Genéricos en JAVA

Introducción

- En JAVA se denomina "Genéricos" a la capacidad del lenguaje de **definir** y **usar tipos** (clases e interfaces) y **métodos genéricos**.
- Los **tipos genéricos** difieren de los "regulares" en que contienen **tipos de datos** como **parámetros formales.**
- Los **tipos** y **los métodos genéricos** se incorporan en JAVA 5 para proveer **chequeo de tipos en compilación en colecciones**.
- No pueden declararse genéricos los tipos enumerativos, las clases anónimas y subclases de excepciones.

```
public class LinkedList <E>  extends AbstractSequentialList <E> implements List<E>, Queue<E>, Cloneable, Serializable
```

LinkedList es un tipo Genérico

E es un parámetro formal que denota un tipo de dato

Los elementos que se almacenan en la lista encadenada son del tipo desconocido E

Con tipos genéricos podemos definir: **LinkedList<String> y LinkedList<Integer>**

- Una **clase genérica** tiene el **mismo comportamiento** para todos sus posibles tipos de parámetros.
- Los **tipos parametrizados** se forman a partir de los **tipos genéricos** al asignarle **tipos reales a los parámetros formales**:

```
LinkedList<E>, Comparator<T>
LinkedList<String>   listaStr;
LinkedList<Integer>  listaInt = new LinkedList<>();
Comparator<String> compara;
```

Tipos Genéricos

Lista de Strings

Lista de Enteros

Comparador de Strings

Tipos Parametrizados

- El **framework de colecciones** del paquete **java.util es genérico** a partir de Java 5.

¿Qué problemas resuelven los "Genéricos"?

En colecciones se **evitan los errores en ejecución** causados por el uso de ***casting*** .

```
List list = new ArrayList(); // Lista que contiene Strings
list.add("abc");
list.add(new Integer(5));

for(Object obj : list){

   String str=(String) obj;
}
```

El compilador devuelve una advertencia "unchecked call"

Sin genéricos

El casting dispara el siguiente error en ejecución: "ClassCastException"

```
List<String> list1 = new ArrayList<>();
list1.add("abc");
list1.add(new Integer(5)); // error de compilación
//"error: incompatible types: Integer cannot be converted"
for(String str : list1){
      System.out.print(str);
}
```

Con genéricos

No es necesario hacer casting

Los **Genéricos** ofrecen una **mejora para el sistema de tipos**:

- permite operar sobre objetos de múltiples tipos.

- provee **seguridad** en compilación pudiendo detectar *errores* en compilación.

Los programas que usan genéricos son **seguros**, reusables, es un **código limpio**.

La **inserción errónea genera un mensaje de error en compilación** que indica exactamente qué es lo que está mal.

Tipos Genéricos y Tipos Parametrizados

Un **tipo genérico** es un **tipo de datos con parámetros formales que denotan tipos de datos.**
Un **tipo parametrizado** en una **instanciación de un tipo genérico** con **argumentos que son tipos reales**.

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

INTERFACE GENÉRICA

Un **tipo genérico** es un tipo de datos que **tiene uno o más parámetros formales** que representan tipos de datos.

Cuando el tipo genérico es instanciado o declarado, se reemplazan los parámetros formales por argumentos que representan tipos reales.

La **interface Collection** tiene un parámetro formal **E** que indica un tipo de datos. En la declaración de una colección específica, **E** es reemplazado por un tipo real.

La **instanciación de un tipo genérico** se denomina **tipo parametrizado**.

```
Collection <String> col=new LinkedList<String>();
List <String> list= new ArrayList<>();  // a partir de JAVA 7
Collection <? extends Number> col=new LinkedList<Integer>();
```

TIPO PARAMETRIZADO

Declaración de tipos Genéricos

En la definición de los **tipos genéricos** la sección correspondiente a los parámetros continúa al nombre de la clase o interface. Es una lista separada por comas y delimitada por los símbolos <>.

```
package genericos.definicion;
public class ParOrdenado <X,Y> {
private X a;
private Y b;
public ParOrdenado (X a, Y b){
  this.a=a;
  this.b=b;
}
public X getA() {   return a; }
public void setA(X a) {   this.a = a; }
public Y getB() {  return b; }
public void setB(Y b) {   this.b = b; }
}
```

El alcance de los identificadores **X** e **Y** es toda la clase **ParOrdenado**.

En el ejemplo, **X** e **Y** son usados en la declaración de variables de instancia y como argumentos y tipos de retorno de los métodos de instancia.

Los parámetros formales (tipos de datos) pueden declararse con cotas. Las cotas proveen acceso a métodos del tipo definido en el parámetro formal (que es desconocido) .

En el ejemplo de la clase **ParOrdenado** no invocamos a ningún método sobre los tipos desconocidos X e Y, es por esta razón que los 2 tipos son sin cotas.

Tipos Parametrizados Concretos

Para usar un **tipo genérico** se deben especificar los **argumentos** que reemplazarán a los **parámetros formales**.

Los argumentos son referencias a **tipos concretos** como String, Long, LocalDate, etc.

```
package genericos.definicion;
import java.awt.Color;
public class TestParOrdenado {
public static void main(String[] args){
    ParOrdenado<String, Long> par = new ParOrdenado<>("hola", 23L);
    System.out.println("("+par.getA()+","+par.getB()+")");
   ParOrdenado<String,Color> nombreColor = new ParOrdenado<>("Rojo", Color.RED);
   System.out.println("("+nombreColor.getA()+","+nombreColor.getB()+")");
    ParOrdenado<Double,Double> coordenadas = new ParOrdenado<>(17.3,42.8);
   System.out.println("("+coordenadas.getA()+","+coordenadas.getB()+")");
  }
}
```

Autoboxing: el 23 es automáticamente convertido a Long

```
(hola,23)
(Rojo,java.awt.Color[r=255,g=0,b=0])
(17.3,42.8)
```

La **instanciación** de ParOrdenado<String,Long>, ParOrdenado<String,Color>, ParOrdenado<Double,Double> son **tipos parametrizados concretos** y se usan como un tipo regular.

Podemos usar **tipos parametrizados** como **argumentos de métodos**, para **declarar variables** y en la expresión **new** para crear un objeto.

Tipos Parametrizados con Comodines

Son **instanciaciones de tipos genéricos** que contienen al menos un comodín (**?**) como tipo de argumento. Un comodín es una construcción sintáctica "**?**" que denota la familia de "todos los tipos".

No son tipos concretos, no pueden usarse en la sentencia new.

```
static int cantElemEnComun(Set<?> s1, Set<?> s2) {
    int result = 0;
    for (Object o1 : s1)
        if (s2.contains(o1)) result++;
    return result;
}
```

Tipos parametrizados comodines sin cota

El **?** indica un conjunto de "algún tipo desconocido".

Es el **conjunto parametrizado más general**, capaz de contener cualquier tipo de elemento. No interesa cuál es el parámetro del tipo real.

Puede ser Set<String>, Set<A>, Set<Long>, etc.

Set<?> representa la flia de todos los conjuntos de "cualquier tipo". No nos interesa el tipo real, es un conjunto que puede contener cualquier tipo.

Se pueden usar para **declarar variables o parámetros de métodos**, como s1 y s2, pero no en una sentencia **new** para crear objetos.

Como no sabemos nada del tipo de los elementos del Set, **solo se puede leer** y **tratar los objetos leídos como instancias de Object**.

Los **tipos parametrizados con comodines sin cota** son útiles en situaciones en las que no es necesario **conocer nada sobre el tipo del argumento**. **¿Cuál es la diferencia entre Set<?> y Set?**

Tipos Parametrizados con Comodines

Un **comodín con una cota superior "? extends T"** es la **familia de todos los tipos** que son **subtipos de T**. **T** es la **cota superior**.

Un **comodín con una cota inferior "? super T"** es la **familia de todos los tipos** que son **supertipos de T**. **T** es la **cota inferior**.

Los **tipos parametrizados con comodines acotados** son útiles en situaciones en las que es necesario contar con un **conocimiento parcial sobre el tipo de argumento** de los tipos parametrizados.

```
List<? extends Number> l;
```

La **familia de todos los tipos de listas** cuyos elementos son **subtipos de Number.**

```
Comparable<? super  String>  s;
```

La **familia de todas las implementaciones Comparable** para tipos que son **supertipos de String.**

```
public final class Byte extends Number implements Comparable<Byte>
public final class Double extends Number implements Comparable<Double>
public final class Float extends Number implements Comparable< Float >
public final class Integer extends Number implements Comparable<Integer>
public final class Long extends Number implements Comparable<Long>
public final class Short extends Number implements Comparable<Short>
```

Subtipos de Number

```
String, Object, CharSequence, Serializable  y Comparable<String>
```

Supertipos de String

Tipos Parametrizados Ejemplos

Suma de los números de una lista de números:

```
public static double sum(List<Number> list){
double sum = 0;
for(Number n : list){
 sum += n.doubleValue();
}
return sum;
}
```

Los **tipos parametrizados concretos son invariantes** por lo tanto List<Double> y List<Integer> no están relacionados con List<Number>, no tienen relación de subtipo ni supertipo y el método sum() no se puede usar.

Los **tipos parametrizados con comodines acotados** ofrecen mayor **flexibilidad** que los tipos invariantes.

```
public static double sum(List<? extends Number> list){
double sum = 0;
for(Number n : list){
sum += n.doubleValue();
}
return sum;
}
```

```
public static void main(String[] args) {
List<Integer> ints = new ArrayList<>();
ints.add(3); ints.add(5); ints.add(10);
double sum = sum(ints);
System.out.println("Suma de ints="+sum);
}
```

Tipos Parametrizados con Comodines Resumen

Un tipo parametrizado con comodín no es un tipo concreto.

Pueden declararse variables del tipo parametrizado con comodín, pero no pueden crearse objetos con el operador **new**. En ese sentido **son similares a las interfaces**.

Las **variables de tipo parametrizado con comodín** hacen referencia a un objeto perteneciente a la **familia de tipos que el tipo parametrizado comodín denota**.

```
List<?> col= new ArrayList<String>();   Lista de algún tipo
List<? extends Number> lista=new ArrayList<Long>();   Lista de subtipos de Number
ParOrdenado<String, ?> par=new ParOrdenado<String, String>();   Par ordenada con abscisa de algún tipo
```

```
List<? extends Number> l = new ArrayList<String>(); ERROR!!!!
```

String no es subtipo de Number y consecuentemente **ArrayList<String>** no pertenece a la familia de tipos denotados por **List<? extends Number>**.

List<?> denota una lista de elementos de algún tipo, desconocido. Es una lista de "sólo lectura".

Reglas de subtipos para genéricos

Los **tipos parametrizados** forman una jerarquía de tipos basada en el tipo base NO en el tipo de los argumentos. Los **tipos parametrizados son invariantes**.

```
public interface List<E> extends Collection<E> {}
public class ArrayList<E> extends AbstractList<E>  implements List<E> {}
```

```
List<Integer> listita = new ArrayList<>();
List<Integer> l= listita;
Collection<Integer> c = listita;
ArrayList<Number> n = listita;
List<Object> o = listita;
List li= listita;
```

¿Cuáles asignaciones cumplen las reglas de subtipo?

Un ArrayList<Integer> es un List<Integer>, un Collection<Integer> y un List, pero NO es un ArrayList<Number> ni un List<Object> .

```
List<Integer> li = new ArrayList<>();
li.add(123);
List<Number> lo = li;
Number nro = lo.get(0);
lo.add(3.14);
Integer i = li.get(1);
```

¿Es válido este código?

Reglas de subtipos para genéricos

Los **tipos parametrizados** forman una jerarquía de tipos basada en el tipo base NO en el tipo de los argumentos. Los **tipos parametrizados son invariantes**.

```
public interface List<E> extends Collection<E> {}
public class ArrayList<E> extends AbstractList<E>  implements List<E> {}
```

```
List<Integer> listita = new ArrayList<>();
List<Integer> l= listita;          ✓
Collection<Integer> c = listita;   ✓
ArrayList<Number> n = listita;     ✗
List<Object> o = listita;          ✗
List li= listita;                  ✓
```

Un ArrayList<Integer> es un List<Integer>, un Collection<Integer> y un List, pero NO es un ArrayList<Number> ni un List<Object> .

List<Integer> no es un subtipo de List<Number>

```
List<Integer> li = new ArrayList<>();
li.add(123);
List<Number> lo = li;
Number nro = lo.get(0);
lo.add(3.14);
Integer i = li.get(1);
```

No compila List<Number> lo=li; NO ES POSIBLE CONVERTIR DE List<Integer> a List<Number>

Si asumimos que compila:

- podríamos recuperar elementos de la lista como Number en vez de como Integer: Number nro = lo.get(0);
- podríamos agregar un objeto Double: lo.add(3.14);
- la línea li.get(1); daría error de *casting*, porque no puedo *castear* un Double a un Integer

Tipos Parametrizados con Comodines Acotados - Ejemplos

```
interface Collection<E> {
      public boolean addAll( Collection<? extends E>  c );
}
```

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

addAll() agrega todos los elementos de una colección en otra colección

"? extends E": está permitido agregar a una colección de tipos E elementos de otra colección que son subtipo de E.

El parámetro **c** de **allAll() produce** elementos para la colección, por lo tanto los **elementos de c** deben ser subtipos de E.

```
package genericos;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
public class TestComodinConExtends {
  public static void main(String[] args) {
      List<Number> nums = new ArrayList<>();
      List<Integer> ints = Arrays.asList(1, 2);
      List<Double> dbls = Arrays.asList(2.78, 3.14);
      nums.addAll(ints);
      nums.addAll(dbls);
      }
}
```

- **nums** es de tipo **List<Number>** que es subtipo de **Collection<Number>**
- **ints** es de tipo **List<Integer>** que es subtipo de **Collection<? extends Number>**
- **dbls** es de tipo **List<Double>** que es subtipo de **Collection<? extends Number>.**
- **E** toma el valor **Number**.

Si el parámetro de addAll() estuviese escrito sin comodines es decir Collection<E>: **¿Podría agregarse a *nums* una lista de enteros y de números decimales?**

NO!! Solamente estaría permitido agregar listas que estuviesen explícitamente declaradas como listas de **Number.**

¿Qué está disponible a través de variables de tipo parametrizado con comodín?

```
public static void printList(PrintStream out,  List<?>  lista) {
//   lista.add("hola");
   for(int i=0, n=lista.size(); i < n; i++) {
        if (i > 0) out.println(", ");
        Object o = lista.get(i);
         out.print(o.toString());
   }
}
```

```
interface List<E> {
    boolean add (E element);
    E get(int index)
}
```

El método **printList()** imprime todos los elementos de la lista pasada como parámetro.

- El método **add(E e)** de List<E> acepta un argumento del tipo especificado por el parámetro **E**. En nuestro ejemplo el tipo es "**?**" (desconocido) por lo tanto el compilador no puede asegurar que el objeto pasado como parámetro al método add() es del tipo esperado por el método. **Por lo tanto, List<?> es de sólo lectura**, **no es posible invocar a los métodos add( ), set() y addAll( ) de List.** **Código Seguro.**

- El método **E get(int)** de List<E> devuelve un valor que es del mismo tipo que el parámetro **E**. En nuestro ejemplo el tipo es "**?**" (desconocido) por lo tanto el método get() puede ser invocado y el resultado puede asignarse a un variable de tipo Object (sabemos que será un objeto).
- **Conclusión:** los tipos parametrizados con comodines son de solo lectura.
- Si en lugar de List<?> usamos List, **¿ocurre lo mismo? ¿cuál es la diferencia?**

¿Qué está disponible a través de variables de tipo parametrizado comodín?

```
List<Integer> li = new ArrayList<>();
li.add(123);
List<? extends Number> lo = li;
Number nro = lo.get(0);
lo.add(3.14);
Integer i = li.get(1);
```

¿Se puede hacer?

- **Se puede asignar li a lo** porque **li** es de tipo **List<Integer>** y es subtipo de **List<? extends Number>.**
- **No se puede agregar un Double a lo** (**List<? extends Number>**) dado que podría ser una lista de algún otro subtipo de Number.

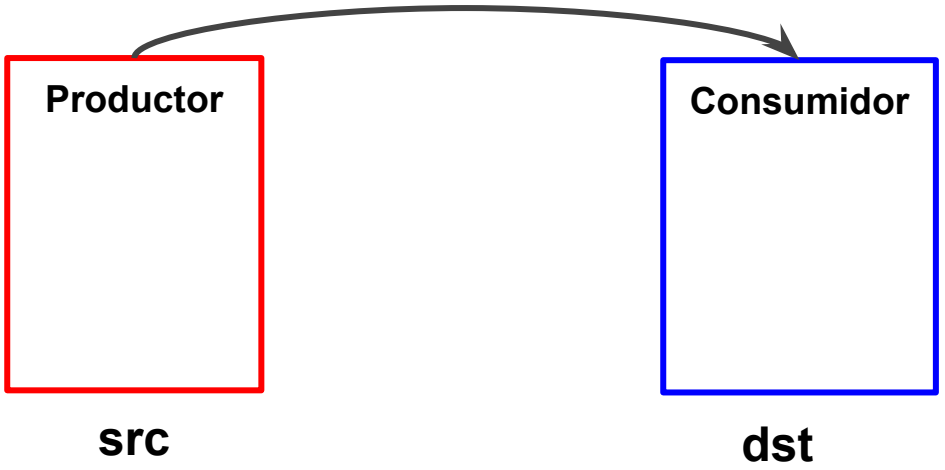
En general: si una variable referencia a una colección que declara contener elementos de tipo **<? extends E>** **es posible recuperar elementos** de la estructura pero **NO es posible agregar elementos a la estructura**. **Es una variable de solo lectura.**

Tipos Parametrizados con Comodines Acotados - Ejemplos

```
public class Collections{
    public static <T> void copy( List<? super T> dst , List<? extends T> src) {
        for (int i = 0; i < src.size(); i++) { dst.set(i, src.get(i)); }
    }
}
```

El método **copy()** de la clase **Collections** copia elementos desde una **lista fuente (src)** a una **destino (dst)**

La **lista destino** tiene que ser capaz de **guardar los elementos de la lista fuente**.



Paradigma productor-consumidor:

productor-extends, **consumidor-super**

- src **produce** para dst y dst **consume** de src.
- **Los elementos de src deben ser subtipos de los de dst.**
- **Los elementos de dst deben ser supertipos de los de src.**

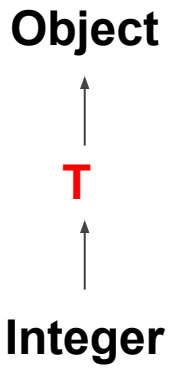
copy: **es un método genérico**, acepta argumentos de tipo List<? super T> y List<? extends T>, devuelve void y se aplica a cualquier tipo T.

List<? super T> dst: la lista destino puede contener elementos de cualquier tipo que sea supertipo de T.

List<? extends T> src: la lista fuente puede contener elementos de cualquier tipo que sea subtipo de T.

Tipos Parametrizados con Comodines Acotados - Ejemplos

```
public class Collections{
public static <T> void copy( List<? super T> dst , List<? extends T> src) {
      for (int i = 0; i < src.size(); i++) { dst.set(i, src.get(i)); }
}
}
```



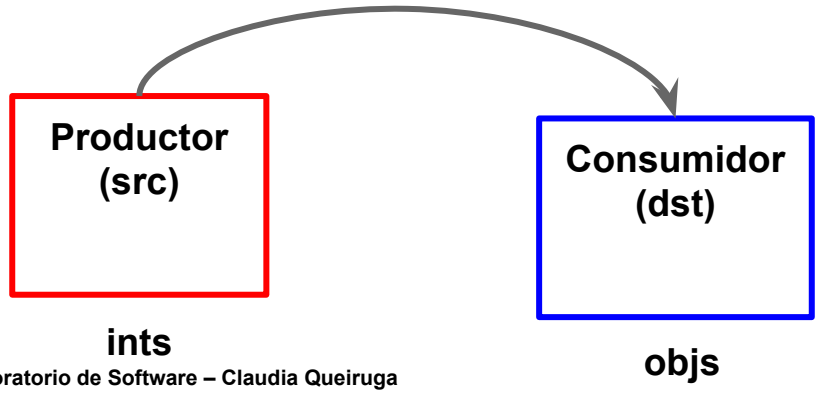

```
package genericos;
import java.util.*;
public class TestMetodosGenericos {
      public static void main(String[] args) {
      List<Object> objs = Arrays.asList(2, 3.14, "hola");
      List<Integer> ints = Arrays.asList(5, 6);
      Collections.copy(objs, ints);
      }
}      Collections.<Number>copy(objs, ints);
```

[5, 6, hola]

Integer es subtipo de T y **Object es supertipo de T**. El compilador elige dentro de los posibles, por ejemplo **Number**

```
List<String> strs = Arrays.asList("2", "hola", "3.14" );
List<Integer> ints2 = Arrays.asList(5, 6);
Collections.copy(strs, ints2);
```

NO ES APLICABLE




```
public static <E> Set<E> union(Set<? extends E> s1, Set<? extends E> s2)
```

S1 y s2 son ambos productores, sus elementos se guardarán en el conjunto receptor del método union()

Métodos Genéricos

De la misma manera que los tipos (clases e interfaces) los **métodos** pueden definirse **genéricos**, es decir pueden ser parametrizados por uno o más tipos de datos.

```
public static <T> int countOccurrences(T[] list, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
   for (T listItem : list)
   if (listItem == null)
   count++;
   } else {
   for (T listItem : list)
   if (itemToCount.equals(listItem))
   count++;
   }
   return count;
}
```

countOccurrences() cuenta la cantidad de **ocurrencias** del elemento **itemToCount** en el arreglo genérico **list** y se aplica a **cualquier tipo T**

Declarar métodos genéricos es similar a declarar tipos genéricos pero el alcance del tipo como parámetro está limitado al método.

Los métodos genéricos expresan dependencias entre los tipos de los argumentos y/o el tipo de retorno del método.

```
class Collections {
public static <T> void copy(List<T> dest, List<? extends T> src) { }
}
```

Cuando se usa un método genérico no hay una mención explícita al tipo que sustituirá al tipo del parámetro formal. El compilador infiere el tipo a partir de los parámetros reales. **Inferencia de tipos**

En nuestro caso como list es un arreglo de Number, entonces T es un Number.

```
countOccurrences (arrNumber,10);
Number arrNumber[]=new Number[5];
arrNumber[0]=10.9; arrNumber[1]=5L;
arrNumber[2]=15; arrNumber[3]=10;
arrNumber[4]=10;
```

Métodos Genéricos

```
public static <T extends Comparable <? superT>> T max(Collection<? extends T> coll)
```

max devuelve el **mayor valor de una colección de elementos de un tipo desconocido T**, de acuerdo al orden natural de sus elementos.

Aquí el parámetro **coll produce** números que cumplen relaciones de orden.

Los **Comparables** y **Comparator** siempre son **consumidores**.

Los métodos genéricos se invocan de la forma usual, los argumentos que representan a los tipos concretos no necesitan ser explicitados, son **inferidos automáticamente**.

```
import java.util.*;
public class TestMetodos {
public static void main(String[] args) {
    List<String> strList=new ArrayList<>();
    strList.add("hola");
    strList.add("chau");
    strList.add("hi");
    strList.add("bye");
    System.out.println(Collections.max(strList));
}
}
```

<T extends Comparable <? superT>> T

Cualquier tipo T que sea comparable es decir que permita comparar objetos entre sí.

El compilador invoca automáticamente al método max() usando como argumento el tipo String.

El compilador infiere automáticamente el tipo del argumento.

Tipos Parametrizados con Comodines
Algunos tips

Si se usan correctamente los **tipos comodines son invisibles** para los usuarios de una clase.

Los **métodos aceptan los parámetros** que deben aceptar y **rechazan** los que deberían rechazar.

Si el usuario de una clase tiene que pensar en tipos de comodines, probablemente haya algo mal con la API.

Ejemplos

```
public class TestListaParametrizada {
   public static void main(String[] args) {
       List<String> listaPalabras = new ArrayList<>();
       //listaPalabras.add(args); // ERROR DE COMPILACIÓN!!!
       for(String arg : args)
              listaPalabras.add(arg);
       String unaPalabra = listaPalabras.get(0); // CASTING  AUTOMÁTICO!!!
   }
}
```

SIN TIPOS GENÉRICOS un error accidental en la inserción causaría una falla de EJECUCIÓN

Map es un tipo genérico del framework de colecciones con 2 parámetros que representan tipos de datos: uno representa el tipo de las claves y el otro el tipo del valor de cada clave.

```
public interface Map<K,V> {}
```

```
public class TestMapParametrizado{
   public static void main(String[] args) {
       Map<String,Integer> tabla = new HashMap<>();
       for(int i=0; i < args.length; i++)
          tabla.put(args[i], i);
       int posicion =tabla.get("hola"); // CASTING  AUTOMÁTICO!!!
       }
}
```

Operaciones de boxing y unboxing permiten convertir automáticamente de tipos primitivos a clases *wrapper*

java TestMapParametrizado chau hola adiós

Ventajas de usar Genéricos

Detección temprana de errores

El compilador puede realizar más chequeos de tipos. Los errores son detectados tempranamente y reportados por el compilador en forma de mensajes de error en lugar de ser detectados en ejecución mediante excepciones.

```
package genericos;
import java.util.Date;
import java.util.LinkedList;
public class TestErrores {
    public static void main(String[] args) {
    List<String> list= new LinkedList<String>();
    list.add("hola");
    list.add(new Date());
}
}
```

Es una lista homogénea de strings

El compilador chequea que la lista sólo contenga strings. En otro caso, el compilador lo rechaza

Con **tipos no-parametrizados** (LinkedList) es posible agregar diferentes tipos de elementos a la colección. La compilación será exitosa.

```
List list= new LinkedList ();
list.add("hola");
list.add(new Date());
```

Ventajas de usar Genéricos

Detección temprana de errores (Continuación)

```
package genericos;
import java.util.*;
public class TestErrores {
  public static void main(String[] args) {
  List<String> list= new LinkedList<String>();
  list.add("hola");
  String str=list.get(0);
  }
}
```

Los elementos se recuperan sin realizar *casting*. Está garantizado que la lista contiene strings.

Con **tipos no-parametrizados** (LinkedList) no hay conocimiento ni garantías respecto del tipo de los elementos que se recuperan. Todos los métodos retornan referencias a **Object** que deben ser *downcasteadas* al tipo real del elemento a recuperar.

```
List list= new LinkedList();
list.add("hola");
list.add(new Date());
String str=(String)list.get(0);
```

El *casting* podría causar errores en ejecución, **ClassCastException**, en caso que el elemento recuperado no sea un string

Ventajas de usar Genéricos

Seguridad de Tipos

En JAVA se considera que un programa es **seguro respecto al tipado** si compila sin errores ni advertencias y en ejecución NO dispara ningún **ClassCastException**.

Un programa bien formado permite que el compilador realice suficientes **chequeos de tipos basados en información estática** y que no ocurran errores inesperados de tipos en ejecución. **ClassCastException** sería un error inesperado de tipos que se produce en ejecución sin ninguna expresión de casting visible en el código fuente.

Interoperabilidad con código *legacy*

¿Cómo se implementan en JAVA los tipos genéricos?

Mediante la **técnica llamada "borrado" o *erasure***: el compilador usa la información de los tipos genéricos y tipos parametrizados para compilar y luego elimina la información del TIPO.

List<Integer>, List<String> y List<String> son traducidas a **List**. El bytecode es el mismo que el de **List**.

Después de la traducción por "borrado" desaparece toda la información del TIPO.

- **Simplicidad**: hay una única implementación de List, no una versión de cada tipo.
- **Compatibilidad**: la misma librería puede ser accedida tanto por genéricos como por no-genéricos.
- **Evolución Fácil**: no es necesario cambiar todo el código a genéricos, es posible evolucionar el código a genéricos actualizando de a un paquete, clase o método.

Podremos mantener versiones del código fuente de nuestras librerías que usen tipos genéricos y otras que usen raw types, es decir sin tipos genéricos. Es posible invocar a un método diseñado con tipos parametrizados con tipos sin parametrizar y viceversa. **Compatibilidad de Migración**.

Compatibilidad entre tipos parametrizados y raw types

Es compatible la asignación entre un raw type y todas las instanciaciones de un tipo genérico.

La asignación de un tipo parametrizado a un raw type está permitida; la asignación de un raw type a uno parametrizado produce advertencias en compilación **"conversiones no chequeadas/seguridad de tipos"**.

Es importante tener en cuenta que se rompe la seguridad de tipos genéricos.

```
List<Integer> li = new ArrayList<Integer>();  // tipo parametrizado
List lo=new ArrayList (); // raw type
li.add(123);
lo=li;
Object nro = lo.get(0);
lo.add("hola");  // inserción de un tipo no permitido no se detecta en compilación
Integer i = li.get(1);
```

La advertencia indica que el compilador desconoce si el ArrayList que se está asignando contiene o no Integer

Este código compila, pero en ejecución dispara un error de casting: ClassCastException

Excepciones

Excepciones

- Una **excepción** es un **evento** o **problema** que ocurre durante la **ejecución de un programa** e **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.

- Las **excepciones** ocurren en los **métodos** y 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) se **lanza la excepción** es decir 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** que consiste en 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** estos errores. 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 lo 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 de archivo o propagar la excepción; otro error posible es intentar ejecutar una sentencia sql errónea.

NO Verificables en Compilación

- **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.

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

in=new FileReader(filename);	c=in.read();
El nombre pasado como parámetro al constructor de la clase **FileReader** podría no existir en el *file system*, por tanto el constructor dispara la excepción: **java.io.FileNotFoundException.**	El método **getWord()** de la clase **InputFile** lee del objeto **FileReader** creado en el constructor de la clase usando el método **read()**. Este método dispara la excepción: **java.io.IOException** 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 claúsula *throws***.**

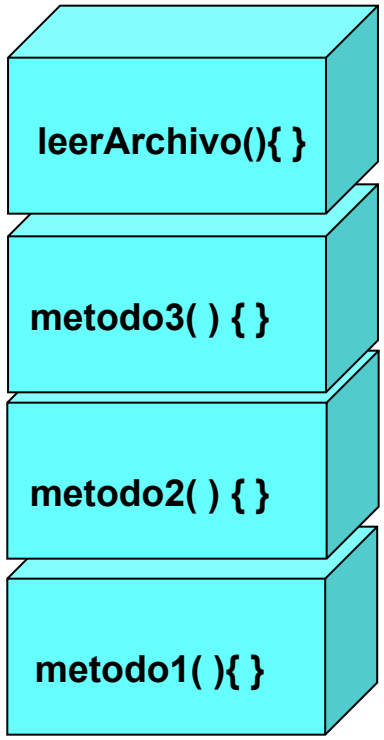
Búsqueda del Manejador de Excepciones

- 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 un **manejador de excepciones adecuado para dicho error** en la **pila de ejecución de métodos**.
- 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.




```
leerArchivo () {
  .....
}
```
→ **dispara una excepción**

```
metodo3(){
  LeerArchivo();
}
```

```
metodo2(){
  metodo3();
}
```

```
metodo1(){
  metodo2();
}
```

- 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 **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 (FileOpenFailedException 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

Recupera el error

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

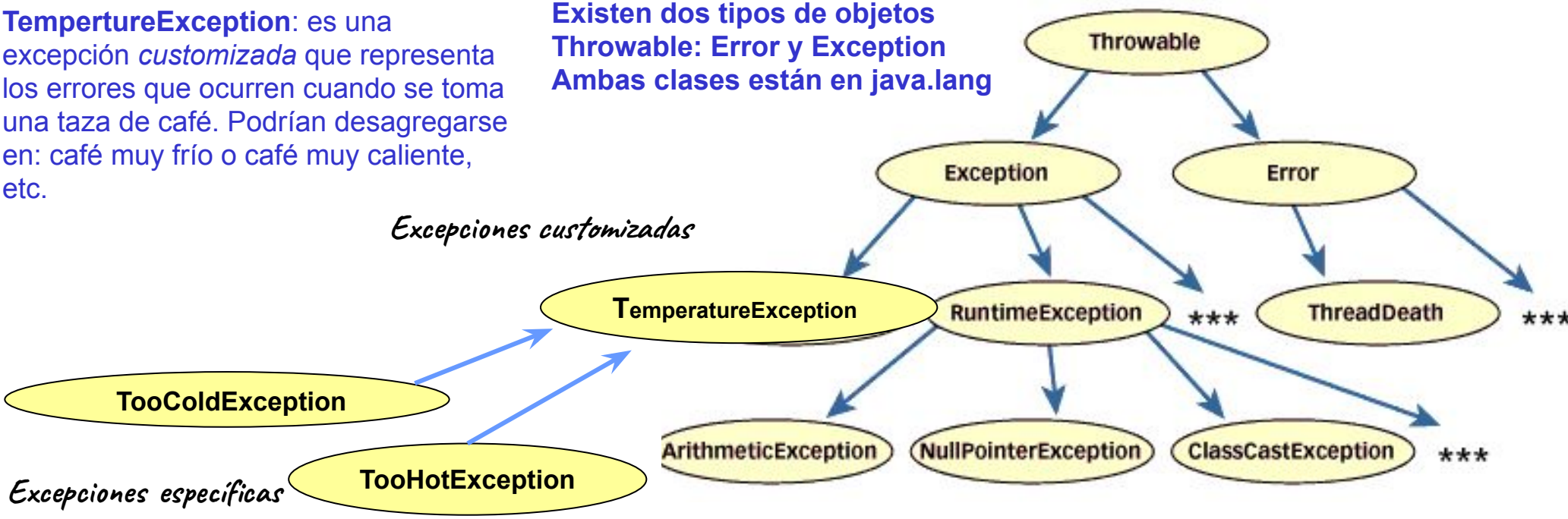
Throwable es la **clase base** de todos los **errores** y **excepciones** en JAVA. Solamente los objetos que son instancias de **Throwable** o de alguna de sus subclases pueden ser disparados por la JVM o por la sentencia **throw**. A su vez el tipo del argumento de la cláusula **catch** solamente puede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

TempertureException: 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



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, ArrayIndexOutOfBounds, 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.

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

```
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 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)

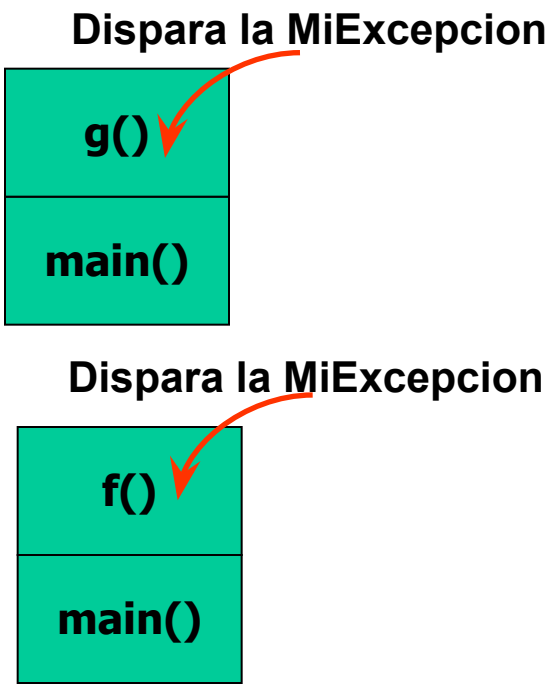
Imprimen el **Throwable** (error) y el ***stack-trace*** del **Throwable**. El ***stack-trace*** muestra la secuencia de métodos invocados que condujo al punto donde se disparó la excepción. La primera versión, imprime en la salida de error estándar y en la segunda y tercera es posible especificar dónde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




```
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 informar al programador las excepciones que podrían disparar los métodos y de esta manera anticipar 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 disparase 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.ou.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 claúsula **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 claúsula **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;
} catch (IOException e) {
   //código del manejador de excepciones
}
```

Re-dispara la excepción capturada

- 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.

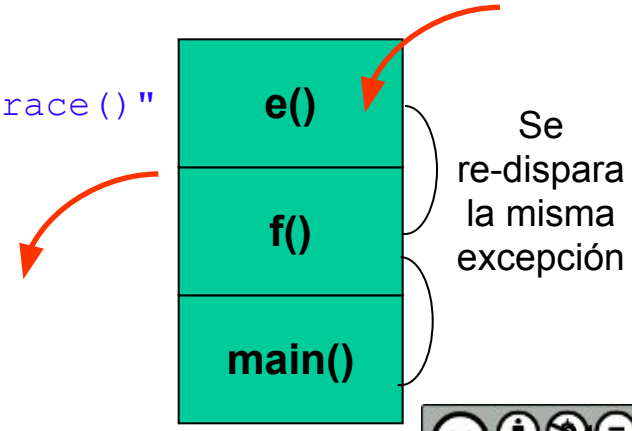
Re-disparar la misma Excepción

```
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)
```

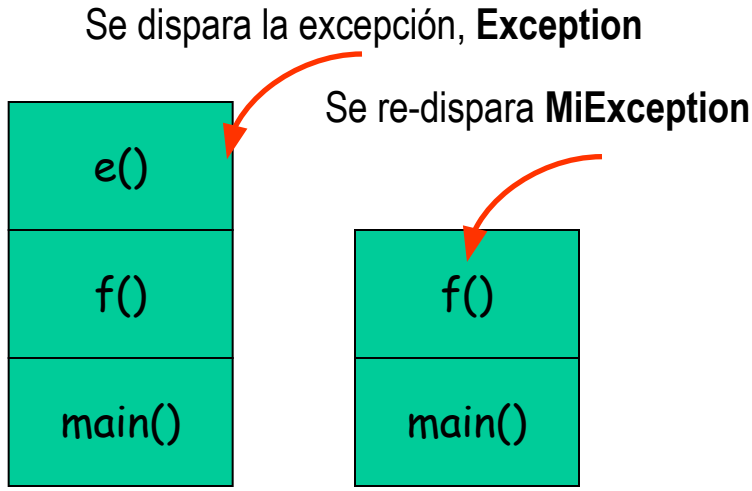
Se dispara la excepción, **Exception**



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)
```



De la excepción MiExcepcion sabe que se originó en f() y se pierde el origen 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();
        x.f();
        x.g();
      } catch (BException e) {}
        catch (CException e) {}
        catch (AException e) {}
   }
}
```

A x=new B();

¿Está bien?

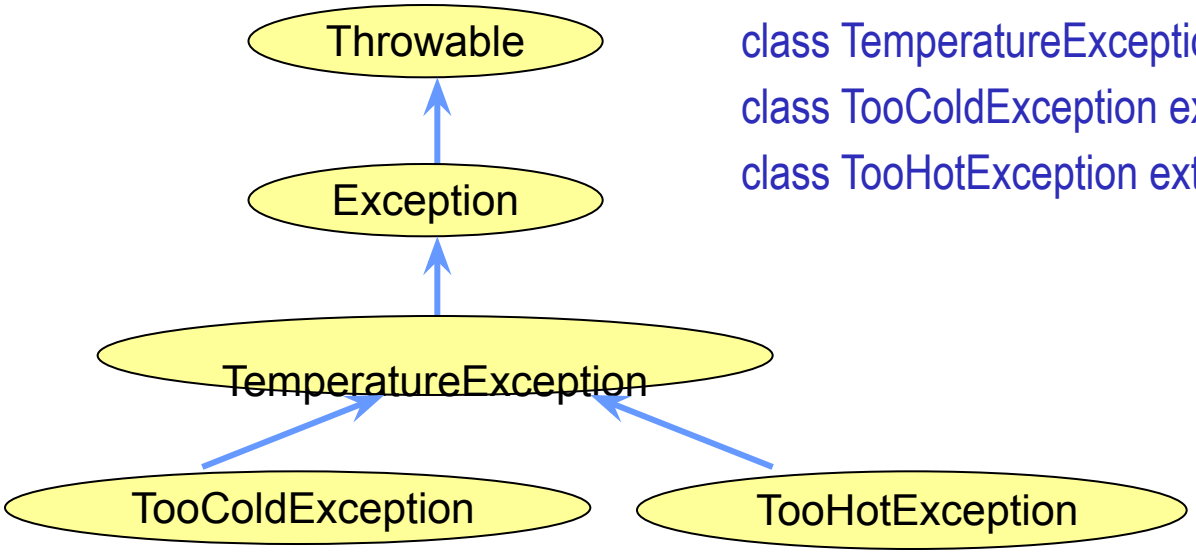
Restricciones en Excepciones

Constructores

- Los constructores no se sobreescriben.
- 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.




```
class TemperatureException extends Exception { }
class TooColdException extends TemperatureException { }
class TooHotException extends TemperatureException { }
```

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) {
       }
      }
 }
}
```

Expresiones LAMBDA

Introducción

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

Originariamente las **clases anónimas** fueron la única manera de **crear objetos función en JAVA**:

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

Interfaces funcionales

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**.

La expresión 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**.

Inferencia de tipos

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)

Cuerpo del Lambda (implementación de la interface)

Operador Flecha: separa los parámetros de la implementación

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);
```

multiplicacion y **suma** son **funciones**, implementan la interface **MathOperation**.

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

servicio es una **función** que implementan la interface **GreetingService**.

Referencias a métodos

En el caso 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.**

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

Expresión Lambda que solo invoca a un método con parámetros:

```
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

comparingInt es un método de clase de la **interface Comparator** que se usa para construir **comparadores personalizados** para **enteros primitivos**.

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

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

Referencias a métodos

```
class Persona {
public int getEdad() {
    return edad;
}

}
```

```
List<Persona> personas = new ArrayList<>();
personas.add(new Persona("Lucía", 30));
personas.add(new Persona("Diego", 25));
personas.add(new Persona("Juana", 35));
personas.sort(Comparator.comparingInt(Persona::getEdad));
```

Referencias a métodos

Crea un comparador que compara objetos de tipo **Persona** en función de su edad.
La lista de personas se ordena utilizando dicho comparador resultando en una lista ordenada por edad.

```
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()** de **Operation** 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 ú 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 catálogo, el conjunto de registros retornados al ejecutar una consulta a una BD, un diccionario, 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** forma parte de 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.**

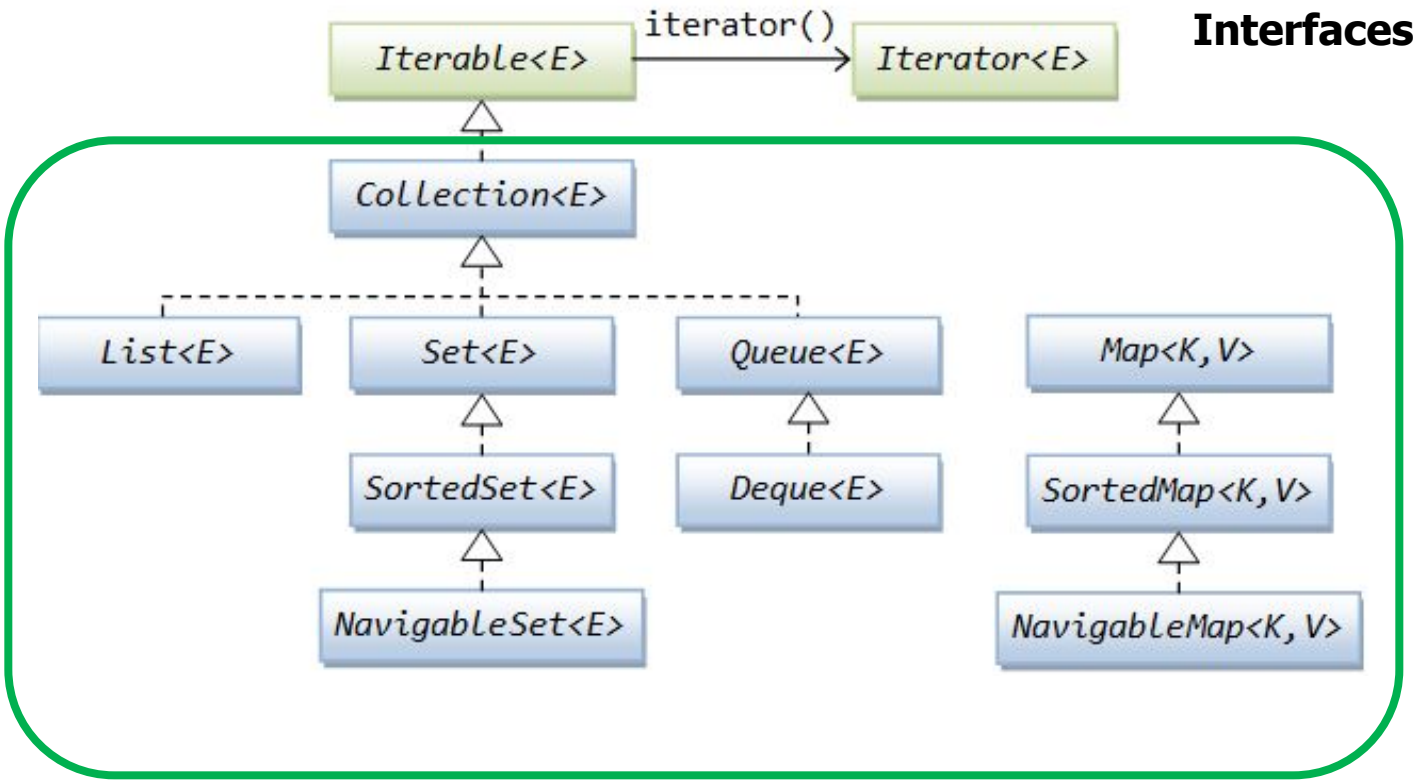
A partir de **JAVA 5** soporta **tipos genéricos.** El **compilador inserta castings** automáticamente y realiza comprobación de tipos cuando se agregan elementos, evitando errores que sólo podían detectarse en ejecución. Como resultado los **programas son más seguros y claros**.

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



Interfaces secundarias

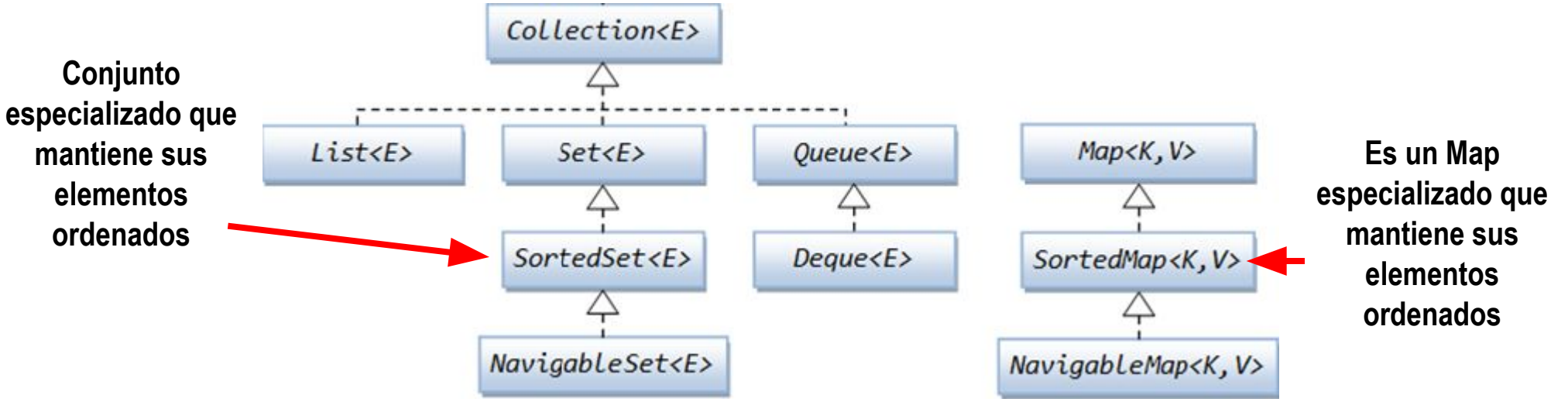
Interfaces centrales

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



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:

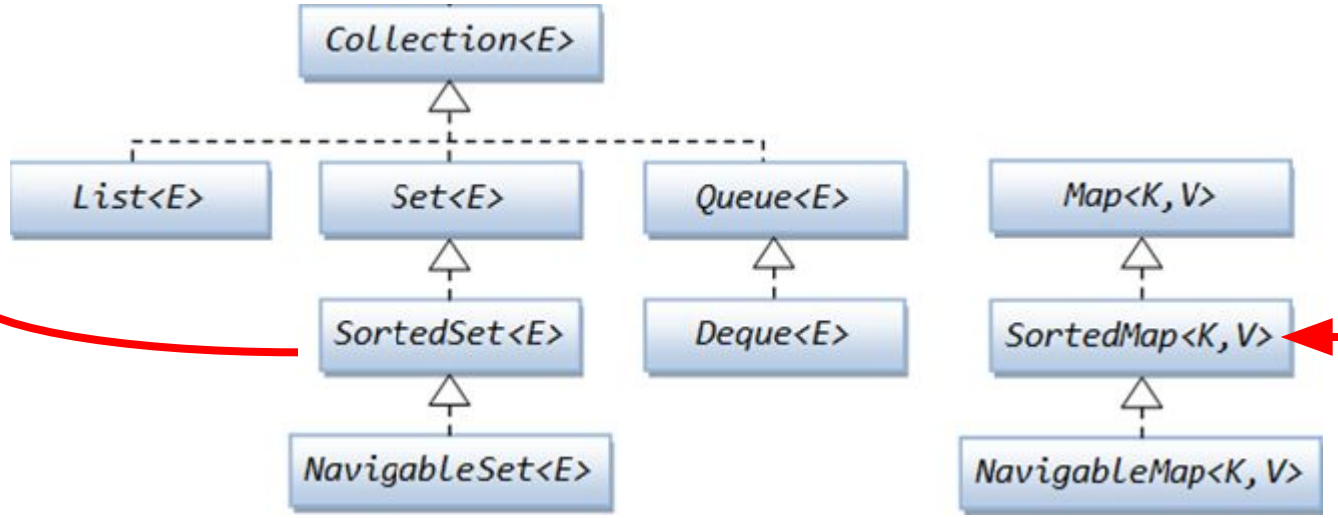
Collection: generaliza el concepto de **grupos de objetos** llamados elementos. Es la raíz de la jerarquía de colecciones. La plataforma Java no provee una implementación directa para la interface Collection pero sí para sus subinterfaces **Set**, **List** y **Queue**.

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 previo a ser procesados. Agrega **operaciones adicionales para inserción, extracción e inspección** de elementos. Típicamente los elementos de una Queue están **ordenados** mediante 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 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 hashing. 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 Colletion

```
public interface Collection<E> extends Iterable<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();
}
```

Es una interface genérica que representa un agrupamiento de objetos de tipo E

Cuando se declara un objeto Collection es posible establecer el tipo de dato que se almacenará en la colección y de esta manera evitar *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 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);
      for(int i: c)
         System.out.println(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 soportada a partir de **JSE 5**

El for-each funciona bien con cualquier cosa que produzca un iterador. En Java 5 la interface Collection extiende Iterable, entonces con cualquier implementación de Set, List y Queue se 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 DemoIterador{
 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<Integer>= 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 Accesso 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 DemoIterador{
  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<Number>= 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);
}
```

```
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.

```
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();
}
```

Implementación de la interface Map

```
public class DemoMap {
public static void main(String[] args) {
   Map<String, Persona> tablaPersona = new HashMap<>();
   Persona[] arregloPer = {
         new Persona("Luis", 27), new Persona("Elena", 20),
         new Persona("Claudia", 30), new Persona("Claudia", 40),
         new Persona("Elena", 22), new Persona("Manuel", 28),
         new Persona("Luis", 44) };
    for (Persona unaPer : arregloPer)
         tablaPersona.put(unaPer.getNombre(), unaPer);
    Collection<Persona> listPer = tablaPersona.values();
     for (Persona unaPer : listPer)
          System.out.println(unaPer);
 }
}
```

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

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

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.

```
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.

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

```
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("Santiago");
    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 DemoOrdenado{
    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 **[Violin, Piano, Saxo, Flauta]**

Implementa **SortedSet**

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

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

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

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

API de Stream

La **API de Streams** se introduce en Java 8 y su propósito es manipular **colecciones de datos** de forma **declarativa** y **funcional**, es decir permite expresar qué se quiere hacer en lugar de codificar una implementación. Es un conjunto de clases e interfaces del paquete java.util.stream

```
List<String> instrumentos = List.of("Piano", "Saxo", "Violin", "Flauta", "Guitarra", "Saxo");
// Mostrar los instrumentos ordenados
instrumentos.stream().sorted()
                     .forEach(System.out::println);
```

Ordena los instrumentos alfabéticamente.
Recorre el stream secuencialmente.

```
int cant_instrumentos_con_a = (int) instrumentos.stream()
                                .filter(instr -> instr.contains("a"))
                                .count();
```

Filtra: selecciona los instrumentos que contienen una letra a.
Cuenta los elementos del stream filtrado.

Un **stream** es una **secuencia** de elementos **creado** a partir de una **fuente** (colección) que soporta operaciones de procesamiento de datos.

Favorece un estilo de código declarativo: se especifica qué se desea hacer, por ej. filtrar, ordenar, contar, recorrer, sin escribir el código de cómo se hace.

API de Stream

```
List<Dish> menu = List.of(
        new Dish(Dish.Type.PASTA, "pasta", true, 350),.......);
```

```
List<Dish> vegetarianDishes = menu.stream()
                .filter(Dish::isVegetarian)
                .collect(toList());
```

Filtra: selecciona las comidas vegetarianas.
Guarda en una lista las comidas filtradas.

```
List<String> lowCaloricDishesName = menu. stream()
                .filter(d -> d.getCalories() < 400)
                .sorted(comparing(Dish::getCalories))
                .map(Dish::getName)
                .collect(toList());
```

1. **Filtra**: selecciona las comidas que contienen menos de 400 calorías.
2. **Ordena** las comidas seleccionadas de acuerdo a las calorías.
3. **Extrae** los nombres de dichas comidas.
4. **Guarda** en una lista nombres de las comidas.

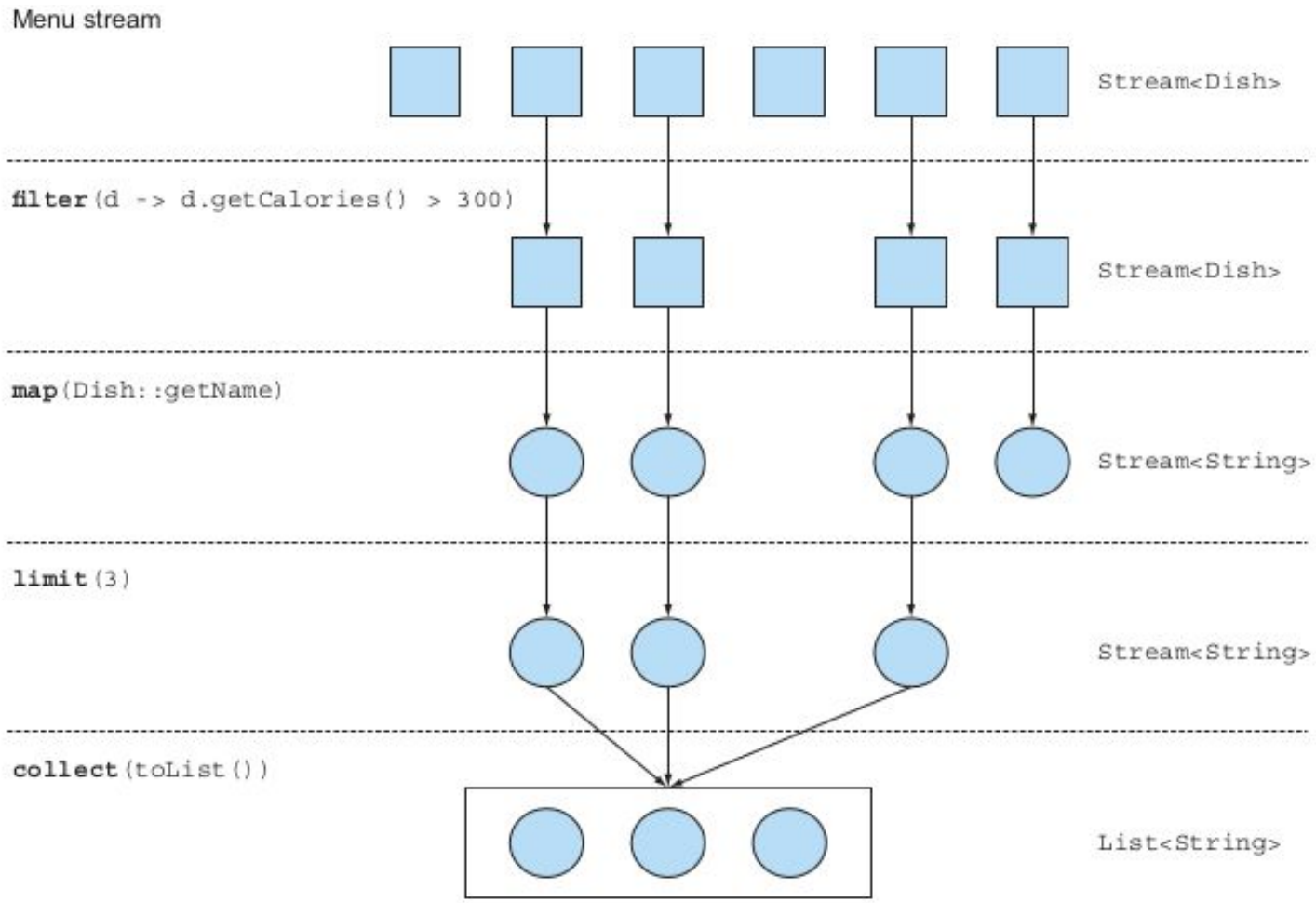
Operaciones intermedias: devuelven otro stream (ej: filter, map, sorted).
Operaciones terminales: consumen el stream y producen un resultado concreto (ej: collect, forEach, reduce). El stream queda cerrado y no se puede reutilizar.
Los **streams** son **inmutables**: no modifican la colección original sobre el que se crea y cada operación intermedia devuelve un nuevo stream.

API de Stream - *Pipelining*

Operaciones de procesamiento de datos

Operaciones intermedias

Operación terminal



Pipelining: las operaciones sobre un stream se encadenan en un pipeline o tubería, no se ejecutan de inmediato..
Evaluación perezosa (Lazy): cuando se invoca una operación terminal se recorre la secuencia y ejecutan todas las operaciones intermedias.

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 utilizan 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**. Ofrecen información descriptiva del programa y que no puede expresarse en código JAVA.
- Las **anotaciones** pueden aplicarse a **paquetes, clases, interfaces, tipos 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 **qué debe hacerse** y las **herramientas lo hacen automáticamente** (producen el código que lo hace).

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

Declaramos que se sobreescribe 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
   }
}
```

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

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

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

@Override

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

La anotación **@Override** en la declaración de un método que sobreescribe 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 genera *warnings*** si **un método sobreescribe 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 sobreescribiendo los métodos en el lugar que deseamos hacerlo.

Anotaciones predefinidas: @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 en futuras versiones.

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

Anotaciones predefinidas: @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, las 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**, obtendremos el siguiente mensaje de advertencia del compilador:

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

Se pueden suprimir varias advertencias:

@SuppressWarnings({"unchecked", "deprecation"})

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"**.

@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

Es una **indicación clara para los programadores** y para las **herramientas de desarrollo** que la **interface** está diseñada para ser utilizada como una **interface funcional**. Ayuda a documentar la intención del diseñador de la interfaz.

Se puede usar más fácilmente con expresiones lambda y referencias a métodos.

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** o **parámetros** que permiten especificar valores para las anotaciones.
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 = "6/10/2024" )
public static void viajarEnElTiempo(Date destino) { }
```

El método viajarEnElTiempo() está anotado con la anotación SolicitudDeMejora

¿Saben quién fue Elisa Bachofen?

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 con un **único elemento** se puede 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 Auditoría 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***.

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

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

En el caso de la anotación **@Test** podríamos pensar en un **testeador "de juguete"** que **analiza** una clase y **ejecuta** todos los métodos anotados 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 Exception> value();
}
```

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

Las **pruebas que tienen éxito** sean las que arrojan **una excepción particular**, por eso esta anotación tiene parámetros.

```
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() { }
}
```

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 Exception> 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);
    }
    }
```

¿La excepción disparada es la esperada?

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

¿Cuál es el resultado?

java RunTest2 anotaciones.Ejemplo2

@ExpceptionTest 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 Exception> [] 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.add(5, null);
}
```

Las meta-anotaciones

La declaración de anotaciones requiere de **meta-anotaciones** 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;

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

3 métodos anotados

Los valores de las anotaciones son expresados entre paréntesis como pares elemento-valor después de la declaración de @CasoDeUso.
Para la anotación del método encriptarPass() se omite el valor del elemento *descripcion*. Se usará el valor definido en @interface CasoDeUso

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.

¿Cuál es la salida?

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

Meta-Anotaciones

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

@Target	Indica a qué elementos se le aplica la anotación. Los valores son los definidos en el enumerativo **ElementType**: **ElementType.ANNOTATION_TYPE:** se aplica solamente a anotaciones **ElementType.TYPE:** se aplica a la declaración de clases, interfaces, anotaciones y enumerativos. **ElementType.PACKAGE:** se aplica a la declaración de paquetes. **ElementType.CONSTRUCTOR:** se aplica a un constructor **ElementType.FIELD:** se aplica a la declaración de un propiedad (incluye constantes **enum**) **ElementType.LOCAL_VARIABLE:** se aplica a la declaración de variables locales **ElementType.METHOD:** se aplica a la declaración de métodos **ElementType.PARAMETER:** se aplica a los parámetros de un método.
@Retention	Indica dónde y cuánto tiempo se mantiene la anotación. Los valores son los definidos en el enumerativo **RetentionPolicy**: **RetentionPolicy. SOURCE**: son retenidas en el código fuente y descartadas por el compilador. No están en los bycodes. Ej.: **@Override**, **@SuppressWarnings** **RetentionPolicy. CLASS**: 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. **RetentionPolicy. RUNTIME**: son retenidas por la JVM en ejecución y pueden ser leídas mediante el mecanismo de reflexión. No son descartadas.
@Documented	Indica que la anotación se incluye en la documentación generada por el **javadoc**. Por defecto, las anotaciones no se incluyen en el javadoc.
@Inherited	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.

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();
}
```

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**

```
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
}
```


MODULE PACKAGE CLASS USE TREE INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class TestEnumHash

java.lang.Object
anotaciones.TestEnumHash

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

Constructor Summary

Constructors

Constructor	Description
TestEnumHash()	

Method Summary

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

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 sobreescriben

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.add(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 framewoks 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

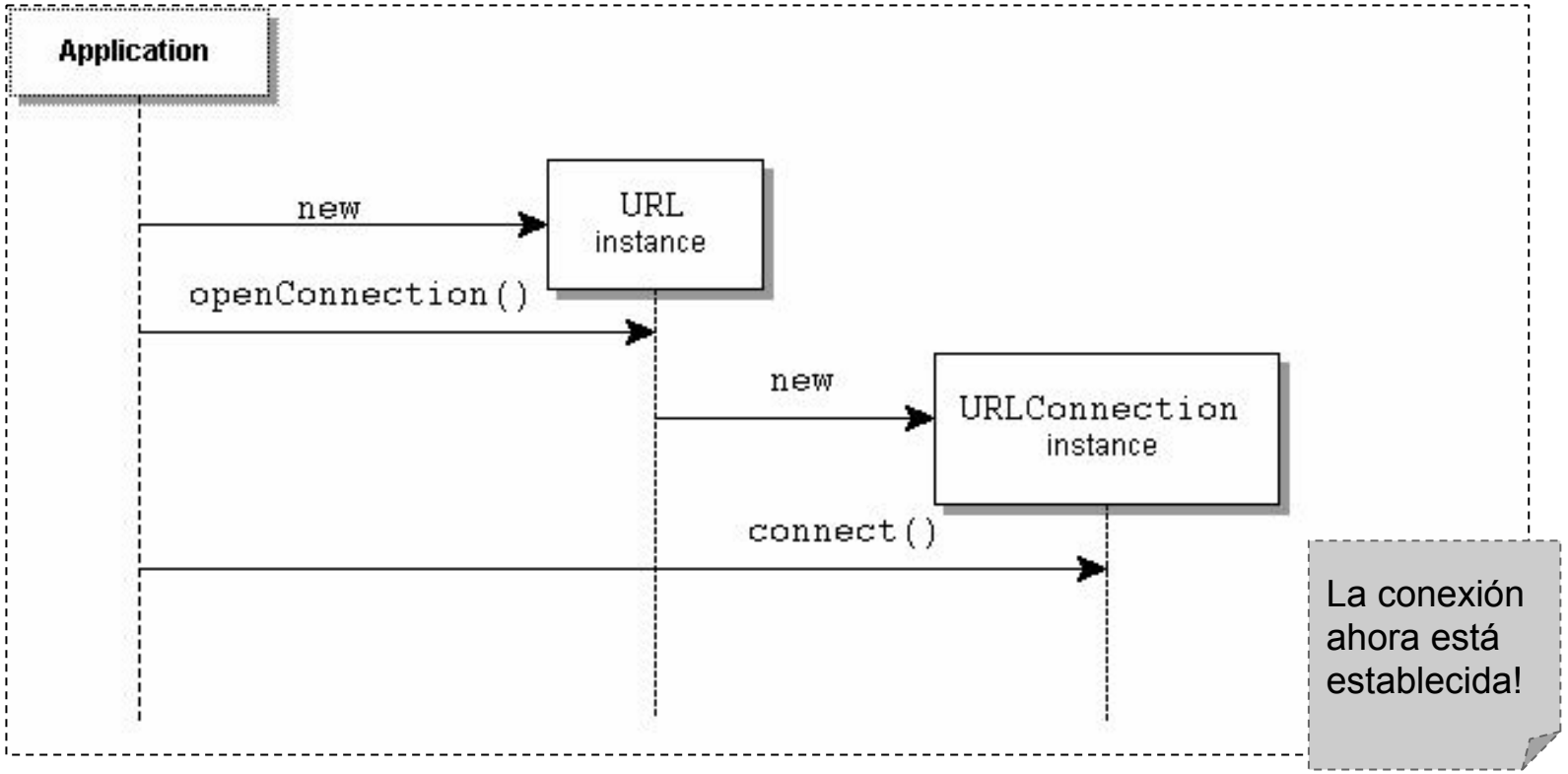
Paquetes de la API

- 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();
```

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

contiene el código de atención del requerimiento

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

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();
    }

}
```

https://docs.oracle.com/en/java/javase/16/docs/api/jdk.httpserver/com/sun/net/httpserver/HttpExchange.html

Si se usa Open JDK

Hay que poner disponible el módulo:

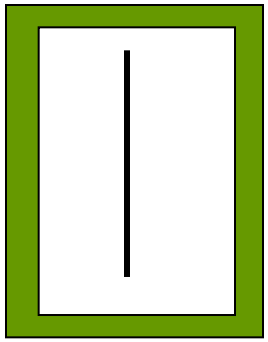
```
module PruebaHttp {

    requires jdk.httpserver;
}
```

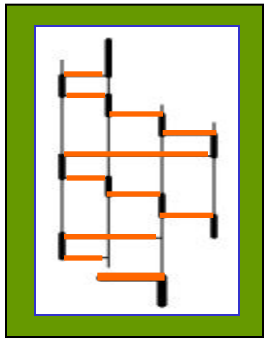
Concurrencia
Threads

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.
- Los 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*

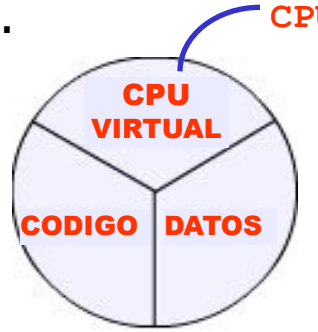
En un programa *multithread*, más de un thread se ejecuta en forma concurrente. El control de ejecución es transferido entre los diferentes threads, cada uno de los cuáles es responsable de distintas tareas.

- 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**.
- Kotlin y por ende JAVA soporta programas **multhreading** a través del lenguaje, de librerías y del sistema de ejecución.

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



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

- La **funcion thread()** proporcionada por el paquete *kotlin.concurrent* que facilita la creación y ejecución de hilos en un programa.
- Hay dos estrategias para usar **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 **funcion thread()** provee el comportamiento genérico de los threads Kotlin: arranque, ejecución, interrupción, asignación de prioridades, etc.
- La **funcion thread()** tiene la siguiente firma:

```
fun thread(start: Boolean = true, isDaemon: Boolean = false, contextClassLoader: ClassLoader? =
null, block: () -> Unit): Thread
```

  - **start**: Indica si el hilo debe comenzar a ejecutarse inmediatamente después de ser creado. Por defecto, es true.
  - **isDaemon**: Indica si el hilo debe ser un hilo daemon y estos no impiden que la JVM se cierre. Por defecto, es false.
  - **contextClassLoader**: Un ClassLoader que se establece como el cargador de clases del nuevo hilo. Puede ser null si se quiere utilizar el cargador de clases actual.
  - **block**: Es un bloque lambda que contiene el código que se ejecutará en el nuevo hilo.
  - La función devuelve una instancia de Thread que referencia al nuevo hilo creado.
- Tanto Kotlin como JAVA son **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, etc.

Ejemplo del uso de thread()

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

```
import kotlin.concurrent.thread

    fun main(args: Array<String>) {

        for(i in 1 .. 5) {
            thread() {
                val name="thread_${i}"
                var contador = 10
                for (j in 1..contador) {
                    println("#${name}:${j}")
                }
        }

    }
}
```

Inicializa el objeto Thread y el thread pasa a estado "vivo"

Un thread termina cuando finaliza el bloque lambda.

Cuando finaliza el for, hay 6 threads ejecutándose en paralelo: el thread que invocó a la **thread()**, en nuestro caso el main thread y los threads que están ejecutando los bloques lambda.

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?

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 temporalmente 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".

```
import java.util.concurrent.TimeUnit
import kotlin.concurrent.thread

fun main(args: Array<String>) {

    for(i in 1 .. 5) {

        thread() {
            val name="thread_${i}"
            var contador = 10
            for (j in 1..contador) {
                println("#${name}:${j}")
                //Antes de JSE 5:
                //Thread.sleep(100)
                //Estilo sugerido:
                TimeUnit.MILLISECONDS.sleep(100)
            }
        }

    }
}
```

- **Thread.sleep()** es un método de clase que detiene la ejecución del hilo actual durante el tiempo especificado en milisegundos, el cual se pasa como parámetro.
- **TimeUnit.MILLISECONDS.sleep()** es utilizado para detener la ejecución del hilo actual, es un método de la clase **TimeUnit**, que es una enumeración utilizada para representar distintas unidades de tiempo, como milisegundos, segundos y minutos.
- 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, aunque podría tardar más en reanudar la ejecución. Además, existe la posibilidad de que se lance una excepción **InterruptedException**.

Métodos de la clase Thread

Join

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. Una vez que el **thread** se completa, el método **join()** retorna inmediatamente.

```
import java.util.concurrent.TimeUnit
import kotlin.concurrent.thread

fun main(args: Array<String>) {
    val t =thread() {
        val name="thread_1"
        var contador = 10
        for (j in 1..contador) {
            println("#${name}:${j}")
            TimeUnit.MILLISECONDS.sleep(100)
         }
    }
    while (t.isAlive){
        println("Esperando...")
        t.join()
    }
    println("Finalizado...")
}
```

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

```
while (t.isAlive()){
    t.join(2000);
    print(".");
}
println(" Finalizado!");
```

Este segmento de código inicia al thread t y cada 2 segundos imprime un "." mientras t 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 indicar 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.**

```
import kotlin.concurrent.thread

    fun main(args: Array<String>) {

        for(i in 1 .. 5) {
            thread() {
                val name="thread_${i}"
                var contador = 10
                for (j in 1..contador) {
                    println("#${name}:${j}")
                    Thread.yield()
                }
        }

    }
}
```

El thread de esta manera realizaría un procesamiento mejor distribuido entre varias tareas del mismo tipo.

Métodos de la clase Thread

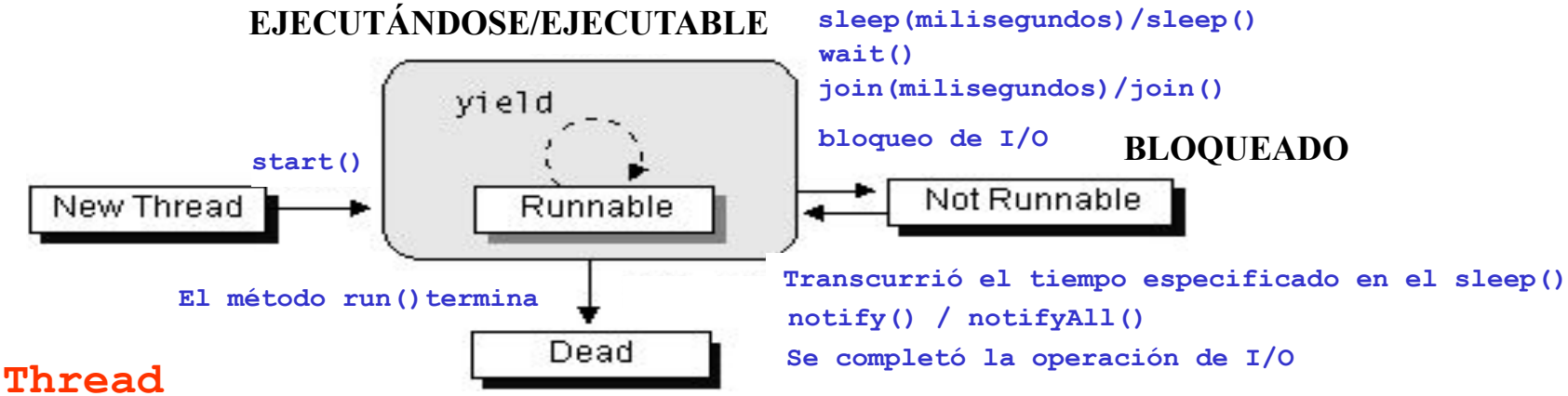
Interrupt

Es un **pedido de interrupción**. El thread que lo recibe se interrumpe a sí 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*.

```
import java.util.concurrent.TimeUnit
import kotlin.concurrent.thread

        fun main(args: Array<String>) {
            val t = thread() {
                for (i in 1..10) {
                    println(i)
                    try {
                        TimeUnit.MILLISECONDS.sleep(4000)
                    } catch (e: InterruptedException) {
                        println("El thread ${Thread.currentThread()} no puede terminar")
                    }
                }
                println("Termino!")
            }
            val startTime = System.currentTimeMillis()
            while (t.isAlive) {
                println("Esperando.......")
                t.join(1000)
                if ((System.currentTimeMillis() - startTime) > 1000 * 60 * 60 && t.isAlive) {
                    println("Cansado de esperar!")
                    t.interrupt()
                    t.join()
                }
            }
            println("Fin!")
        }
```

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  ...
fun main() {
   SwingUtilities.invokeLater  {
       var fin = false
       val label = JLabel("00:00:00")
       val buttonStart = JButton("Arrancar")
       buttonStart.addActionListener  {
           val relojThread =  thread() {

               val formato = DateTimeFormatter.ofPattern( "HH:mm:ss")
               while (!fin) {
                   label. text=LocalTime.now().format(formato)
                   TimeUnit. SECONDS.sleep(1)
               }
               label. text="00:00:00"
           }
       }

       val buttonStop = JButton( "Parar")
       buttonStop.addActionListener  {
           fin=true
       }

       val frame = JFrame("Reloj")
       frame. defaultCloseOperation = JFrame. EXIT_ON_CLOSE
       frame.setSize(300, 200)
       frame. contentPane. layout=FlowLayout()
       frame. contentPane.add(label)
       frame. contentPane.add(buttonStart)
       frame. contentPane.add(buttonStop)
       frame. isVisible = true
   }
}

} // Fin de la clase Reloj
```

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

Crea los recursos para ejecutar el thread, organiza la ejecución del thread y comienza la ejecución del while. **Estado RUNNABLE**

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

Esta asignación hace que la condición de continuación del while 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 Kotlin 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 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 su ejecución 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:

```
val t = thread() { . . . }
```

Con EJECUTORES:

```
val e = unEjecutor
e.execute { . . . }
```

Construye el contexto apropiado para ejecutar objetos **Runnable**

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

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

Ejecutores

Con threads creados por el programador:

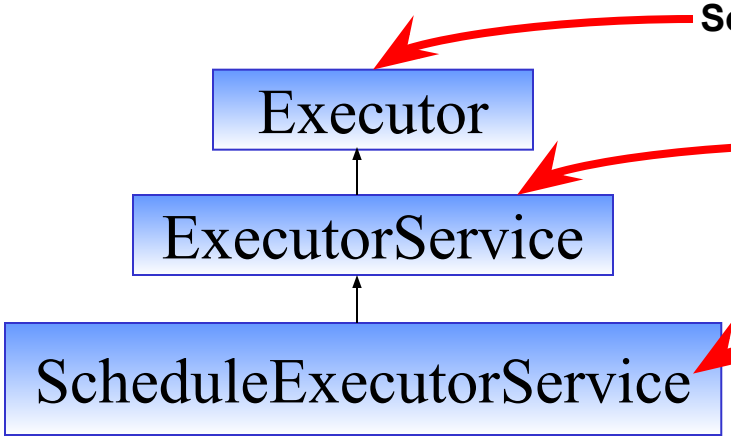
```
val t = thread() { . . . }
```

Con EJECUTORES:

```
val e = unEjecutor
e.execute { . . . }
```

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:



Soporta el arranque de nuevas tareas

Es un Ejecutor con un ciclo de vida de un servicio: shutdown() y shutdownNow() que interrumpe la aceptación de tareas nuevas; el método submit() es similar al execute() y permite conocer el estado de la tarea.

Agrega soporte para la organización de la ejecución de tareas. El método schedule() permite ejecutar una tarea luego de transcurrido un tiempo.

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 Runnables 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 un pool de threads** se puede invocar a los siguientes métodos de clase de la clase **java.util.concurrent.Executors**:

```
val exec = Executors.newFixedThreadPool(int nThreads)
val 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.**

```
val exec = Executors.newCachedThreadPool()
val 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

```
import java.util.concurrent.Executors


fun main(args: Array<String>) {

    val executorService = Executors.newCachedThreadPool()

    for (i in 1..10) {

        executorService.execute{. . .}

    }

    executorService.shutdown()

}
```

Tarea Específica

```
val executorService = Executors.newFixedThreadPool(10)
```

Ejemplo 2

```
import java.util.concurrent.Executors

fun main(args: Array<String>) {

    val executorService = Executors.newSingleThreadExecutor()

    for (i in 1..10) {
        executorService.execute {. . .}
    }

    executorService.shutdown()
}
```

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

Tarea Específica

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 filesystem evitando manejar la sincronización

Compartir Recursos
Condición de Carrera

```
class Counter {

    private var c = 0

    fun increment() {
        c++
    }

    fun decrement() {
        c--
    }

    fun value():Int{
        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 multtithreading 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.

```
class Recurso {
   @Synchronized
   fun f(): Int { }
   @Synchronized
   fun 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

```
class SynchronizedCounter {

    private var c = 0

    @Synchronized
    fun increment() {
        c++
    }

    @Synchronized
    fun decrement() {
        c--
    }

    @Synchronized
    fun value(): Int {
        return c
    }

}
```

Ejemplo Productor/Consumidor

```
import java.util.concurrent.TimeUnit
import kotlin.concurrent.thread

    fun main(args: Array<String>) {
        val bolsa = Bolsa()
        val i = 1
        thread() {
                Thread.currentThread().name = "${i}"
                for (j in 0..9) {
                    bolsa.put(j)
                    println("Productor#${Thread.currentThread().name} escribió: ${j}")
                    TimeUnit.MILLISECONDS.sleep((1L..100L).random())

                }
            }
       thread() {
                Thread.currentThread().name = "${i}"
                for (k in 1..10) {
                    println("Consumidor#${Thread.currentThread().name} leyó: ${bolsa.get()}")
                    TimeUnit.MILLISECONDS.sleep((1L..100L).random())
                }
            }
    }
```

Objeto Compartido

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**

```
class Bolsa {

    private var contenido = 0
    private var disponible = false

    @Synchronized
    fun put(value:Int) {
        // El objeto bolsa fue bloqueada por el Producdor
          ......
        //el objeto bolsa fue desbloqueado por el Producdor
    }

    @Synchronized
    fun get(): Int {
        return contenido
        //El objeto bolsa fue bloqueada por el Consumidor
          ......
        //el objeto bolsa fue desbloqueada por el Consumidor
    }
}
```

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 dentro 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

```
class Bolsa {
   private var contenido = 0
   private var disponible = false

   @Synchronized
   fun get(): Int {
       while (!disponible) {
            (this as java.lang.Object).wait()
       }
       disponible = false
       (this as java.lang.Object).notifyAll()
       return contenido
   }

   @Synchronized
   fun put(value: Int) {
       while (disponible) {
            (this as java.lang.Object).wait()
       }
       contenido=value
       disponible = true
       (this as java.lang.Object).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

Productor/Consumidor

```
fun main(args: Array<String>) {
    . . .
    val bolsa = Bolsa()
    thread() { . . .
               bolsa.put(j)
               println("Productor#${Thread.currentThread().name} escribió: ${j}")
               . . .
             }
    thread() { . . .
               println("Consumidor#${Thread.currentThread().name} leyó: ${bolsa.get()}")
               . . .
         }
}
```

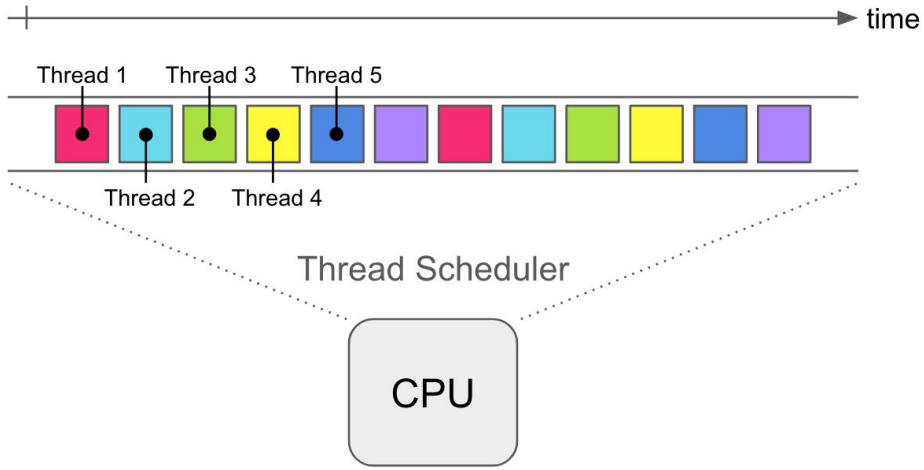
```
Productor #1   escribió: 0
Consumidor #1 leyó: 0
Productor #1 escribió: 1
Consumidor #1 leyó: 1
Productor #1 escribió: 2
Consumidor #1 leyó: 2
Productor #1 escribió: 3
Consumidor #1 leyó: 3
.............
Productor #1 escribió: 9
Consumidor #1 leyó: 9
```

Salida de la función *main*

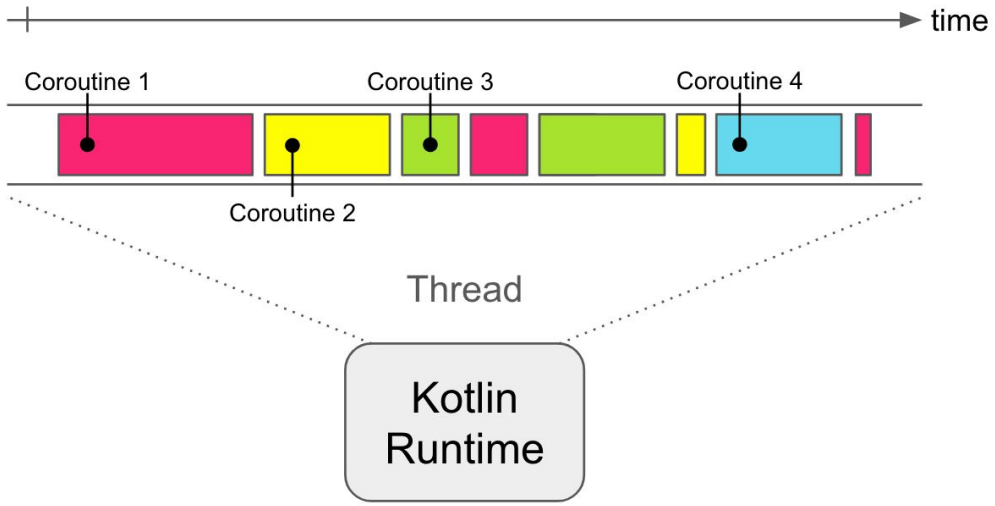
Concurrencia
Corrutinas

Corrutinas

- Como vimos un **thread** es un flujo de control secuencial dentro de un **proceso**.



- Una **corrutina** es un flujo de control secuencial no bloqueante asincrónico dentro un **thread**, siendo el cómputo más *lightweight* dentro de la programación concurrente.



Corrutinas vs Threads

- Las **corrutina** proveen concurrencia pero no paralelismo y pueden suspenderse sin bloquear el **thread.**

- Las **corrutinas** son similares a los **threads** en la programación concurrente tradicional, pero están basadas en multitareas cooperativas.

- Los **threads** son siempre globales, mientras que las **corrutinas** tienen alcance.

- El *Scheduling* de las **corrutinas** es *non-preemptive* y es realizado por el lenguaje o el programador. Los cambios entre diferentes contextos de ejecución lo hacen las propias **corrutinas** en lugar del sistema operativo o la máquina virtual.

- Las **corrutinas** se caracterizan por:
  - Ser muy eficientes.
  - No utilizan *context switching*.
  - No reservan espacio extra en la pila de ejecución.
  - No requieren de sincronización.

Mi primera corrutina en Kotlin

- Las corrutinas forman parte del paquete **kotlinx.coroutines**, por lo que se necesita en el proyecto especificar la siguiente dependencia: *implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.2")*.
- Para iniciar una corrutina se debe usar un constructor (principalmente: **launch**, **runBlocking**, **async**) y pasar las sentencias a ejecutar como un bloque de código.
- Por ejemplo, veamos un programa que imprime "*Hello Word*" con corrutinas.

```
import kotlinx.coroutines.*
fun main() = runBlocking { //inicia una nueva corrutina y bloquea su hilo contenedor
    launch { // dispara una nueva corrutina y continua.
        delay(1000L) // non-blocking delay de 1 segundo (unidad default ms)
        println("World!") // imprime luego del delay
    }
    println("Hello")  // la corrutina principal continúa mientras que la anterior se suspende
}
```

Constructor launch

- El constructor de corrutina **launch** devuelve un **Job** que es el *handle* de la corrutina lanzada.

- El *handle* de una corrutina puede ser usado para esperar explícitamente la finalización o la cancelación la misma.

```
val job = launch { // lanzar una nueva corrutina y
                   //mantiene una referencia a su Job
    delay(1000L)
    println("World!")
}

println("Hello")
job.join() // espera hasta que la corrutina se complete
println("Done")
```

```
Hello
World!
Done
```

```
val job = launch { // lanzar una nueva corrutina y
                   //mantiene una referencia a su Job
    delay(1000L)
    println("World!")
}

println("Hello")
job.cancelAndJoin() // cancela y espera a que se complete
println("main: Now I can quit.")
```

```
Hello
main: Now I can quit.
```

```
job.cancel() // cancela sin espera
```

Un ejemplo más sofisticado

- El constructor **launch** puede tomar un parámetro opcional: **CoroutineContext** que especifica un conjunto de elementos como el **dispatcher**, el **CoroutineName**, etc. que determinan el contexto de la corrutina.
- Otro parámetro opcional del **launch** es el **CoroutineStart** que define las opciones de arranque: **DEFAULT**: inicia inmediatamente, **LAZY**: sólo inicia invocando al **start()**, **ATOMIC**: ejecuta atómicamente o **UNDISPATCHED**: ejecuta inmediatamente hasta el primer punto de suspensión en el thread actual.

```
fun main() = runBlocking {
    val jobs: List<Job> = List(10) { // lista de 10 jobs
    launch(
        Dispatchers.Default +
        CoroutineName("#$it"),
        CoroutineStart.LAZY) // no se inicia al instante
        {
            delay(1000L)
            println("Hello World from $it!")
        }
    }

    jobs.forEach { it.start() } //inicia las 10 corrutinas simultaneamente
}
```

Usa un pool threads compartidos en background.

Contexto de la corrutina

Nombre especificado explícitamente

Define la opción de inicio

```
Hello World from 1!
Hello World from 2!
Hello World from 3!
Hello World from 4!
Hello World from 5!
Hello World from 6!
Hello World from 7!
Hello World from 8!
Hello World from 9!
Hello World from 0!
```

Dispatchers y threads

- El **dispatcher** de la corrutina que determina qué thread o threads se utilizaran para la ejecución, pudiendo confinar la ejecución a un thread específico, despacharla a un pool de threads o dejar que se ejecute sin confinar.
- Alternativas para el uso de **dispatchers**:
  - Usar uno de un grupo de varias implementaciones de **CoroutineDispatcher:**
    - **Dispatchers.Default:** Usa un pool threads compartidos en background.
    - **Dispatchers.Main**: Se limita al Main thread y por lo general es single-threaded .
    - **Dispatchers.IO**: Está diseñado para transferir tareas IO bloqueantes a un grupo compartido de threads.
    - **Dispatchers.Unconfined**: No se limita a ningún thread específico,permite que la corrutina se reanude en cualquier subproceso utilizado por la función de suspensión correspondiente, sin imponer ninguna política de subprocesos específica.
  - Crear un único thread dedicado para ejecutar la corrutina con **newSingleThreadContext**, pero resultando muy caro desde el punto de vista del uso de recursos.
  - Usar ***java.util.concurrent.Executor*** arbitrario que pueda convertirse en un **dispatcher** con la función de extensión **asCoroutineDispatcher**.

Un ejemplo usando distintos Dispatcher

```
Hello World using Dispatchers.Default and working in thread DefaultDispatcher-wo
Hello World using Dispatchers.IO and working in thread DefaultDispatcher-worker-
Hello World using Dispatchers.Unconfined and working in thread main @coroutine#5
Hello World using a java executor (newSingleThreadExecutor) and working in threa
Hello World in the main runBlocking andworking in thread main @coroutine#2
Hello World using a newSingleThreadContext and working in thread MyOwnThread @c
```

```
import kotlinx.coroutines.*
import java.util.concurrent.*

fun main() = runBlocking<Unit> {

    launch { // contexto de la corrutina padre, principal ru
        println("Hello World in the main runBlocking andworking in thread ${Thread.currentThread().name}")
    }

    launch(Dispatchers.Default) { // usa el dispatcher Dispatchers.Default
        println("Hello World using Dispatchers.Default and working in thread ${Thread.currentThread().name}")
    }

    launch(Dispatchers.IO) { // usa el dispatcher Dispatchers.IO
        println("Hello World using Dispatchers.IO and working in thread ${Thread.currentThread().name}")
    }

    launch(Dispatchers.Unconfined) { // usa el dispatcher Dispatchers.Unconfined
        println("Hello World using Dispatchers.Unconfined and working in thread ${Thread.currentThread().name}")
    }

    launch(newSingleThreadContext("MyOwnThread")) { // crea su propio thread
        println("Hello World using a newSingleThreadContext and working in thread ${Thread.currentThread().name}")
    }

    launch(Executors.newSingleThreadExecutor().asCoroutineDispatcher()) { // usa un java.util.concurrent.Executor, newSingleThreadExecutor
        println("Hello World using a java executor (newSingleThreadExecutor) and working in thread ${Thread.currentThread().name}")
    }
}
```

Modificador suspend

- En Kotlin, las corrutinas se usan para implementar **funciones de suspensión** y pueden cambiar contextos sólamente en los **puntos de suspensión**.
- El modificador **suspend** permite lanzar un método en una corrutina.
- Los métodos marcados con **suspend** sólo pueden ser invocados desde una corrutina o desde otro método **suspend**.

```
import kotlinx.coroutines.*
fun main() = runBlocking {
    launch { doWorld() }
    println("Hello")
}

// función suspend
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

Alcance de Corrutinas

- Utilizando el constructor **coroutineScope** se crea un ámbito de corrutina que no finaliza hasta que todos los hijos lanzados hayan terminado.
- Para crear corrutinas de nivel superior, alcance de aplicación, se utiliza el constructor **GlobalScope.launch**.
- Los constructores **runBlocking** y **coroutineScope** difieren principalmente en que el primero bloquea el thread actual, mientras que el segundo sólo suspende; liberando el computo para otros usos.

```
fun main() = runBlocking {
    doWorld()
}

suspend fun doWorld() = coroutineScope {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello")
}
```

```
Hello
World!
```

```
// Ejecuta secuencialmente doWorld seguido de "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}
// Ejecuta concurrentemente ambas secciones
suspend fun doWorld() = coroutineScope {
    launch {
        delay(2000L)
        println("World 2")
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
```

coroutineScope se utiliza p
múltiples operaciones concu

```
Hello
World 1
World 2
Done
```

Constructor async

- El constructor de corrutina **asynch** devuelve un objeto **Deferred<T>**.
- Es necesario invocar a **await()** para obtener el objeto resultado de tipo T del **Deferred<T>**.
- Una función común no puede llamar al **await**, se debe usar **launch** para lanzar una corrutina nueva desde una función.
- Se debe usar el **async** solo cuando se esté dentro de una **corrutina** o de una función **suspend**.

```
// Ejecuta secuencialmente doWorld seguido de "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}
// Ejecuta concurrentemente ambas secciones.
suspend fun doWorld() = coroutineScope {
    val res1 = async {
        delay(2000L)
        "World 2"
    }
   val res2 = async {
        delay(1000L)
       "World 1"
    }
    println("Hello ${res1.await()}, ${res2.await()}")
}
```

```
Hello World 2, World 1
Done
```

Flujo asíncrono `flow`

- **Flow<T>** representa un flujo de datos asíncrono que emite valores secuencialmente.
- Usualmente representan *cold streams*, no se produce ningún valor si no hay nadie que lo colecte.
- Las principales maneras de construir un flujo son:
  - **flowOf**() crea un flujo a partir de un conjunto fijo de valores.
  - **asFlow**() construye un flujo sobre distintos conjuntos de elementos.
  - **flow** {} fabrica un flujo a partir de llamadas secuenciales a la función **emit**.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun simple(): Flow<Int> = flow { // constructor del flow
    for (i in 1..3) {
        delay(100)
        emit(i) // emite el proximo valor
    }
}

fun main() = runBlocking{
    launch {
        for (k in 1..3) {
            println("Hello Wold_$k")
            delay(100)
        }
    }
    simple().collect { value -> println(value) } // Recibe los valores del flujo
}
```

```
Hello Wold_1
1
Hello Wold_2
2
Hello Wold_3
3
```

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking {
    val sum = (1..5).asFlow()
        .map { it * it }
        .reduce { a, b -> a + b }
    println(sum)
}
```

```
55
```

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun main() = runBlocking {
    val sum = flowOf(1, 2, 3, 4, 5)
        .map { it * it }
        .reduce { a, b -> a + b }
    println(sum)
}
```

```
55
```

Canales channel y channelFlow

- Un **canal** es conceptualmente muy similar a cola bloqueante con dos operaciones en suspensión **send** y **receive**.
- **channelFlow** es utilizado para crear flujos destinados a trabajar de forma concurrente y en lugar de utilizar la función **emit** utiliza **send**.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() = runBlocking {

    val channel = Channel<Int>()
    launch {
        for (x in 1..5) channel.send(x * x)
        channel.close() // cierra el canal

    }
    launch {
        for (message in channel) println("Hello World_$message")

    }
    println("Done!")
}
```

```
Done!
Hello World_1
Hello World_4
Hello World_9
Hello World_16
Hello World_25
```

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun canalFlow(): Flow<Int> = channelFlow  {
        launch {for (i in 1..2) send(i) }
        launch {for (i in 3..5) send(i) }
}

fun main() = runBlocking{
      launch {
        for (k in 1..3) {
            println("Hello Wold_$k")
            delay(100)
        }
    }
   canalFlow().collect{value -> println(value)}
}
```

```
Hello Wold_1
1
2
3
4
5
Hello Wold_2
Hello Wold_3
```

Canales vs Flujos

- La primera y más obvia diferencia es que **los canales empiezan a emitir datos inmediatamente** sin importar si algún consumidor recibe los datos.

- Los **flujos** son una buena opción para el **procesamiento secuencial de datos**.

- Los canales se deben cerrar explícitamente una vez finalizado su objetivo para evitar pérdidas.

- Para delimitar las emisiones al ciclo de vida de un determinado componente se recomienda el uso de los flujos y si por el contrario la generación de datos no debe estar sujeta a un ciclo de vida específico, se debe usar canales.