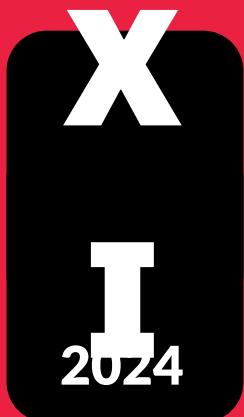


OOP+Java



UNRN

Universidad Nacional
de Río Negro



TIP5



Cuestiones generales para revisar



**Abran
hilo**

Probablemente repitamos el ejercicio

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?



Sintaxis OOP en Java

¿Cómo se crea una clase?

```
public class NombreClase {  
    espacio para atributos  
    public NombreClase(){  
        constructor  
    }  
    espacio para métodos  
}
```

Definición de clases

```
package ar.unir;
public class Contador {
    public int numero;
    public Contador(int n){
        numero = n;
    }
    public int incremento(){
        numero = numero + 1;
        return numero
    }
}
```

//Paquete
// Clase

// Atributos
// Constructor

//Método

en un archivo llamado **Contador.java**

Definición de clases

```
package ar.unir;
public class Contador {

    public int numero;
    public Contador(int n){
        numero = n;
    }
    public int incremento(){
        numero = numero + 1;
        return numero
    }
}
```

//Paquete
// Clase

// Atributos
// Constructor

//Método

en un archivo llamado **Contador . java**

Instanciando

MiClase ref = new MiClase(argumentos);



**Al instanciar,
llamamos a uno de
los constructores**

Con argumentos, parametrizado

```
public class Contador {  
    private int tope;  
    private int i;  
  
    public Contador(int maximo){  
        tope = maximo;  
        i = 0;  
    }  
}
```



Con argumentos

Con 'por defecto', sin argumentos

```
public class Contador {  
    public static final int POR_DEFECTO = 10;  
    private int tope;  
    private int i;  
  
    public Contador(){  
        tope = POR_DEFECTO;  
        i = 0;  
    }  
}
```



Aunque es posible , la inicialización va en el constructor

```
public class Contador {  
    public static final int POR_DEFECTO = 10;  
    private int tope = POR_DEFECTO;  
    private int i = 0;  
  
    public Contador(){  
        // ya vamos a ver por qué el constructor es importante  
    }  
    ... resto de los métodos  
}
```

La inicialización de los atributos va en el constructor

**En un ratito vemos
un par de detalles
adicionales**



¿Preguntas?



¿Cuál es la salida de este pequeño programa?

```
public static void main(String[] args){  
    Contador i = new Contador(10);  
    Contador j = new Contador(10);  
    sout(i.incrementar());  
    sout(j.incrementar());  
    sout(i.incrementar());  
}
```

Y como quedan al finalizar main, en ¿10, 11, 12 o 13?

**Cada instancia tiene
su propio estado**

**Los Contador en
i y j son dos separados**

**Las clases, definen
un tipo (como
typedef)**

La variable i que contiene una referencia a un Contador

```
Contador i = new Contador(10);
```

Las clases van en CamelloCase y sus atributos en DromedarioCase

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?



Ahora veamos

Un contador con ‘límite’

```
public class Contador {  
    public int limite;  
    public int posicion;  
  
    public Contador(int tope){  
        limite = tope;  
        posicion = 0;  
    }  
  
    public int incremento(){  
        if(posicion < limite) {  
            posicion = posicion +1;  
        }  
        return posicion;  
    }  
}
```

¿Cuál es la salida de este pequeño programa?

```
public static void main(String[] args){  
    Contador i = new Contador(2);  
    sout(i.incrementar());  
    sout(i.incrementar());  
    sout(i.incrementar());  
}
```

Y como quedan al finalizar main

¿Cuál es la salida de este pequeño programa?

```
public static void main(String[] args){  
    Contador j = new Contador(10);  
    j.posicion = 100;  
    j.tope = -1;  
    sout(j.incrementar());  
    sout(j.incrementar());  
    sout(j.incrementar());  
}
```

Y como quedan al finalizar main

**La misión del
comportamiento
es mantener el
estado del objeto
*consistente***

**Para ello, tenemos
que cortar el
acceso directo a
los atributos**

Con los calificadores de acceso

public /protected /private

ver la otra
se simular
nada!

Solo accesible a las clases dentro del mismo paquete

public

Calificador de acceso que indica que es accesible por todos los objetos dentro y fuera de la clase.

private

Calificador de acceso que indica que solo será accesible por los objetos de la clase.

protected*

Calificador de acceso que indica que es accesible por los objetos de la clase y sus descendientes

1

Encapsulamiento Ocultar la estructura interna de una clase.

4

**Esto significa
qué**

¡Los atributos **son** privados!

```
public class Contador {  
    private int numero;  
    private int i;  
    public Contador(int tope){  
        numero = tope;  
        i = 0;  
    }  
    public int incremento(){  
        i++;  
        return i;  
    }  
}
```

y como aplican
también a
métodos

Podemos crear métodos **internos**

```
private static void verificar(int[] arreglo)
```

```
protected static void verificar(int[] arreglo)
```

que **solo** pueden ser usadas por la clase que la define

Los atributos **son** privados

```
public class Contador {  
    private int numero;  
    private int i;  
    public Contador(int tope){  
        numero = tope;  
        i = 0;  
    }  
    public int incrementar(){  
        if (posicion >= tope) {  
            throw new ContadorException("En el máximo");  
        }  
        i = i + 1;  
        return i;  
    }  
}
```

{ Los atributos **private**
protected con justificación
}
y **nunca** public

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?



En un método

¿Cómo nos podemos referir a la instancia?

Superposición de identificadores

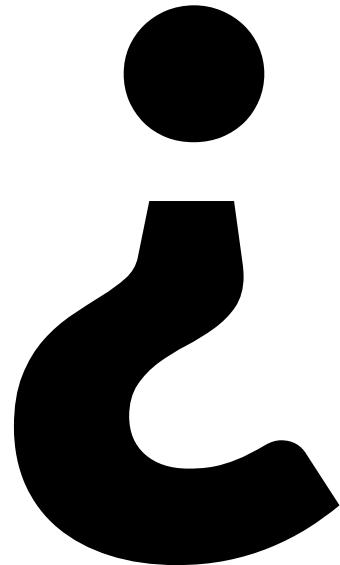
```
public class Tiempo{  
    private int hora;  
    private int minutos;  
    private int segundos;  
  
    public Tiempo(int hora, int minutos, int segundos){  
        hora = hora;  
        minutos = minutos;  
        segundos = segundos;  
    }  
}
```

¿como desempatamos?

¿a qué nos estamos refiriendo?

Desempatando con this

```
public class Tiempo{  
    private int hora;  
    private int minutos;  
    private int segundos;  
  
    public Tiempo(int hora, int minutos, int segundos){  
        this.hora = hora;  
        this.minutos = minutos;  
        this.segundos = segundos;  
    }  
}
```



¿this?
What's this



1

6

this

La referencia al objeto

Todos los *objetos* tienen una
referencia a sí mismos.



Paquetes

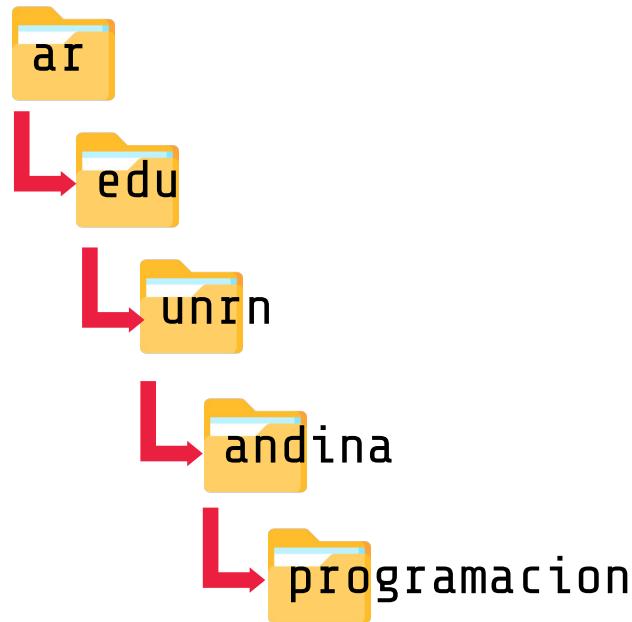
¿Qué son?

Una forma de organizar y agrupar clases en proyectos más grandes.

Formalmente, son la URL de la organización invertida

ar.edu.unrn.andina.programacion.

arreglos
modulo
consola



Pero suele ser largo

La primera de cada archivo java debe reflejar su ubicación

```
package ar.edu.unrn.andina.programacion.arreglos;
```

(Esto lo hace solo IntelliJ)

**Los paquetes
deben comenzar
en
ar.unrn e ir en
minúsculas**

**Se suele enviar a
las excepciones a
un paquete
específico**

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?

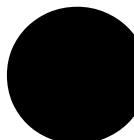


Métodos



Detalles sobre el constructor

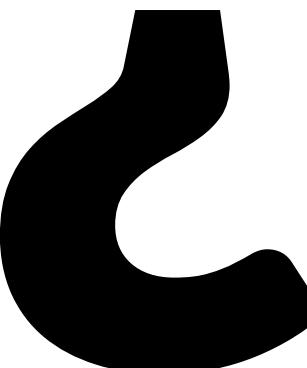
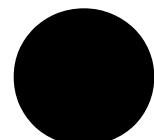
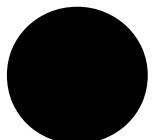




Qué hacía el constructor

Estructura base

```
public class MiClase{  
    private String unaCadena;  
  
    public MiClase(String argumento){  
        unaCadena = "Hola objetos " + argumento;  
        // El Constructor le da un valor a los atributos  
        // Contiene las instrucciones sobre la inicialización  
    }  
}
```



**Una clase,
puede tener
más de uno**

opcionalmente,
con argumentos

¡Múltiples constructores!

```
class MiClase{  
    String unaCadena;  
  
    MiClase(){  
        unaCadena = "Hola Objetos";  
        // Sin argumentos, y si no dice nada...  
    }  
  
    MiClase(String cadena){  
        unaCadena = cadena; // Con argumentos  
    }  
}
```

Dos instanciaciones válidas

**¡Donde la clase
se vuelve objeto!**

```
MiClase uno = new MiClase();
```

```
MiClase dos = new MiClase("Romero");
```



Clase, ¡hoy te convertís en héroe-objeto!

Como sabe
a qué
constructor
llamar

Introducción a
**sobrecarga de
métodos**



Al llamar un
método

**Se busca el método
cuyos *tipos* de
argumento coincidan**

Esto funciona con constructores

```
MiClase uno = new MiClase();
```

```
MiClase dos = new MiClase("Roberto");
```



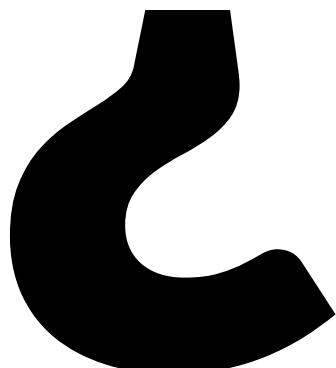
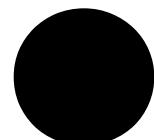
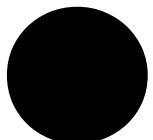
¡Solo se ejecuta uno!

peeeeeero

**Los podemos
encadenar**

¡Pero se pueden encadenar!

```
class MiClase{  
    String unaCadena;  
  
    MiClase(){  
        this("el argumento por defecto");  
    }  
  
    MiClase(String argumento){  
        unaCadena = argumento; // Con argumentos  
    }  
}
```



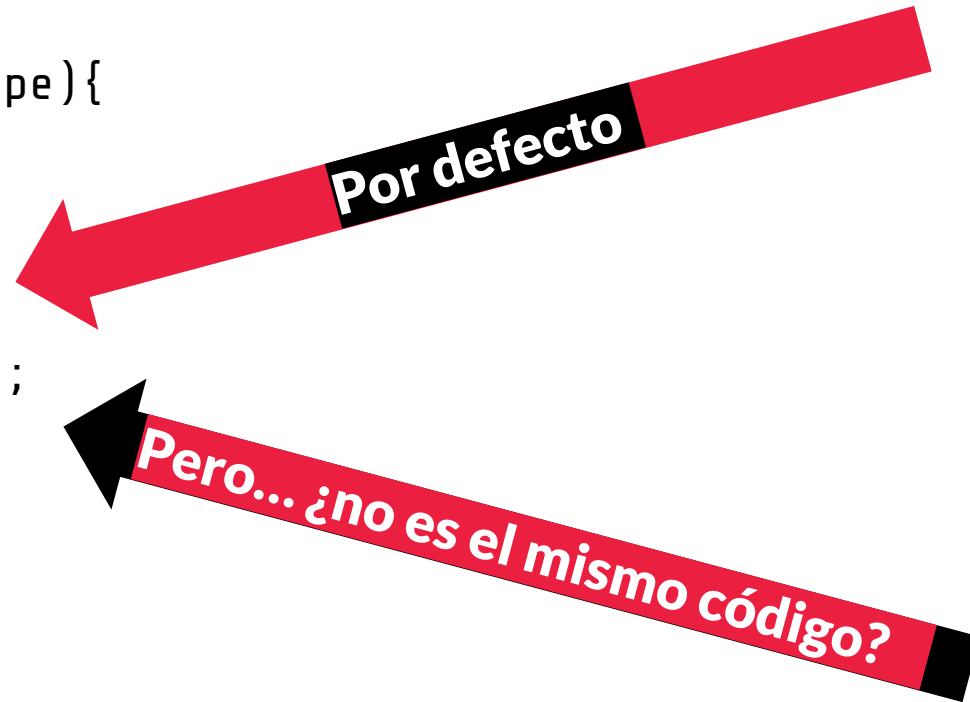
Cuando encadenar

Para reducir la duplicación de código

Un ejemplo

Encadenado de constructores

```
public class Contador {  
    public static final int POR_DEFECTO = 10;  
  
    public Contador(int tope){  
        numero = tope;  
        i = 0;  
    }  
    public Contador(){  
        numero = POR_DEFECTO;  
        i = 0;  
    }  
}
```



¡Lo mismo pero *con diferentes argumentos*!

```
public Contador(int tope){  
    numero = tope;  
    i = 0;  
}  
public Contador(){  
    numero = POR_DEFECTO;  
    i = 0;  
}
```

Por lo general tenemos uno mas completo

```
public class Contador {  
    public static final int POR_DEFECTO = 10;  
  
    public Contador(int tope){  
        numero = tope;  
        i = 0;  
    }  
  
    public Contador(){  
        this(POR_DEFECTO);  
    }  
}
```



Por defecto + encadenamiento

Es técnicamente lo mismo, pero es menos claro

```
public class Contador {  
  
    public Contador(int tope){  
        this();  
        numero = tope;  
    }  
  
    public Contador(){  
        numero = POR_DEFECTO;  
        i = 0;  
    }  
}
```



Es necesario llamarlo para completar la inicialización

Ya tenemos dos, ¿por qué no tres?

```
public class Contador {  
  
    public Contador(int tope){  
        numero = tope;  
        i = 0;  
    }  
  
    public Contador(int[] arreglo){  
        this(arreglo.length);  
    }  
}
```

Más sobre la Sobrecarga



Sobrecarga de métodos

```
class Sobrecargado{  
    int metodoSobrecargado(int a, int b);  
    int metodoSobrecargado(int a, float b);  
}
```

Sobrecarga de métodos

```
class Sobrecargado{  
  
    int metodoSobrecargado(int a, int b);      //1  
    int metodoSobrecargado(int a, float b);     //2 } ← OK  
  
    int metodoSobrecargado(int uno, int dos); //3 ← No OK  
  
}
```

**solo 've' la
combinación de tipos
en los argumentos**

El identificador que tenga el argumento no es tenido en cuenta

El orden de los tipos importa

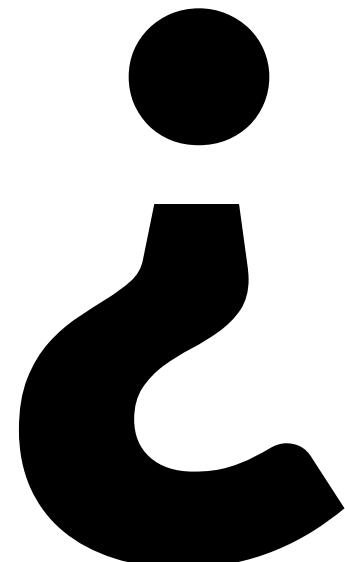
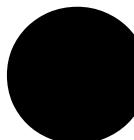
```
class SobrecargadoDos{  
    void metodoSobrecargado(int a, float b);  
    void metodoSobrecargado(float a, int b);  
}
```

}

También OK

Pero con el retorno no alcanza

```
class SobrecargadoDos{  
    int metodoSobrecargado();  
    float metodoSobrecargado(); } ← No OK  
}
```



Para que se usa

```
public class Contador {  
    public int limite;  
    public int posicion;  
  
    public Contador(int tope){  
        limite = tope;  
        posicion = 0;  
    }  
  
    public int incremento(){  
        if(posicion < limite) {  
            posicion = posicion +1;  
        }  
        return posicion;  
    }  
  
    public int incremento(int i){  
        //verifi  
    }  
}
```

Continuando con el Contador de antes

```
public class Contador {  
    public int limite;  
    public int posicion;  
  
    public Contador(int tope){  
        limite = tope;  
        posicion = 0;  
    }  
}
```

Una implementación de incremento sería

```
public int incremento() {  
    if (posicion >= tope) {  
        throw new ContadorException("Llegamo al máximo");  
    }  
    posicion = posicion + 1;  
    return posicion;  
}
```

Podemos agregar una versión ‘mas general’

```
public int incremento(int cantidad) {  
    if (posicion + cantidad > tope) {  
        throw new ContadorException("Nos pasamos!");  
    }  
    posicion = posicion + cantidad;  
    return posicion;  
}
```

```
public int incremento() {  
    if (posicion >= tope) {  
        throw new ContadorException("Llegamo al máximo");  
    }  
    posicion = posicion + 1;  
    return posicion;  
}
```

```
public int incremento(int cantidad) {  
    if (posicion + cantidad > tope) {  
        throw new ContadorException("Nos pasamos!");  
    }  
    posicion = posicion + cantidad;  
    return posicion;  
}
```

¡Pocos cambios! Pero en definitiva, es lo mismo

Mucho menos código para probar :-)

```
public int incremento() {  
    return incremento(1);  
}
```

Se reduce a un caso especial de la otra forma

**Da contexto
específico bajo el
mismo nombre**

Mismo comportamiento, diferente información de entrada

1

5

Sobrecarga

**Dos métodos con el mismo
nombre y diferente combinación
de tipos en los argumentos**



¿Preguntas?



Veamos más de cerca Object

1

**Todas las clases
extienden a Object
o a quien
indiquemos con
extends**

**El resultado de esto,
es que todo es un
Object**

2

**Es el mínimo común
denominador en
comportamiento**

Object define (entre otras cosas)

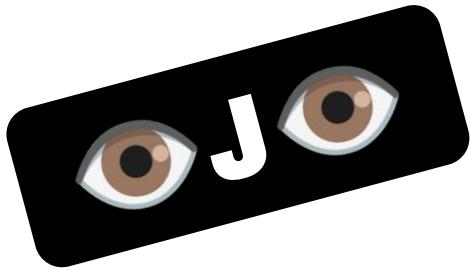
- equals - si este es igual a otro
- hashCode - un número que representa
- toString - representación textual del contenido (se llama automáticamente al concatenar)

Clase usuario

```
public class Usuario {  
  
    private LocalDate fechaNacimiento;  
    private String nombre;  
    private String apellido;  
  
    public User(LocalDate fechaNacimiento, String nombre, String apellido) {  
        this.fechaNacimiento = fechaNacimiento;  
        this.nombre = nombre;  
        this.apellido = apellido;  
    }  
}
```

equals
es la igualdad de
dos objetos

*¿apuntan a
la misma
cosa?*



**equals no es lo
mismo que ==**

*¿apuntan al
mismo
lugar?*

==

**es la igualdad de
referencias**

Atributos

El contrato de equals

Reflexivo

Un objeto debe ser igual a sí mismo

Simétrico

`a.equals(b)` tiene que ser igual que `b.equals(a)`

Transitivo

si $a.equals(b)$ y $b.equals(c)$, entonces $a.equals(c)$

Consistente

El valor de `equals` solo cambia con los atributos, no se admite aleatoriedad.

Las comparaciones base en equals

```
this == otro
```

```
otro == null
```

```
getClass() == otro.getClass()
```

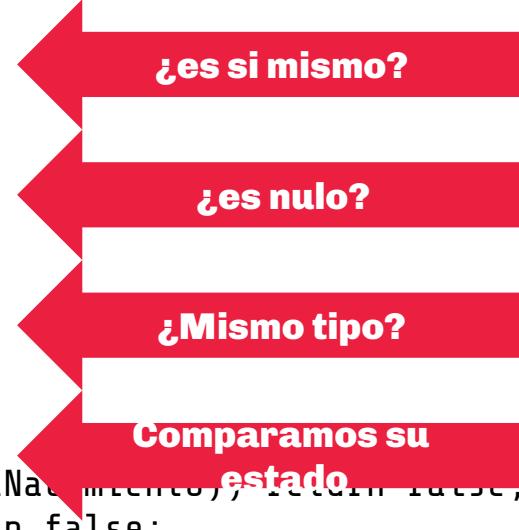
¿es si mismo?

¿es nulo?

¿Mismo tipo?

La igualdad de dos usuarios

```
@Override  
public boolean equals(Object objeto) {  
    if (this == objeto){  
        return true;  
    }  
    if (objeto == null){  
        return false;  
    }  
    if (getClass() != objeto.getClass()){  
        return false;  
    }  
    Usuario user = (Usuario) objeto;  
  
    if (!fechaNacimiento.equals(user.fechaNacimiento)) return false;  
    if (!nombre.equals(user.nombre)) return false;  
    return apellido.equals(user.apellido);  
}
```



¿es si mismo?

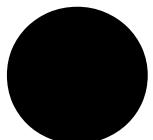
¿es nulo?

¿Mismo tipo?

Comparamos su estado

Al entrar Object en juego, las conversiones pueden fallar

```
if (getClass() == objeto.getClass()){
    Usuario user = (Usuario) objeto;
    // resto de la comparación.
}
```



**No podemos
usar la
sobrecarga con
equals**

**Teóricamente sí,
pero...**

**No es posible
garantizar que los
atributos del
contrato se
cumplan**

Por otro lado:

¿@Override?

**¿Acaso el
compilador no
sabe qué estamos
extendiendo?**

**Evita que la
sobrecarga nos
sorprienda**

Si solo implementamos la sobrecarga, esto no puede suplantar la versión de Object (no coincide el tipo)

```
public boolean equals(Object objeto);
```

```
public boolean equals(Usuario otro);
```

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?





Depende de sus atributos

Pero hay algo para
respetar

El protocolo de equals y hashCode

¿hashcode?

Es su identificación

¡Y fuertemente usada para conjuntos y estructuras!

*Relacionado con
su posición en
memoria*

**Si dos objetos son
equals**

Su hashCode
es igual

Pero, dos objetos
diferentes pueden
tener el m
hashCo

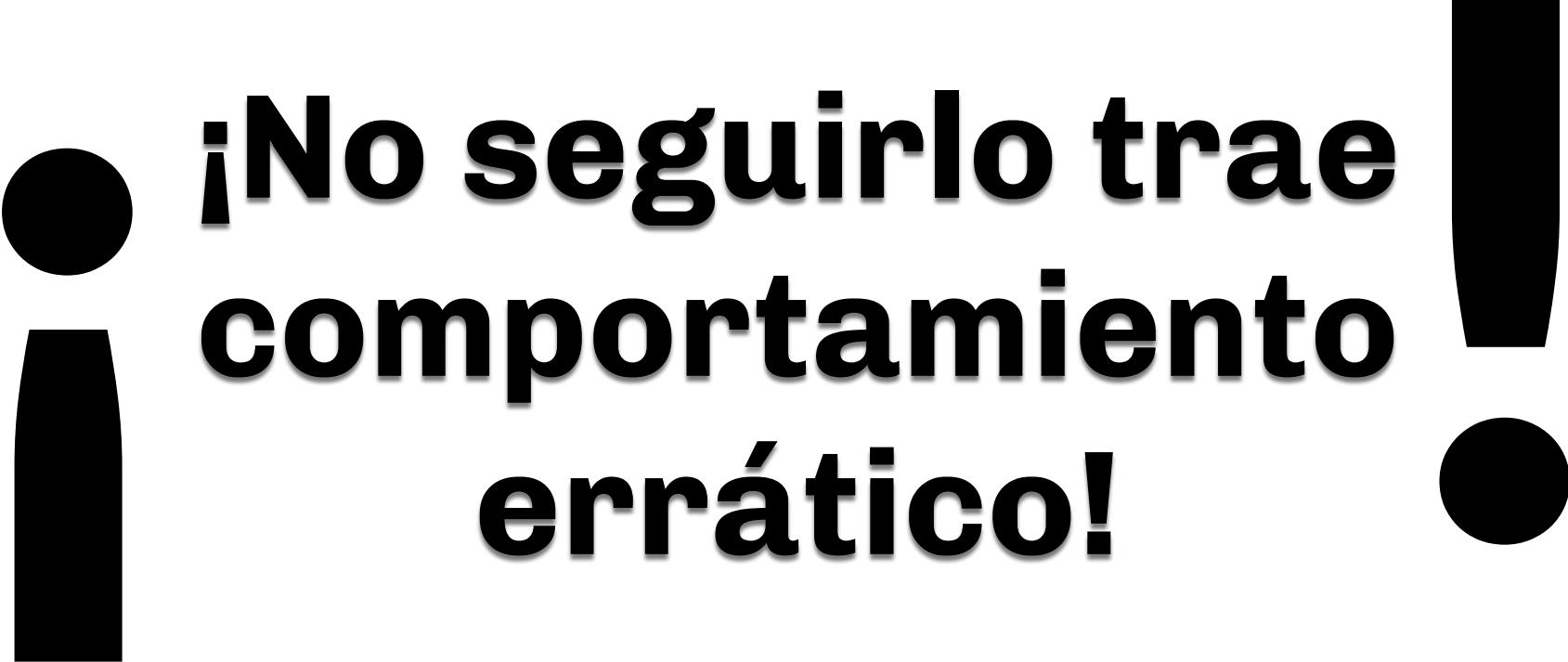


2026

Por lo que no se debe de usar para implementar equals



```
@Override  
public int hashCode() {  
// Objects.hashCode(objeto);  
    return Objects.hash(todos, los, atributos, juntos);  
    return result;  
}
```



**¡No seguirlo trae
comportamiento
errático!**

**Por suerte es
fácilmente
testable**

*`assertEquals` de dos objetos
construidos con los mismos valores

iCollections!

Se usa con
Set - Conjuntos
Map - Diccionarios

instanceof

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?



TP6

Clases y Objetos 1

unrn.edu.ar

UNRN

Universidad Nacional
de Río Negro



| unrnionegro