



# Introducción a Java

//Parte 3

IT BOARDING

**BOOTCAMP**



# Índice



**01** Interface Comparator

**02** Expresiones lambda

**03** Factory Method  
(parte 2)

IT BOARDING

**BOOTCAMP**

TEMA

# // La potencia que nos da el uso de interfaces

IT BOARDING

**BOOTCAMP**



“Ya vimos cómo desacoplar clases, ahora veremos cómo desacoplar procesos”.

IT BOARDING

BOOTCAMP



# Algoritmo de ordenamiento

```
public class SortUtil
{
    public static void ordenar(int arr[]){
        for(int j=0; j<arr.length; j++){
            for(int i=0; i<arr.length-1; i++){
                if( arr[i]>arr[i+1] ){
                    int aux=arr[i];
                    arr[i] = arr[i+1];
                    arr[i+1] = aux;
                }
            }
        }
    }
}
```



# Algoritmo Bubble sort, con tipos genéricos

```
public class SortUtil
{
    public static <T> void ordenar(T arr[]){
        for(int j=0; j<arr.length; j++){
            for(int i=0; i<arr.length-1; i++){
                if( arr[i]>arr[i+1] ){
                    int aux=arr[i];
                    arr[i] = arr[i+1];
                    arr[i+1] = aux;
                }
            }
        }
    }
}
```

# Interface Comparator



// Implementamos esta interface para establecer un criterio de precedencia entre dos elementos tipo T

```
package java.lang.util;  
  
public interface Comparator<T>  
{  
    int compare(T a,T b);  
}
```



# Algoritmo Bubble sort, con Comparator

```
public class SortUtil
{
    public static <T> void ordenar(T arr[],Comparator<T> c){
        for(int j=0; j<arr.length; j++){
            for(int i=0; i<arr.length-1; i++){
                if( c.compare(arr[i],arr[i+1])>0 ){
                    int aux=arr[i];
                    arr[i] = arr[i+1];
                    arr[i+1] = aux;
                }
            }
        }
    }
}
```





## Invocando a SortUtil.ordenar, con int[]

// Toda la abstracción se termina cuando llega el momento de utilizar el método genérico y desacoplado.

```
Integer iArr[] = {6,3,5,1,8,2,9,7,4};  
SortUtil.ordenar(iArr,new ComparatorIntAscImpl());
```

```
Integer iArr[] = {6,3,5,1,8,2,9,7,4};  
SortUtil.ordenar(iArr,new ComparatorIntDescImpl());
```

✖

✖

✖

✖

✖

✖

✖

✖

✖

# Interface Comparator, otras implementaciones

// Desarrollaremos dos nuevas implementaciones de Comparator

```
public ComparatorStringAscImpl
implements Comparator<String>
{
    @Override
    int compare(String a,String b)
    {
        return a.compareTo(b);
    }
}
```

```
public ComparatorStringQWEImpl
implements Comparator<String>
{
    @Override
    int compare(String a,String b)
    {
        return a.length()-b.length();
    }
}
```



# Interface Comparator, dos implementaciones

// Implementamos esta interface para establecer un criterio de precedencia entre dos elementos tipo T

```
public ComparatorIntAscImpl
implements Comparator<Integer>
{
    @Override
    int compare(Integer a,Integer b)
    {
        return a-b;
    }
}
```

```
public ComparatorIntDescImpl
implements Comparator<Integer>
{
    @Override
    int compare(Integer a,Integer b)
    {
        return b-a;
    }
}
```



# Invocando a SortUtil.ordenar, con String[]

// Toda la abstracción se termina cuando llega el momento de utilizar el método genérico y desacoplado.

```
String sArr[] = {"Pablo", "Alberto", "Juan", "Carlos"};  
SortUtil.ordenar(sArr, new ComparatorStringAscImpl());
```

```
String sArr[] = {"Pablo", "Alberto", "Juan", "Carlos"};  
SortUtil.ordenar(sArr, new ComparatorStringQUEImpl());
```

✖ ✖

✖ ✖

✖ ✖

# Expresiones lambda 1

// Una expresión lambda permite implementar una interface “on the fly”

```
String sArr[] = {"Pablo", "Alberto", "Juan", "Carlos"};  
Comparator<String> c1 = (a,b)->a.compareTo(b);  
SortUtil.ordenar(sArr,c1);
```

```
String sArr[] = {"Pablo", "Alberto", "Juan", "Carlos"};  
Comparator<String> c2 = (a,b)->a.length()-b.length();  
SortUtil.ordenar(sArr,c2);
```

# Expresiones lambda 2

// Una expresión lambda permite implementar una interface “on the fly”

```
Integer iArr[] = {6,3,5,1,8,2,9,7,4};  
Comparator<Integer> c1 = (a,b)->a-b;  
SortUtil.ordenar(iArr,c1);
```

```
Integer iArr[] = {6,3,5,1,8,2,9,7,4};  
Comparator<Integer> c2 = (a,b)->b-a;  
SortUtil.ordenar(iArr,c2);
```



# Desacoplar todo, incluso el algoritmo

Vimos que podemos abstraernos del tipo usando **tipos genéricos**.

También logramos **abstraernos** del criterio de precedencia mediante implementaciones de **Comparator**.

¿Cómo **podríamos**, incluso, ser independientes del método o algoritmo de ordenamiento en sí **mismo**?

# ¿Dudas? ¿Preguntas?

IT BOARDING

**BOOTCAMP**







# Gracias.

IT BOARDING

**BOOTCAMP**

