

# **Polimorfismo II**

## **interfaces y genéricos**

**UNRN**

Universidad Nacional  
de Río Negro

**XI**  
**V**  
**2025**



---

# Interfaces



# Son una jerarquía paralela a las clases

```
/**  
 * Esta interfaz indica que es posible hacerle  
 * mantenimiento a algo.  
 */  
public interface Mantenible {  
    public int hacerMantenimiento();  
}
```

*Solo definen el prototipo  
del comportamiento*

*¡separado de  
la herencia!*

# Establecen un 'contrato' de comportamiento

# Liberando al polimorfismo

Indicando lo que  
debiera de hacer  
sin el como

# ¡Pero las clases pueden implementar más de una!

```
public class Auto implements Mantenible, Manejable {  
    public int hacerMantenimiento(){  
        Le hagamos mantenimiento  
    }  
    public int acelerar(){  
        El resto del comportamiento del auto  
    }  
}
```

—

# Cuando una clase ‘implementa’ la interfaz **firma el contrato**

# ¡Las clases pueden implementar más de una!

```
public class Computadora implements Mantenible {  
    @Override  
    public int hacerMantenimiento(){  
        Le hagamos mantenimiento  
    }  
    El resto del comportamiento de la computadora  
}
```

# ¿A que le estamos haciendo mantenimiento?

```
public class Taller{  
    public void mantener(Mantenible equipo){  
        trabajo.hacerMantenimiento();  
    }  
}
```

**Se establece la relación  
*hace o puede hacer*  
como una acción potencial**

# Los nombres de las interfaces

terminan en dor,

able o ible

Comparador

Ordenable  
{verbo-sustantivo /  
verbos-adjetivo}

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

**¿Preguntas?**



**Maridan  
bien con...**

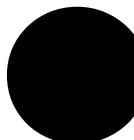
---

# Genéricos



# Dada nuestra implementación

```
public class ArregloConvencional {  
    private int[] arreglo;  
  
    public void modificar(int posicion, int valor);  
    public void insertar(int posicion, int valor);  
    public int extraer(int posicion);  
    public int obtener(int posicion);  
}
```

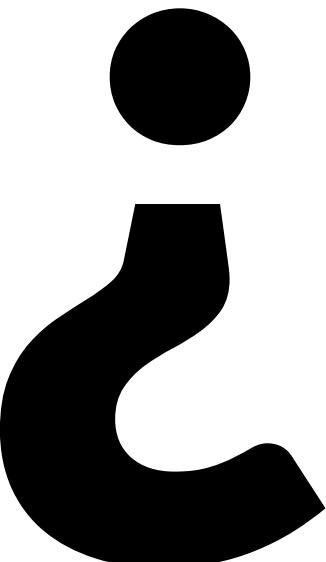
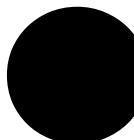


**y si queremos  
guardar  
float's u algún  
Object**

# ¿Pero qué pasa con el código?

```
public class ArregloFloats {  
    private float[] arreglo;  
  
    public void modificar(int posicion, float valor);  
    public void insertar(int posicion, float valor);  
    public float extraer(int posicion);  
    public float obtener(int posicion);  
}
```

*Porque mucho no cambia...*



# duplicamos el código

# Cambiamos a algo mas genérico como base

```
public class ArregloObjetos {  
    private Object[] arreglo;  
    private int largo;  
  
    public void modificar(int posicion, Object valor);  
    public void insertar(int posicion, Object valor);  
    public Object extraer(int posicion);  
    public Object obtener(int posicion);  
}
```

*Porque mucho no cambia...*

# La entrada es directa

```
ArregloObjetos arreglo = new ArregloObjetos(10);
Integer numero = 10;

arreglo.insertar(numero, 0);

Integer salida = (Integer)arreglo.obtener(0);
```

Pero cada salida requiere un cast

# Para bien y para mal

```
ArregloObjetos arreglo = new ArregloObjetos(10);
Integer numero = 10;

Auto movil = new Auto("Nissan");

arreglo.insertar(movil, 0);

Integer salida = (Integer)arreglo.obtener(0);
```



¡Esto puede fallar!

**Podemos insertar de todo**

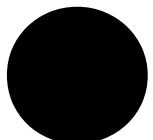
Perdemos el  
‘tipado’

# **¿La herencia podría ayudar?**

# Pero, ¿esto funciona?

```
public class ArregloInteger extends ArregloObjetos {  
    public void modificar(int posicion, Integer valor);  
    public void insertar(int posicion, Integer valor);  
    public Integer extraer(int posicion);  
    public Integer obtener(int posicion);  
}
```

**¿La sobrecarga se puede aplicar a todos los métodos?**



**La  
sobrecarga  
ayudaría**

# No con todos

```
public class ArregloInteger extends ArregloObjetos{  
    public void modificar(int posicion, Integer valor);  
    public void insertar(int posicion, Integer valor);  
    public Integer extraer(int posicion);  
    public Integer obtener(int posicion);  
}
```

**El tipo de retorno, no es tenido en cuenta para la sobrecarga...**

**No resuelve  
el problema  
completamente...**

**Para esto  
entra**

# Genéricos

Polimorfismo paramétrico

# ¡Versión genérica!

```
public class ArregloGenerico<T> {  
    private T[] arreglo;  
  
    public void modificar(int posicion, T valor);  
    public T obtener(int posicion);  
}
```

**Hay un detalle en la construccion que vamos a ver en un ratito**

# ¡Ahora somos libres!

```
ArregloGenerico<Integer> e = new ArregloGenerico<Integer>();  
ArregloGenerico<Vehiculo> a = new ArregloGenerico<Vehiculo>();  
ArregloGenerico<Producto> p = new ArregloGenerico<Producto>();
```

**Nuestro arreglo almacena sólo un tipo a la vez ahora.**

# Dada esta clase Genérica a T

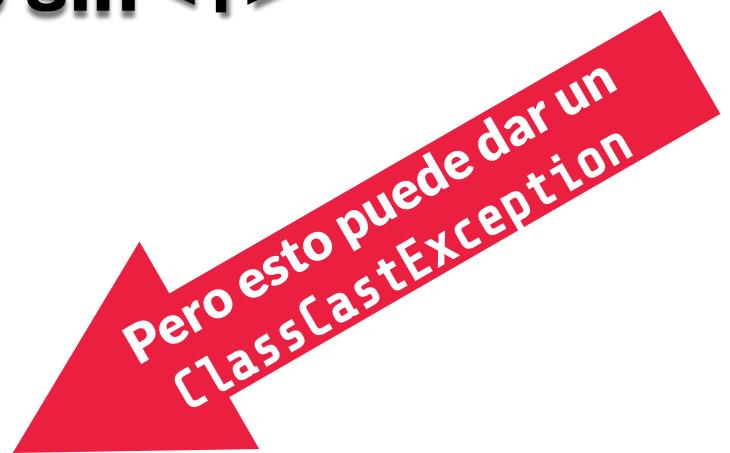
```
public class Caja<T> {  
    private T value;  
  
    public void cargar(T value) {  
        this.value = value;  
    }  
  
    public T descargar() {  
        return value;  
    }  
}
```

```
/**  
 * Representa una caja genérica simple que puede contener un  
 * único valor de cualquier tipo.  
 * Funciona como un contenedor básico para almacenar,  
 * reemplazar y recuperar un objeto.  
 * La caja puede contener `null` si el tipo `T` lo permite.  
 *  
 * @param <T> El tipo del valor que contendrá la caja.  
 */
```

```
Integer numero = 10;  
Caja<Integer> box = new Caja<Integer>();  
box.cargar(numero);  
box.cargar("cadena"); //fallaria
```

# Podemos usar un genérico sin <T>

```
Caja box = new Caja();
box.cargar((Object) Integer(10));
Integer valor = (Integer) box.descargar();
```



Pero esto puede dar un  
ClassCastException

Pero hay un  
problema en la  
construcción

---

# No podemos crear algo de tipo T

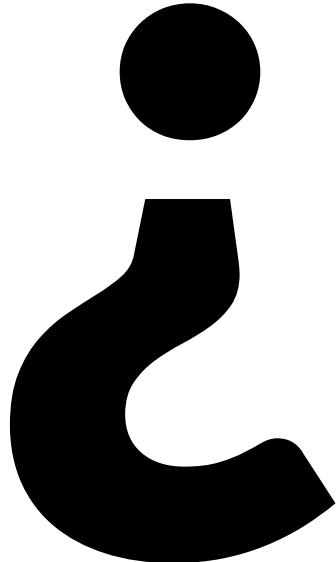
```
public class ArregloGenerico<T> {  
    protected T[] arreglo;  
  
    public ArregloGenerico(int tamaño){  
        this.arreglo = new T[tamaño];  
    }  
    ... resto de los métodos  
}
```

# Pero si podemos hacer casts

```
public class ArregloGenerico<T> {  
    protected T[] arreglo;  
  
    @SuppressWarnings("unchecked")  
    public ArregloGenerico(int tamaño) {  
        this.arreglo = (T[])new Object[tamaño];  
    }  
}
```



Es generico  
despues de todo

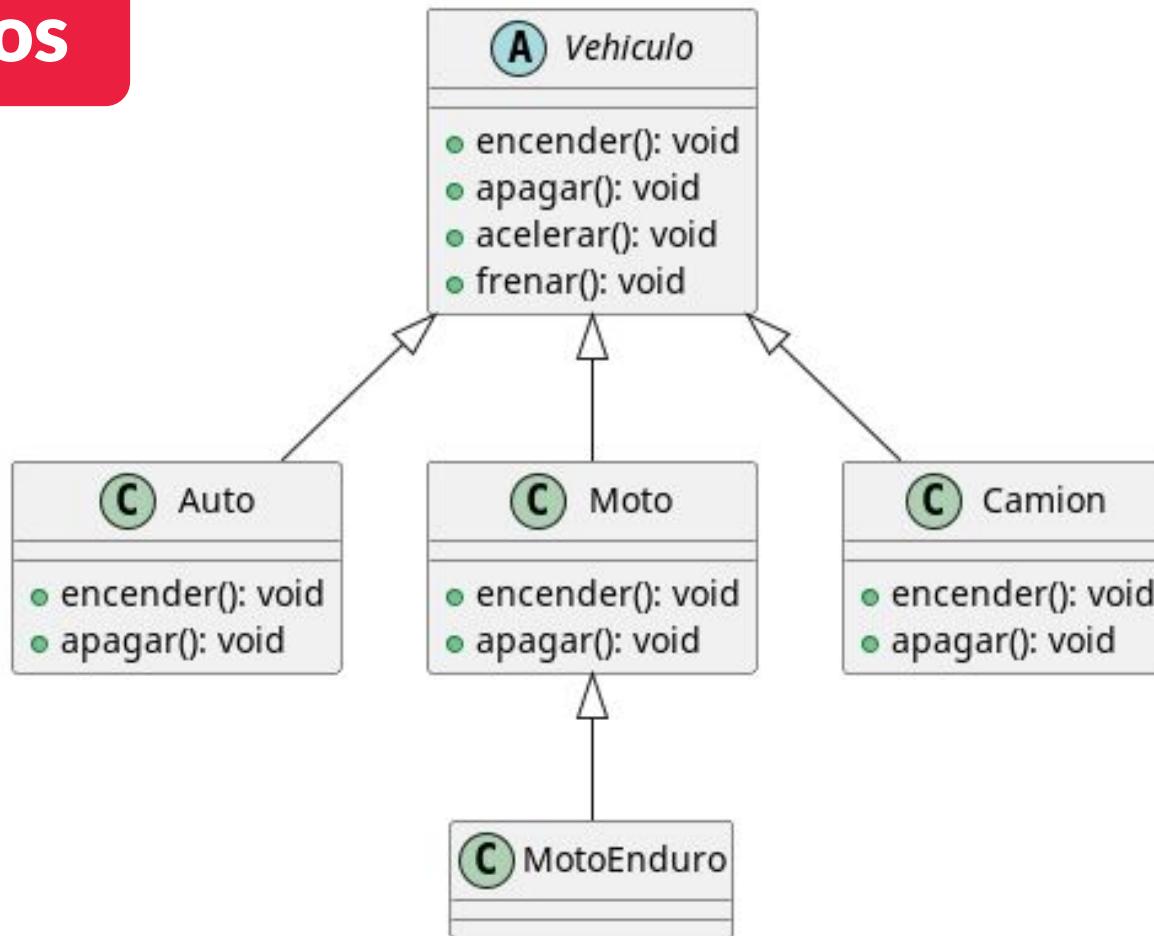


# Por qué



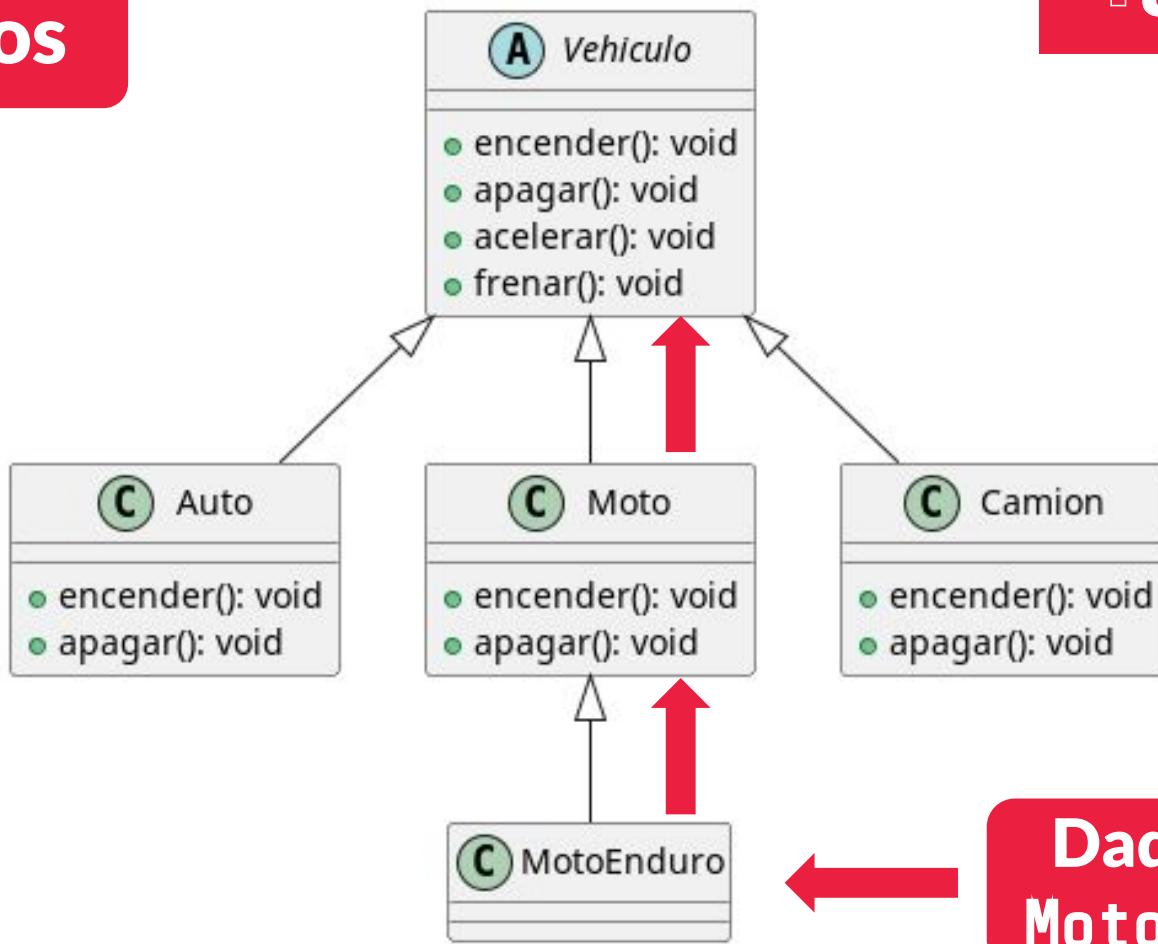
# Refuerzo de casteos

# Vehículos



# Vehículos

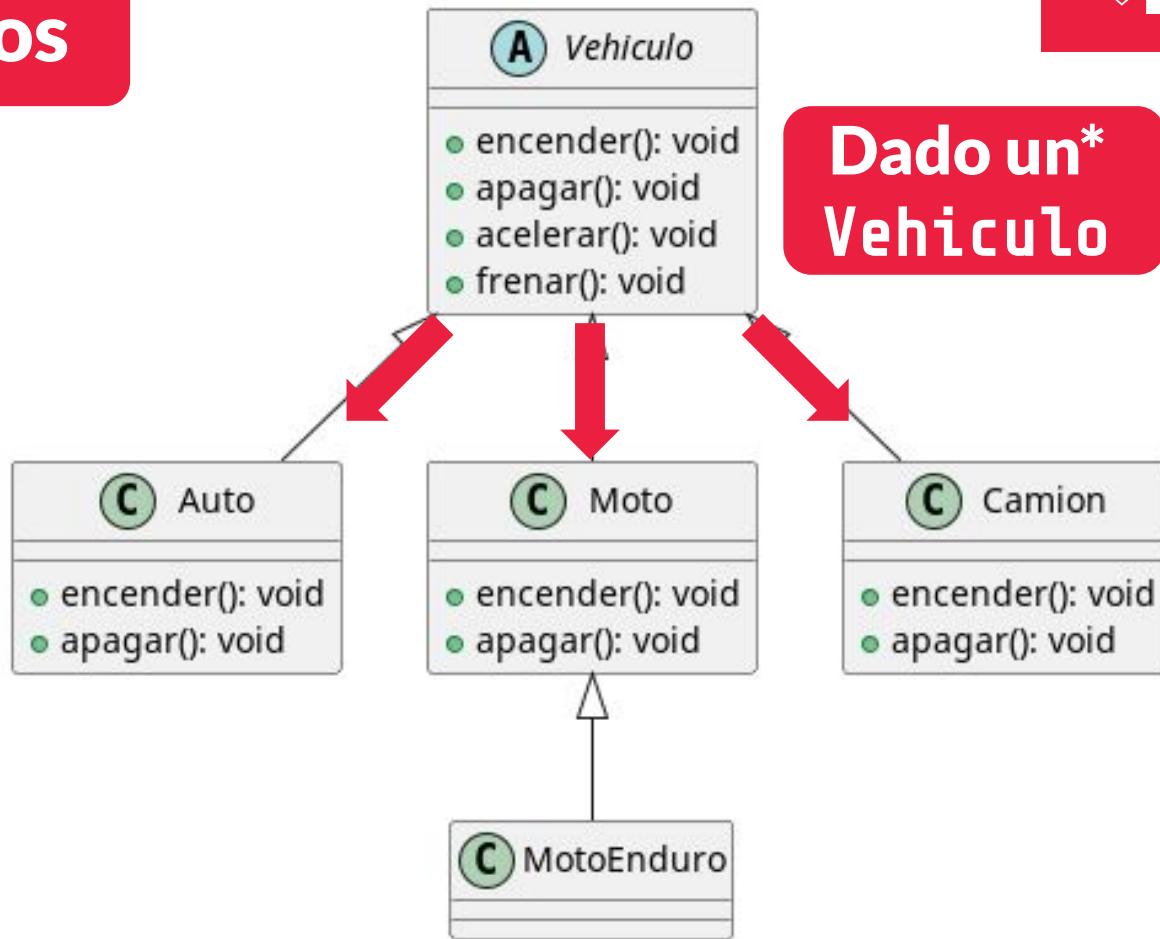
↑ Upcasting



Dada una\*  
MotoEnduro

# Vehículos

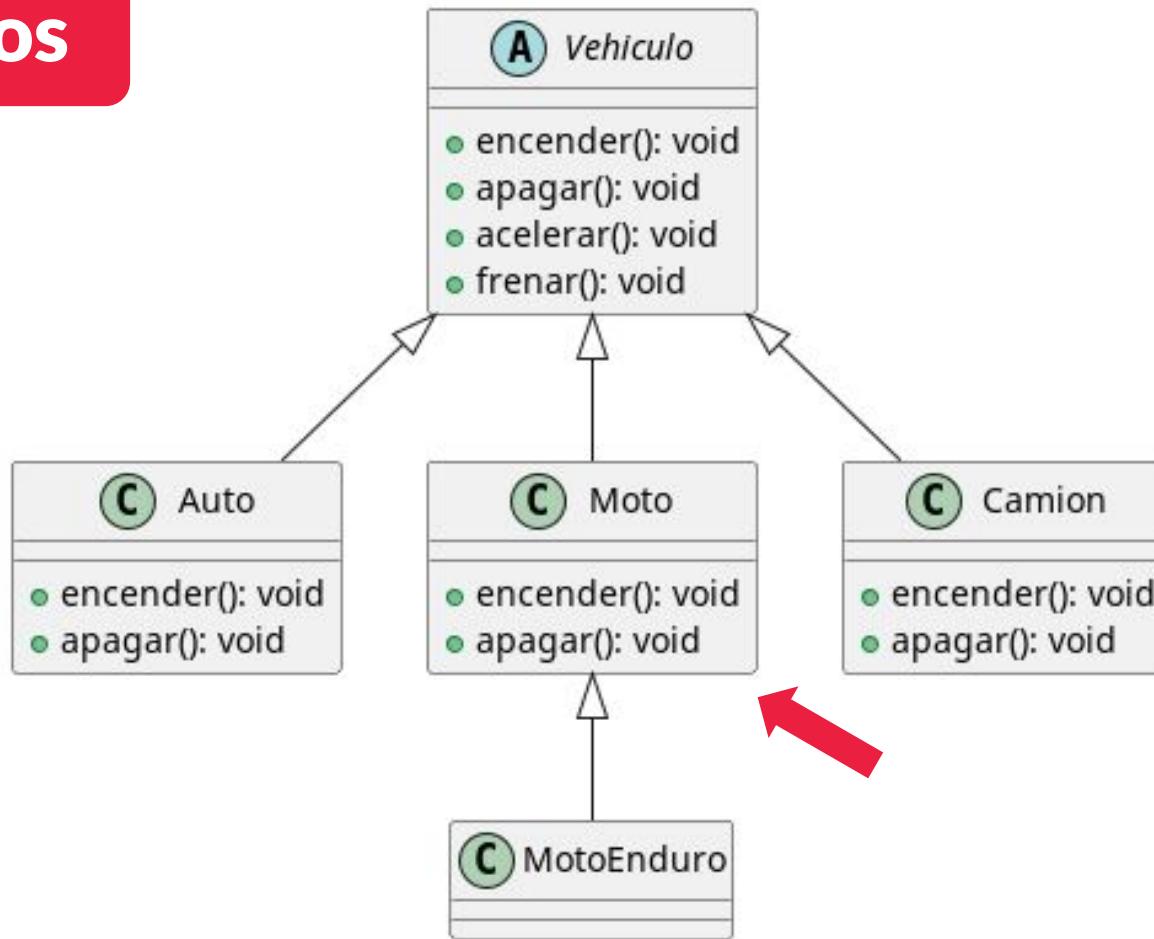
↓ Downcast



# **En definitiva**

**Si la referencia es  
Object, puede  
literalmente ser  
cualquier cosa**

# Vehículos



---

# Type erasure

**Esto significa que  
el tipo de T  
se pierde en tiempo  
de ejecución**

**Y el Type Erasure  
hace que  
un T sea Object**

# El Type Erasure sale de la compatibilidad con código previo a 1.5

- \\_(ツ)\_/ -

Tiene una vuelta  
mas

# ¡Se reemplaza con Object! en tiempo de ejecución

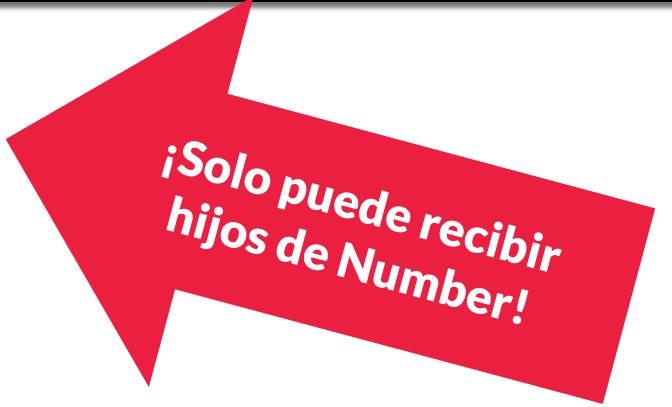
```
public class Caja<Object> {
    private Object value;

    public void cargar(Object value) {
        this.value = value;
    }

    public Object descargar() {
        return value;
    }
}
```

# Pero podemos poner límites

```
public class Caja<T extends Number> {  
    private T value;  
  
    public void cargar(T value) {  
        this.value = value;  
    }  
  
    public T descargar() {  
        return value;  
    }  
}
```



*¡Solo puede recibir  
hijos de Number!*

# **El Type Erasure ahora va a Number**

**(qué es lo más general que  
acepta el T genérico [caja])**

**Que es la razón  
de existir de los  
genéricos**

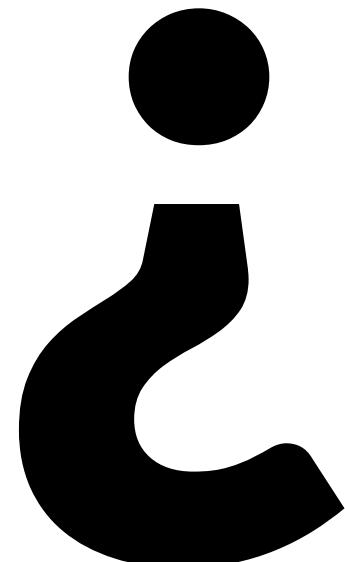
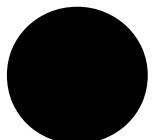
**Hay más sobre  
delimitaciones  
de genéricos**

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

**¿Preguntas?**



Muy lindo lo  
genérico



# Como podemos usarlo

# Interfaces genéricas importantes



# Iterable

Indica que algo se puede  
recorrer linealmente con un  
Iterator

# Tiene esta forma

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

Y con algunos métodos más, pero estos son los importantes

---

# Uso

```
public class ArregloIterable implements Iterable<Integer>
    extends ArregloConvencional{

    Iterator<Integer> iterator(){
        ...//en un ratito vemos como se implementa
    }
}
```

---

# **Nos da la conexión con el `for` mejorado**

```
for (var i : miArregloIterable){  
    Le damos uso a i  
}
```

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

**¿Preguntas?**



# Comparable

Indica que algo se puede ordenar con si mismo

# Comparaciones

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

**Ordenamiento total de si mismo contra otro**

# No hay cast que pueda fallar

```
public Auto implements Comparable<Auto> extends Vehiculo{  
    ...  
    public int compareTo(Auto otro){  
        return Integer.compare(this.numeroSerie, otro.numeroSerie;  
    }  
}
```

¡Porque ya es un Auto!

**El arreglo que recibe los**  
**T debe indicar que estos**  
**son Comparables**

# Para que al ordenar, podamos...

```
class ArregloOrdenado<T extends Comparable<T>>{  
    public void ordenar(){  
        ...  
        if (uno.compareTo(dos) == -1){  
            ... // uno va antes  
        }  
        ...  
    }  
}
```

Un T que es Comparable para T

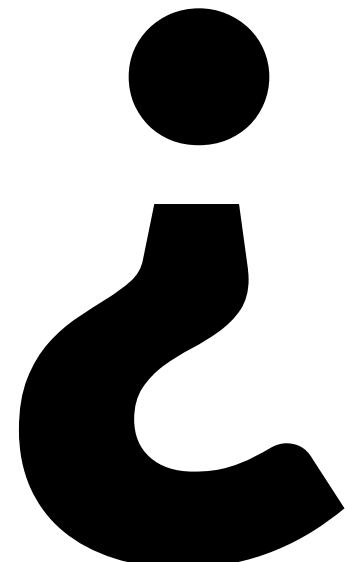
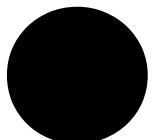
¡Ordenamiento genérico!

No sabemos que  
es, pero lo  
podemos ordenar !



# O con la librería de Java

```
class ArregloOrdenable<T extends Comparable<T>>{  
    void ordenar(){  
        Arrays.sort(arreglo);  
    }  
}
```



# Como hacemos con múltiples criterios de ordenamiento

---

# **Ordenado solo por número de serie (simplificado)**

```
class Auto implements Comparable<Auto> {  
    private int serie;  
    private String marca;  
    private String modelo;  
    int compareTo(Auto otro){  
        return serie - otro.serie;  
    }  
}
```

---

# **Podemos retornar “una forma” de ordenar**

```
class Auto {  
    Comparator<Auto> porMarca()...  
    Comparator<Auto> porModelo()...
```

---

# clases internas



# Comparador

Indica que algo se puede ordenar entre si

# La clase interna privada es utilizable por fuera

```
public class Auto {  
    private String marca;  
  
    private static class PorMarca implements Comparator<Auto> {  
        int compare(Auto uno, Auto otro){  
            return uno.marca.compareTo(dos.marca);  
        }  
    }  
    public Comparator<Auto> porMarca(){  
        return new PorMarca();  
    }  
}
```

Como es de tipo  
~~Comparador~~, que  
es público  
funciona

No podemos instanciar o  
referenciar la implementación  
interna

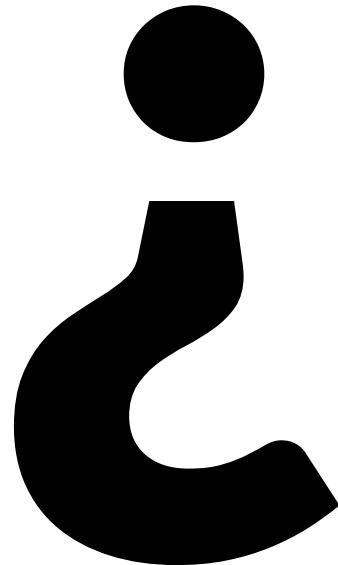
Pero hay una  
sintaxis mas  
cómoda

# clases internas anónimas

# La clase interna anónima es una forma más directa

```
public class Auto {  
    private int serie;  
  
    public Comparator<Auto> porSerie(){  
        return new Comparator<Auto>() {  
            @Override  
            public int compare(Auto uno, Auto dos){  
                return Integer.compare(uno.serie, dos.serie);  
            }  
        }  
    }  
}
```

# Cuando usar uno u otro



# Iterator

**Es la forma para registrar una  
posición en una secuencia y  
obtener valores individuales**

# Comparaciones

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

```
public Iterator<String> iterator() {  
    return new Iterator<String>() {  
        private int i = 0;  
        @Override  
        public boolean hasNext() {  
            return i < elementos.length;  
        }  
    };  
}
```

```
@Override  
public String next() {  
    if (!hasNext()) {  
        throw new NoSuchElementException();  
    }  
    String elemento = elementos[i];  
    i++;  
    return elemento;  
}  
};  
}
```

**Las clases  
internas anónimas  
no tienen  
constructor**

**Si hace falta  
uno, va como  
clase interna**

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

**¿Preguntas?**



**¿Qué es this en una  
instancia de la clase  
interna?**

# ¿Quién es this en compare?

```
public class Auto {  
    private int serie;  
  
    public Comparator<Auto> porSerie(){  
        return new Comparator<Auto>() {  
            @Override  
            public int compare(Auto uno, Auto dos){  
                return Integer.compare(uno.serie, dos.serie);  
            }  
        }  
    }  
}
```

**Podemos desempatar y  
acceder con  
ClaseExterna.this.**

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

**¿Preguntas?**



# ¿y si queremos implementar?

```
public void ordenar();
```

# Comodines genéricos

**Podemos pedir que**  
**ArregloDinamico**  
**solo acepte**  
**Comparable s**

# Podemos agregarlo al arreglo, con un extra.

```
public class ArregloOrdenado<T extends Comparable>{  
    public void modificar(int posicion, T valor);  
    public void insertar(int posicion, T valor);  
    public T extraer(int posicion);  
    public T obtener(int posicion);  
    public void ordenar(){  
        Comparable uno = arreglo[0];  
        Comparable dos = arreglo[1];  
        int orden = dos.compararCon(uno);  
        implementación del ordenamiento  
    }  
}
```

Indicando que sea lo que sea que guardemos, tiene que ser

Comparable

# Podemos agregarlo al arreglo, con un extra.

```
public class ArregloOrdenado<T extends Comparable<T>>{  
    public void modificar(int posicion, T valor);  
    public void insertar(int posicion, T valor);  
    public T extraer(int posicion);  
    public T obtener(int posicion);  
    public void ordenar(){  
        T uno = arreglo[posicion];  
        T dos = arreglo[posicion+1];  
        int orden = dos.compararCon(uno);  
    }  
}
```

Indicando que sea lo que sea que guardemos, tiene que ser

Comparable

Pero también,  
que sea del  
mismo **Tipo**

¿**y**  
**Comparato**?

# ? super T establece un mínimo

```
public class ArregloOrdenado<T extends Comparable<T>>{  
    resto de la clase  
    public void ordenar(Comparator<? super T> comparador){  
        ...implementación  
    }  
}
```

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

**¿Preguntas?**



Y los tipos  
primitivos

# **autoboxing**

## wrapper automático

---

# Dada esta caja genérica

```
public class Caja<T> {  
    private T value;  
  
    public void cargar(T value) {  
        this.value = value;  
    }  
  
    public T descargar() {  
        return value;  
    }  
}
```

# ¡Entra y sale derecho!

```
Caja<Integer> cajaEntero = new Caja<>();
```

```
int entero = 42;  
cajaEntero.cargar(intero);
```

```
int valorRecuperado = cajaEntero.descargar();  
System.out.println("Valor recuperado: " + valorRecuperado);
```



# Como cualquier otro valor.

```
Caja<Double> cajaDoble = new Caja<>();
```

```
double numero = 42.4;  
cajaDoble.cargar(numero);
```

```
int valorRecuperado = cajaDoble.descargar();  
System.out.println("Valor recuperado: " + valorRecuperado);
```



A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

**¿Preguntas?**



# Dudas del TP7



Abran  
hilo

# TP8

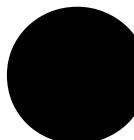
## Arreglos III

# Sobre el TP8

# ArregloDinamico

Que el arreglo pueda cambiar de tamaño.

**Cuál es la  
complejidad al  
cambiar de  
tamaño el  
arreglo**



**Qué métodos  
cambian el  
tamaño  
siempre**

---

# **¿Cuál es su complejidad?**

insertar  
extraer/borrar

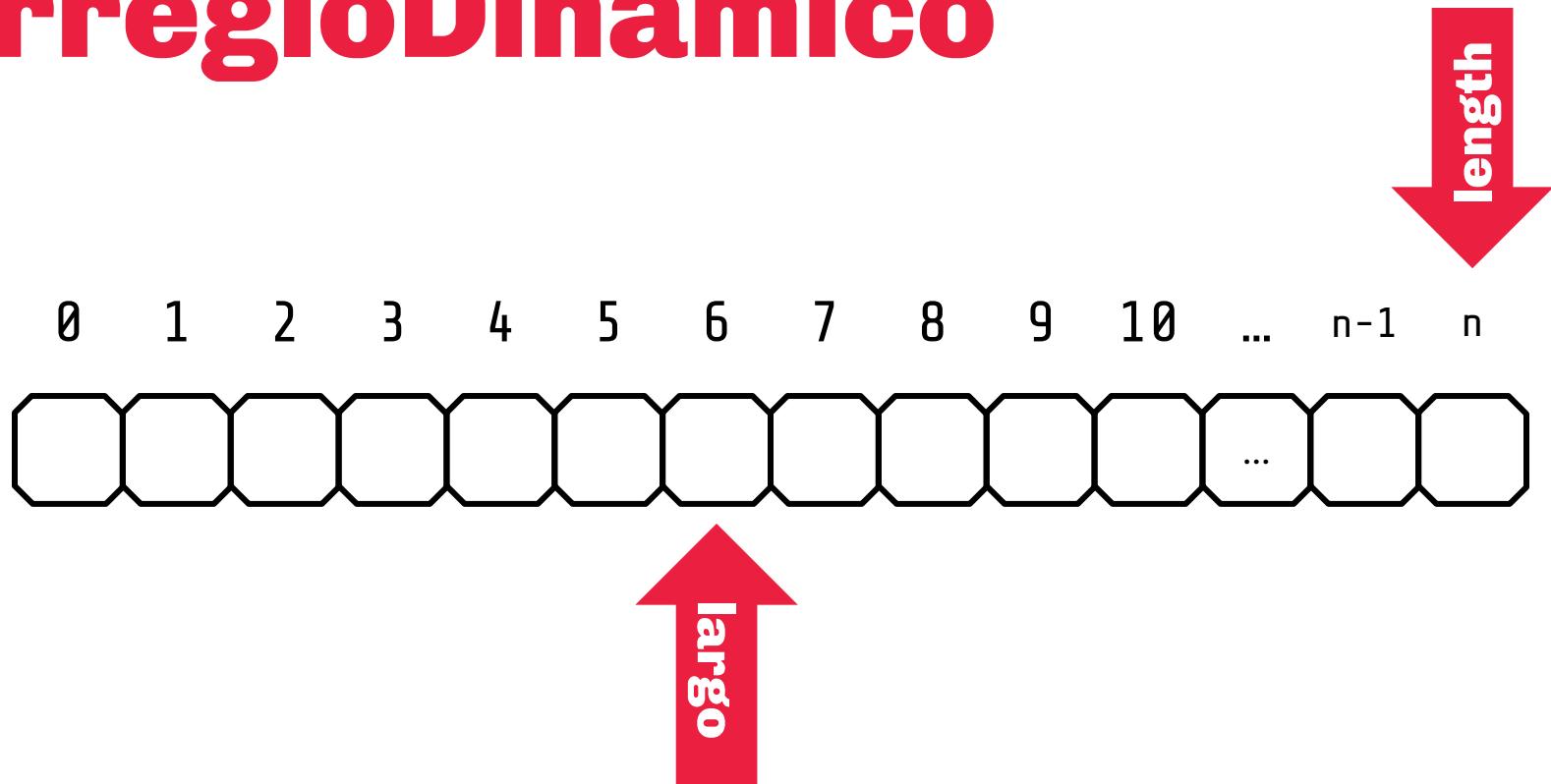
$O(n)$ 

**¡Hay que copiar el contenido en un  
arreglo nuevo!**

Como lo  
podemos  
mejorar

Se desea crear un arreglo que **no necesite** cambiar el arreglo interno en todas las operaciones que requieran cambio de tamaño.

# ArregloDinamico



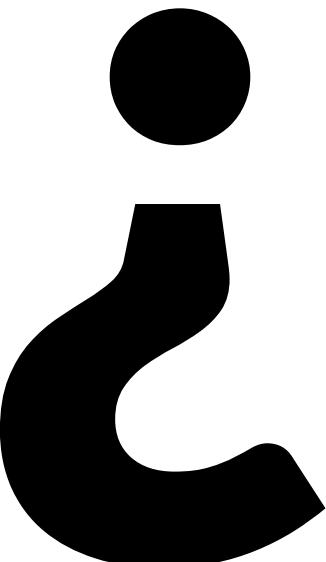
# ArregloDinamico

```
public class ArregloDinamico extends Arreglo {  
    private int largo;  
    public ArregloMejorado(int largo){  
        super(largo * 2);  
        this.largo = largo;  
    }  
  
    public void ampliar(){  
        this.arreglo = Arrays.copyOf(this.arreglo, this.arreglo.length * 2);  
    }  
  
    public int largo(){  
        return largo;  
    }  
}
```

# O(**k**)

**Siendo k la cantidad de veces que es necesario ampliar el arreglo**

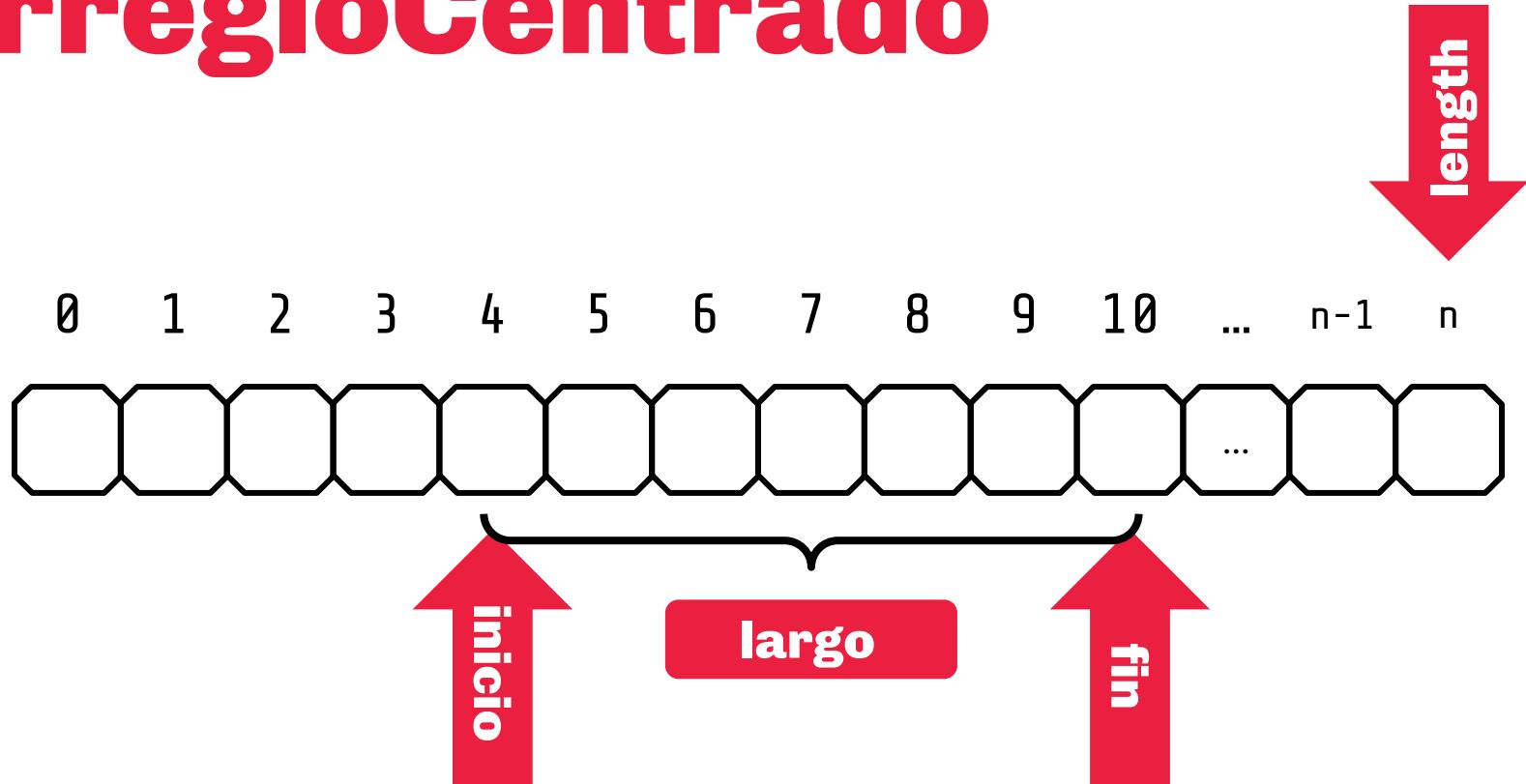
Como lo  
podemos  
mejorar

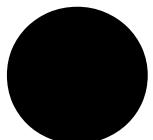


**¿Y si tenemos  
que insertar  
al principio?**

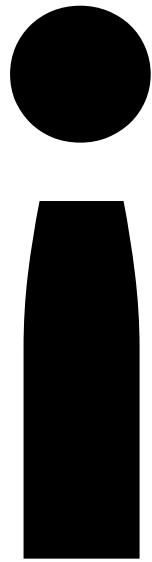


# ArregloCentrado

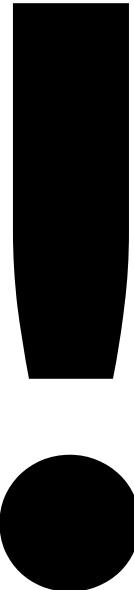




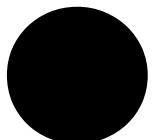
**El  
comportamiento  
cambia**



Nope!



**Por lo que los  
tests debieran  
de funcionar**



**Y si  
necesitamos  
que guarde  
cualquier cosa**

# ArregloGenerico

Cuando queremos guardar otra cosa que no sea un int

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

**¿Preguntas?**



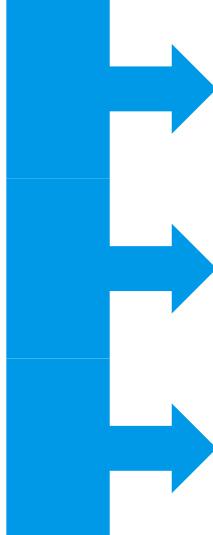
Como lo  
podemos  
mejorar

# ArregloFragmentado

Usando ArregloGenerico, podemos hacer  
¡un arreglo aún más avanzado!

## ArregloFragmentado

ArregloGeneric  
o



ArregloDinamic  
o

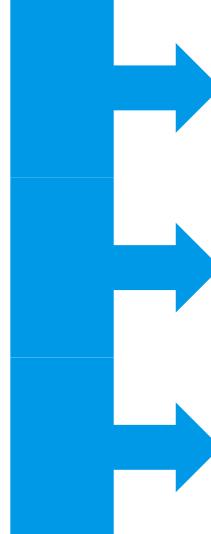
ArregloDinamic  
o

ArregloDinamic  
o



## ArregloFragmentado

ArregloGeneric  
o



ArregloCentrad  
o

ArregloCentrad  
o

ArregloCentrad  
o



Desde 'afuera'  
no hay cambios

:-D

# Hasta la próxima

