

Polimorfismo III

genéricos

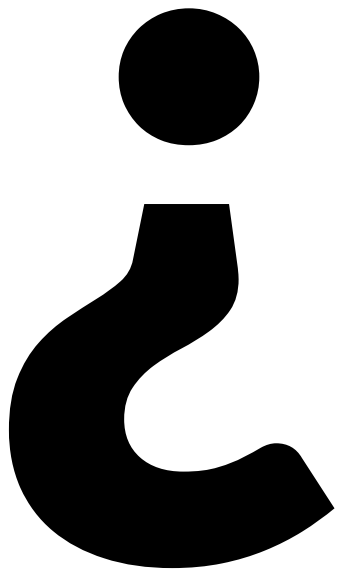
UNRN

Universidad Nacional
de Río Negro

XV

2024





Dudas del TP8





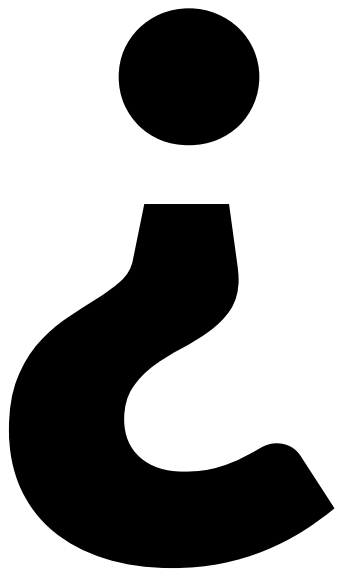
¿Preguntas?



**Abran
hilo**

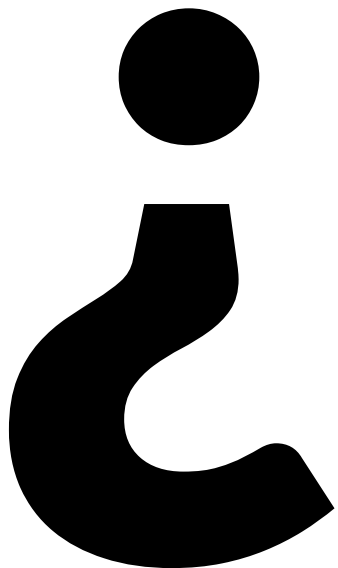
Mejoras para el





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





**Qué métodos
cambian el
tamaño
siempre**



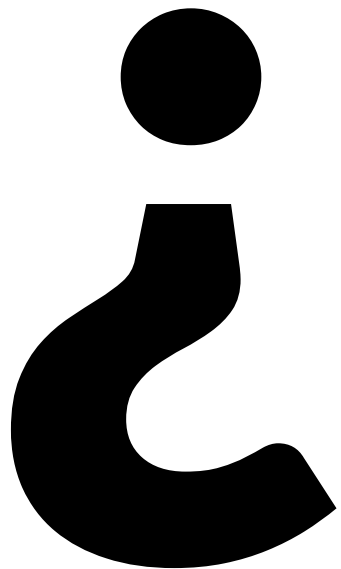
¿Cual es su complejidad?

insertar

extraer/borrar

$O(n)$

¡Hay que copiar el contenido en un arreglo nuevo!

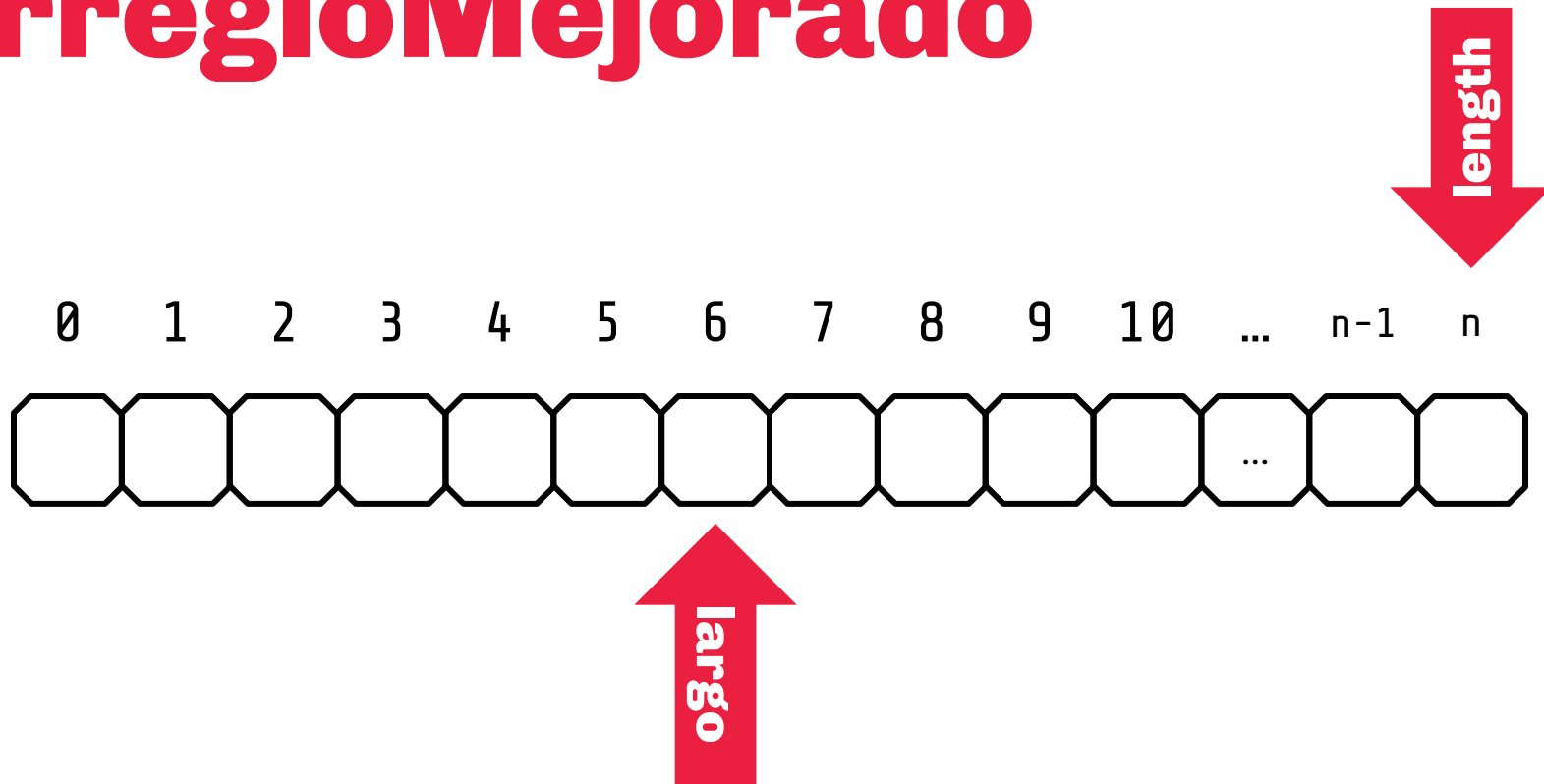


**Como lo
podemos
mejorar**



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

ArregloMejorado

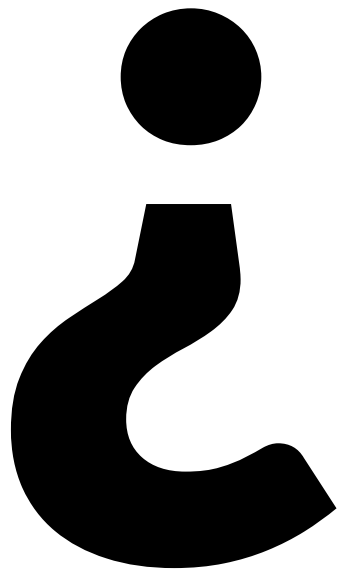


ArregloMejorado

```
public class ArregloMejorado 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;  
    }  
}
```

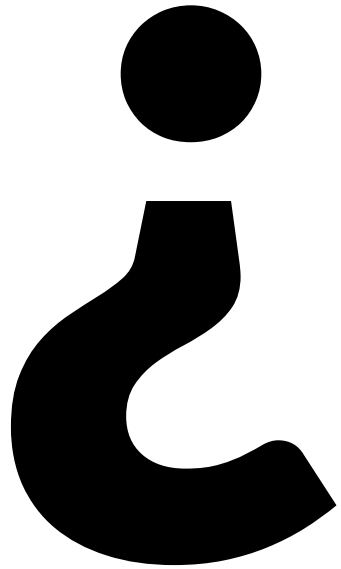
$O(k)$

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



**Como lo
podemos
mejorar**

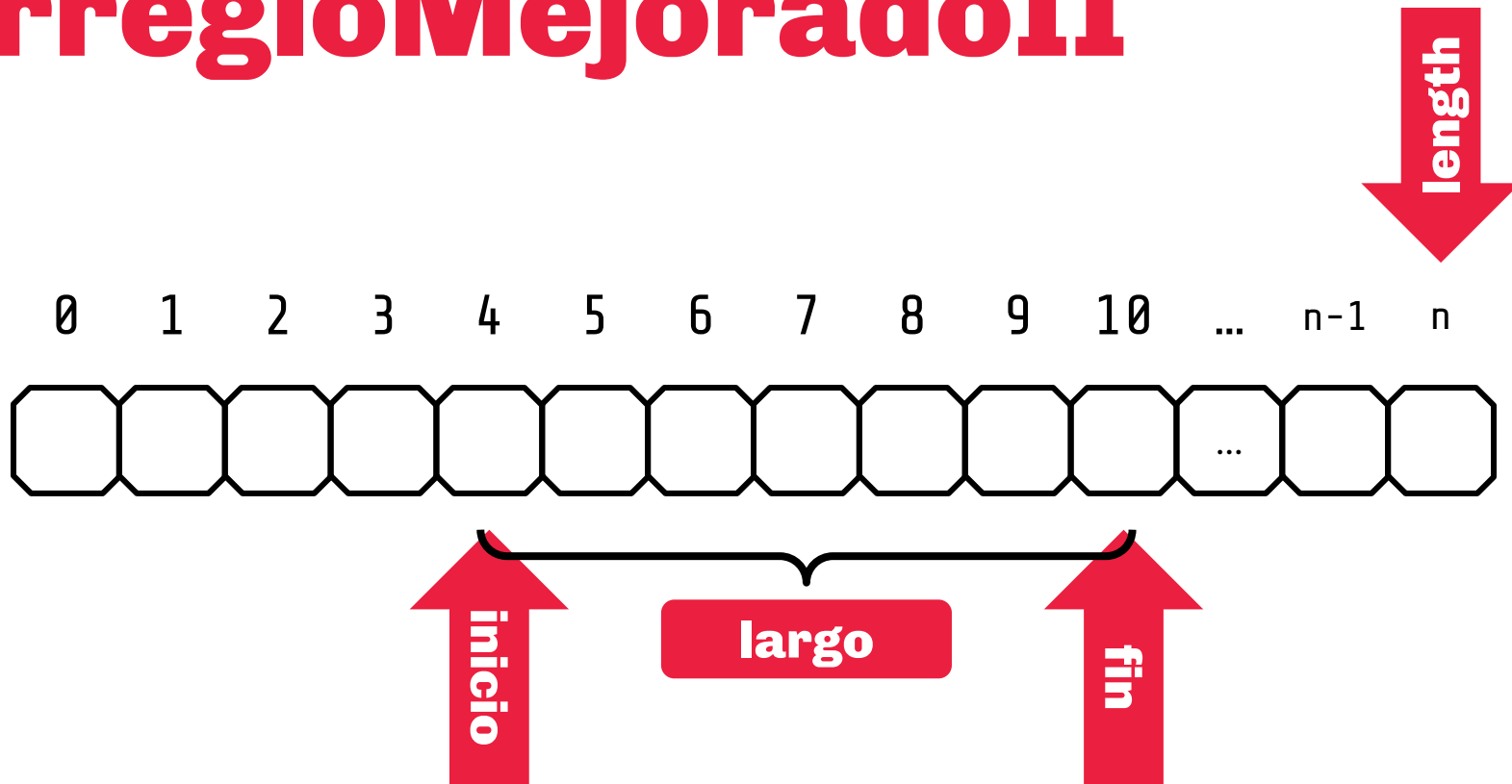


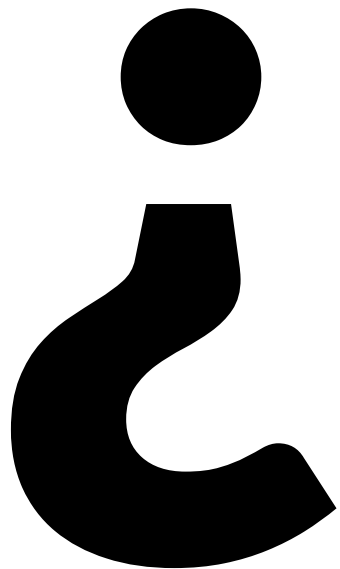


**¿Y si tenemos
que insertar
al principio?**



ArregloMejoradoII





El comportamiento cambia



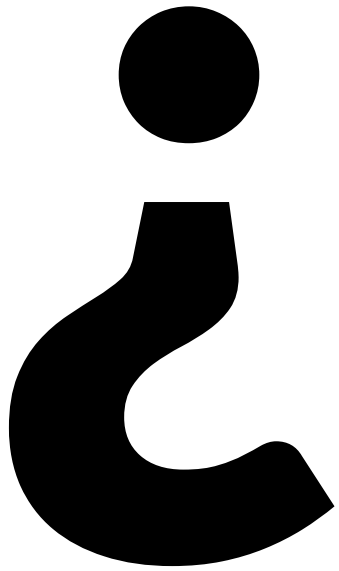
i Nope !

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



¿Preguntas?

{ pero... }



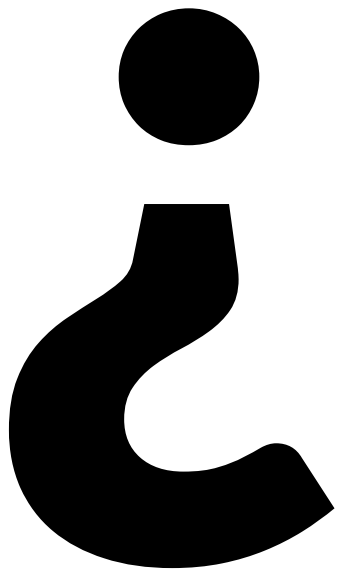
**y si queremos
guardar float's
u algún Object**



¿Pero qué pasa con el código?

```
public class Arreglo {  
    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**



1. Cambiamos a algo mas genérico como base

```
public class ArregloDinamicoObjetos {  
    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

```
ArregloDinamicoObjetos arreglo = new ArregloDinamicoObjetos(10);  
Integer numero = 10;  
  
arreglo.insertar(numero, 0);  
  
Integer salida = (Integer)arreglo.obtener(0);
```

La salida requiere una conversión

Para bien y para mal

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

```
Auto movil = new Auto("Nissan");
```

```
arreglo.insertar(movil, 0);
```

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



¡Esto puede fallar!

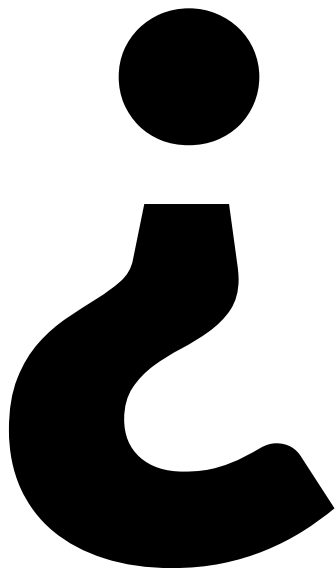
Podemos insertar de todo

La herencia puede
ayudar

Pero, ¿esto funciona?

```
public class ArregloDeInteger extends Arreglo{  
    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 ArregloDeInteger extends ArregloDinamico{  
    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 nos resuelve
*el problema...***

Para esto entra

Genéricos

Polimorfismo paramétrico

¡Versión generica!

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

¡Ahora somos libres!

```
ArregloDinamico<Integer> enteros = new ArregloDinamico<>();  
ArregloDinamico<Auto> enteros = new ArregloDinamico<>();  
ArregloDinamico<Producto> enteros = new ArregloDinamico<>();
```

Nuestro arreglo almacena sólo un tipo a la vez ahora.

—

Pero hay un problema en la construcción

¿Es este código válido?

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

¿Es este código válido?

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



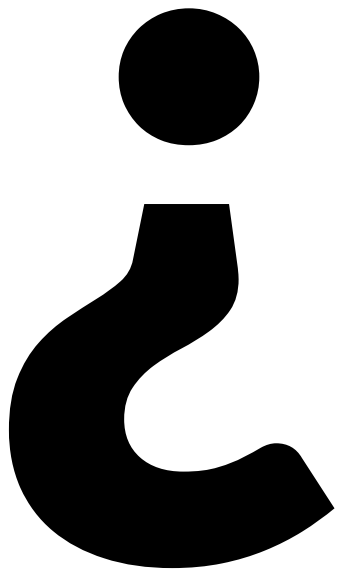
NOPE

Hay dos cosas importantes acá

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



**Este cast no se
verifica**

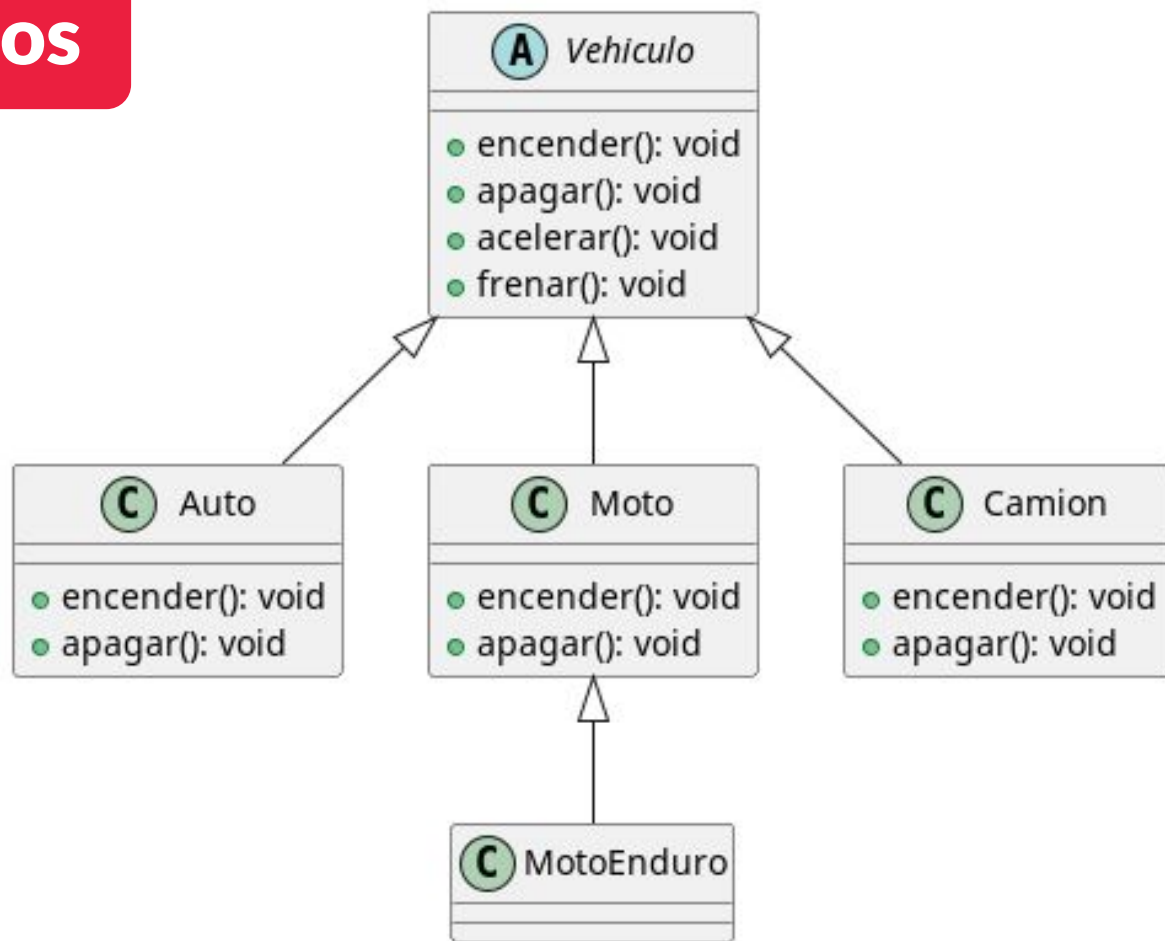


Por qué



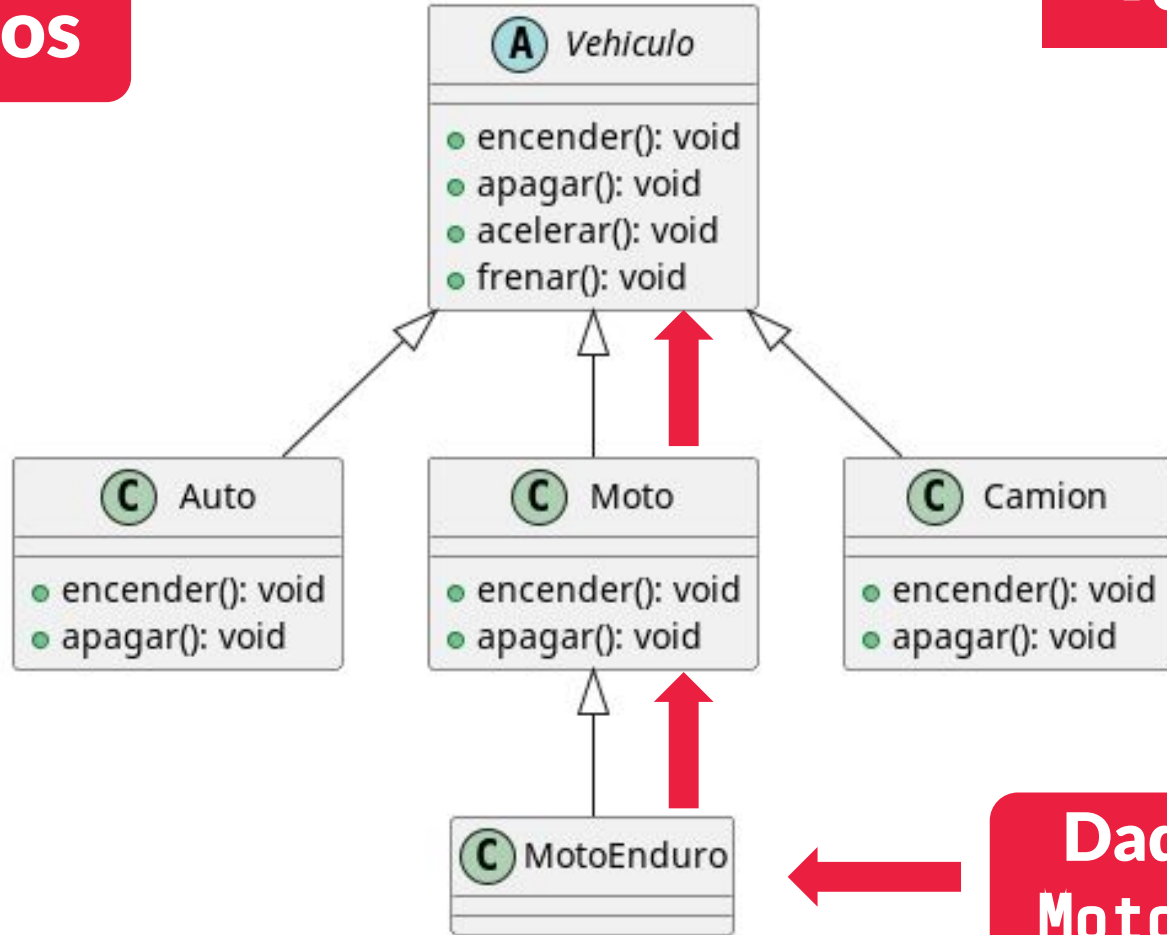
Refuerzo de casteos

Vehículos



Vehículos

↑ **Upcasting**

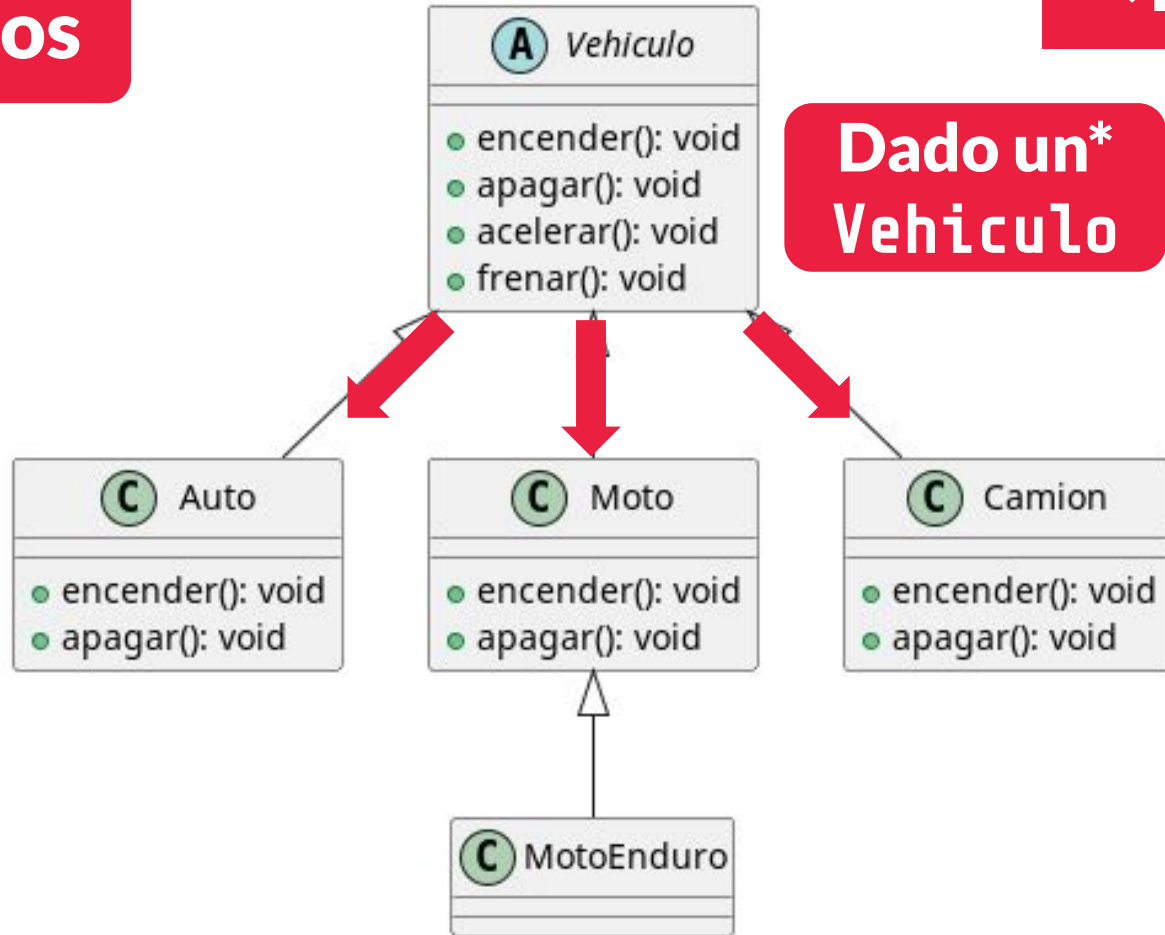


Dada una*
MotoEnduro

Vehículos

↓ Downcast

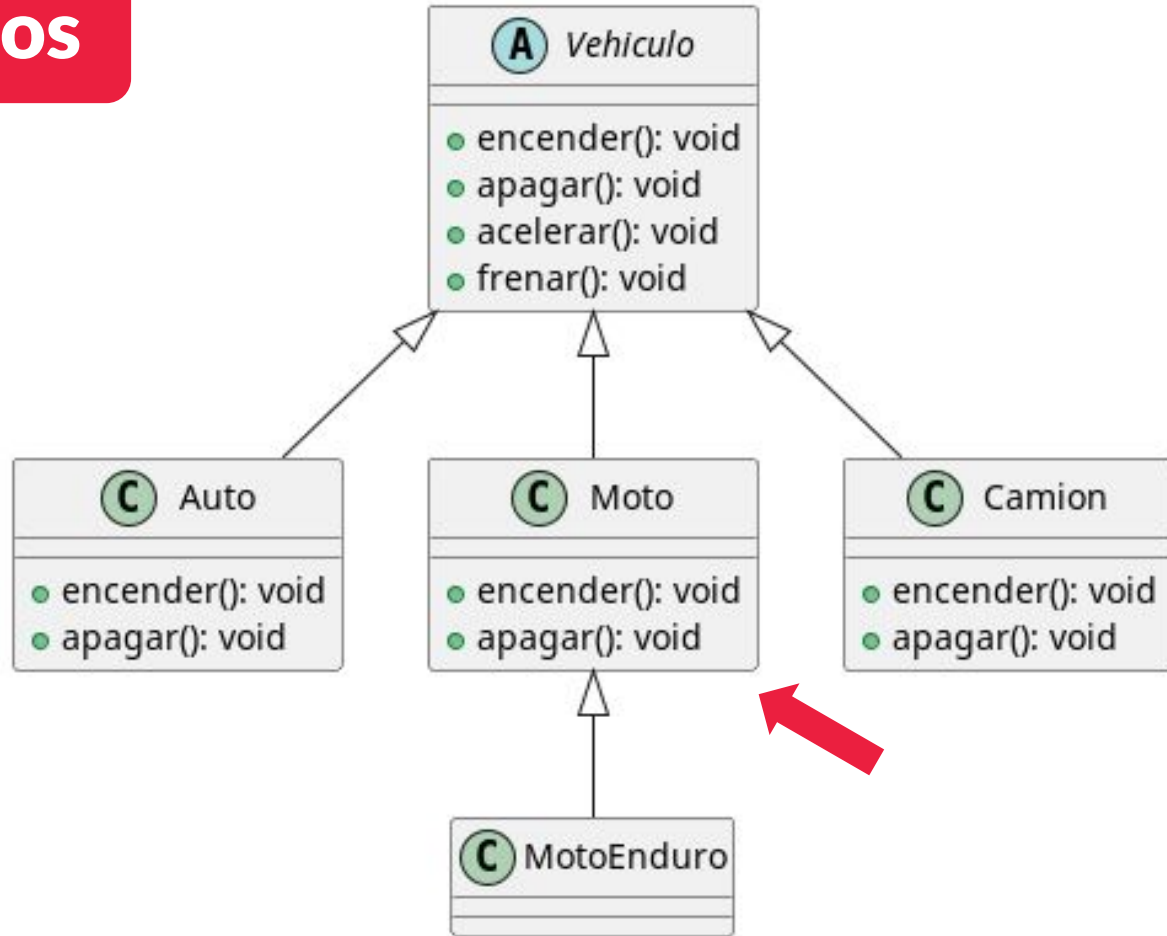
Dado un*
Vehículo



En definitiva

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

Vehículos



Por eso vimos que este código no está completo.

```
@Override
public boolean equals(Object objeto) {
    if (this == objeto){
        return true;
    }
    if (objeto == null){
        return false;
    }
    if (getClass() != objeto.getClass()){
        return false;
    }
    Vehiculo otro = (Vehiculo) objeto;

    return this.marca.equals(otro.marca);
}
```



Falta lo del getClass

Pero es mejor usar pattern matching

```
if (objeto instanceof Vehiculo otro) {  
    return this.marca.equals(otro.marca);  
}
```



¿Preguntas?

Type erasure

**La información
genérica (el tipo de
T) se pierde en
tiempo de ejecución**

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;  
    }  
}
```

¡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`)



Volviendo a la instanciación

El Type erasure hace que T sea Object

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




no hay que
verificar

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

~_ (ツ) _ / ~

Podemos usar un genérico sin <T>

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



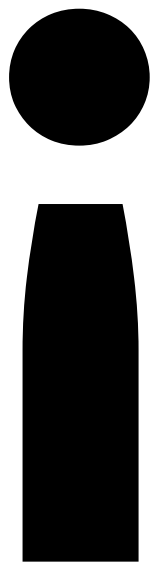
Pero esto puede dar un
`ClassCastException`



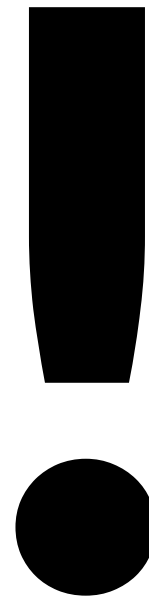
**Que es la razón
de existir de los
genericos**

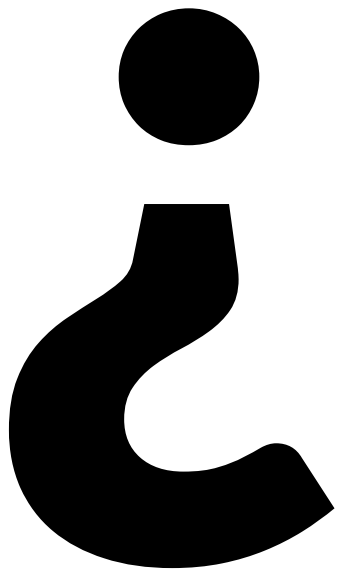


¿Preguntas?



**Muy lindo lo
genérico**





**Como podemos
implementar un**

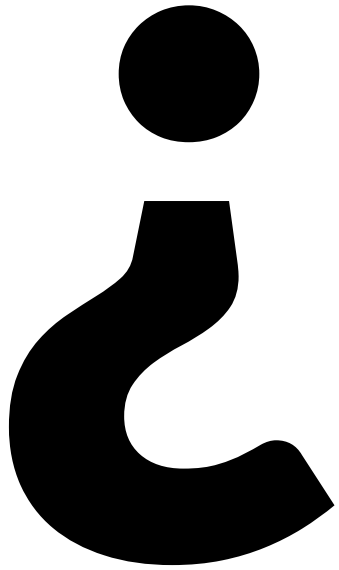


Como podemos implementar algo como

```
void insertarOrdenado(T  
valor)
```

```
void ordenar()
```

**Si no sabemos que
hay adentro...**



Como podemos comparar dos tipo T



Interfaces II

Podemos usar interfaces para comparar cosas

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

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparator.html>

De manera genérica

1

**Quien recibe los T
debe indicar que
deben incluir la
interfaz**

Pero podemos poner límites

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



**¡Solo puede recibir
cosas Comparable!**

2

**Los objetos deben
implementar la
interfaz para sí**

No hay cast que pueda fallar

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

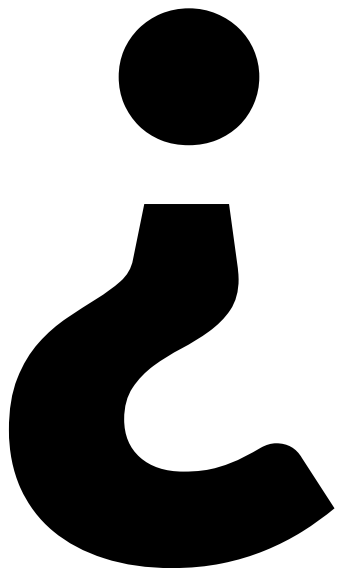
¡Porque ya es un Auto!

En resumen

**Hace de nuestros
objetos,
comparables**
(de una forma consistente)



¿Preguntas?



**Y si también
queremos
comparar por
otro atributo**



Hay otra interfaz similar

Compara dos objetos de tipo T

```
public interface Comparator<T> {  
    int compare(T uno, T otro);  
}
```

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html>

¿Y esto para que se les ocurre que existe?

Interfaces Anónimas

Implementación anónima

```
public static Comparator<Auto> porChassis(){  
    return new Comparator<Auto>() {  
        @Override  
        public int compare(Auto p1, Auto p2) {  
            return p1.chassis > p2.chassis;  
        }  
    };  
}
```

¿Se está instanciando que?



¿Preguntas?

¿y si queremos implementar?

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

Comodines genéricos

Podemos pedir que

ArregloDinamico

solo acepte

Comparables

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 **T**ipo**

¿ y ?
Comparator

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



¿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(entero);
```

```
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);
```





¿Preguntas?



Genéricos e Interfaces

unrn.edu.ar

UNRN

Universidad Nacional
de **Río Negro**



| **unrionegro**

Arreglo Generico

Cuando queremos guardar otra cosa que no sea un int

ArregloFragmentado

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

ArregloFragmentado

ArregloGenerico

ArregloDinamico

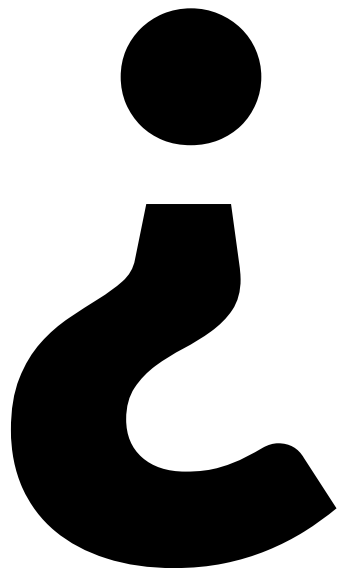


ArregloDinamico



ArregloDinamico





**Y si
necesitamos
que guarde
cualquier cosa**



ArregloFragmentado

ArregloGenerico

ArregloGenerico



ArregloGenerico



ArregloGenerico



Arreglo Fragmentado

