumidor

```
public class Bolsa {  
    private int contenido;  
    private boolean disponible = false;  
    public synchronized int get() {  
        while (disponible == false) {  
            try { wait(); } ←  
            catch (InterruptedException e) { .. }  
        }  
        disponible = false;  
        notifyAll(); ←  
        return contenido;  
    }  
    public synchronized void put(int value) {  
        while (disponible == true) {  
            try { wait(); } ←  
            catch (InterruptedException e) { .. }  
        }  
        contenido = value;  
        disponible = true;  
        notifyAll(); ←  
    }  
}
```

Libera el lock (del objeto Bolsa) tomado por el Consumidor, permitiendo que el Productor agregue un dato nuevo en la Bolsa y, luego espera ser notificado por el Productor.

El Consumidor notifica al Productor que ya leyó, dándole la posibilidad de producir un nuevo valor.

Libera el lock (del objeto Bolsa) tomado por el Productor, permitiendo que el Consumidor lea el valor actual antes de producir un nuevo valor.

El Productor notifica al Consumidor cuando agregó un dato nuevo en la Bolsa

# Producer/Consumidor

```
public class ProductorConsumidorTest {  
    public static void main(String[] args) {  
        Bolsa c = new Bolsa();  
        Productor p1 = new Productor(c, 1);  
        Consumidor c1 = new Consumidor(c, 1);  
  
        p1.start();  
        c1.start();  
    }  
}
```

Productor #1 escribió: 0  
Consumidor #1 leyó: 0  
Productor #1 escribió: 1  
Consumidor #1 leyó: 1  
Productor #1 escribió: 2  
Consumidor #1 leyó: 2  
Productor #1 escribió: 3  
Consumidor #1 leyó: 3  
.....  
Productor #1 escribió: 9  
Consumidor #1 leyó: 9

Salida del programa  
ProductorConsumidorTest