# JAVA

## Equals Autoboxing & Enumerados

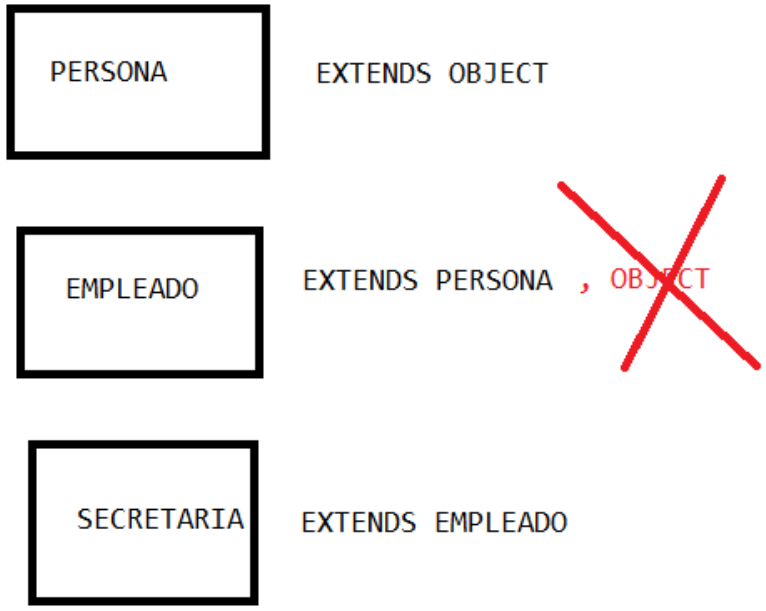
TOGETHER. FREE YOUR ENERGIES

# Objetivo

- **Lograr que aprendamos a programar en Java, hasta el punto de construir aplicaciones reales en la tecnología.**

# Agenda de hoy

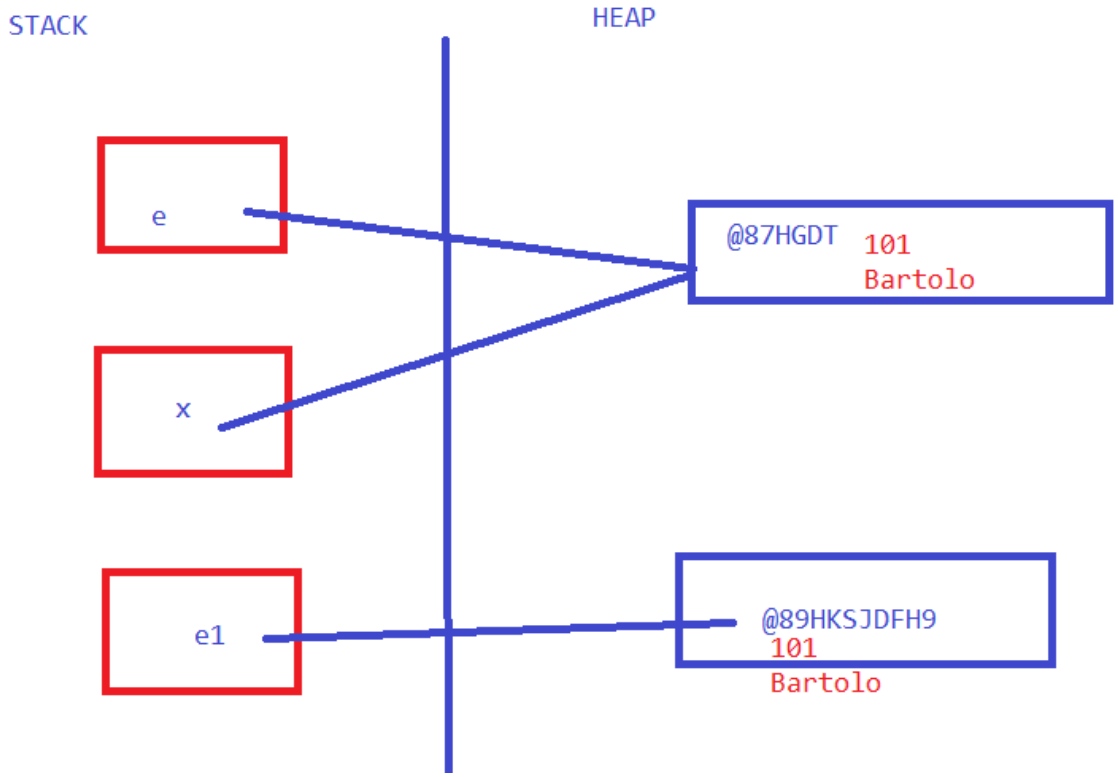
- ✓ **Parte 1: Herramientas de base y J2EE**
  - ✓ **Módulo 1: El lenguaje Java**
    - ✓ **equals y hashCode**
    - ✓ **Generics**
    - ✓ **Autoboxing**
    - ✓ **Tipos Enumerados**



# equals y hashCode I

## equals

- **El método equals sirve para implementar un test de igualdad 'state based', en el que dos objetos se consideran iguales si tienen el mismo estado.**
- **Provee la noción de 'igualdad lógica', que difiere de la mera identidad.**
- **Existe un contrato, que juramos respetar cuando implementamos el equals, para no tener bugs horrorosos y difíciles de encontrar.**
  - **El equals es una relación de equivalencia**
  - **El equals es consistente**
  - **Comportamiento con null values**




```
public static void testEquals() {
    Empleado e = new Empleado(101, "Bartolo");
    Empleado x= e;
    System.out.println(x.equals(e)); // da TRUE

    Empleado e1 = new Empleado(101, "Bartolo");
    System.out.println(x.equals(e1)); // da FALSE
}
```

# equals y hashCode II

Entonces siempre implemento el método equals?

Noooo! En los siguientes casos no conviene:

- **Cada instancia de la clase es única**
- **No nos importa que nuestra clase tenga 'igualdad lógica'**
- **Alguna superclase de nuestra clase ya hace overriding de equals, y con esa igualdad alcanza**

# equals y hashCode III

## Una receta para un método equals perfecto

```
public boolean equals(Object otherObject){
[...]
}
```

```
obj1.equals(obj2);
```

1. Nombrar el parámetro otherObject.

2. Testear si this es idéntico a otherObject:

```
if (this == otherObject) return true;
```

# equals y hashCode IV

## Una receta para un método equals perfecto

3. Chequear si otherObject es null y devolver false si lo es.

```
if (otherObject == null) return false;
```

4. Comparar las clases de this y otherObject

Si la semántica del equals puede cambiar en las subclases

```
if (getClass() != otherObject.getClass()) return false;
```

Sino

```
if (!(otherObject instanceof ClassName)) return false;
```

# equals y hashCode V

5. Castear otherObject a una variable del mismo tipo que this.

```
ClassName other = (ClassName) otherObject
```

6. Comparar ahora los campos de ambos, según nuestra noción de igualdad.
Usar  == para tipos primitivos, equals para objetos.
Si todos los campos matchean, devolver true, sino false

```
return field1 == other.field1
&& field2.equals(other.field2)
&& . . .;
```

Si redefinimos equals en una subclase, incluirrr una llamada a super.equals(other)

# equals y hashCode VI

## hashCode

- **El valor que devuelve hashCode es un entero que mapea un objeto dentro de un bucket en una tabla de hash.**

- **Un mismo objeto devuelve siempre el mismo hash code. Sin embargo, varios objetos pueden tener el mismo hashCode**


que se implementa equals se debe implementar también hashCode

# equals y hashCode VII

Una receta posible para un método hashCode

```
public int hashCode(){
[...]
}
```

Para cada campo significativo f en el objeto, hacer

Si f es una referencia a un objeto, invocar el hashCode de ese objeto.

hashCodeADevolver = hashCodeADevolver + siguientePrimo * f.hashCode()

# equals y hashCode VIII

## Una receta posible para un método hashCode

(Si f es null, no suma nada)

Si f es un tipo primitivo, crear el "wrapper" (en unos slides más definiremos esto rigurosamente) e invocar a hashCode del wrapper

```
class A
{
	public int hashCode()
	{
		return 7 * campoString.hashCode()
		+ 11 * new Double(campoDouble).hashCode()
		+ 13 * campoFecha.hashCode();
		//+ ,,,,, (los demás campos significativos)
	}
}
```

# Ejercicio

## Ejercicio IX de la guía

# Generics I

Antes de Java 1.5...

```
public class ArrayList
{
        public Object get(int i) { . . . }
        public void add(Object o) { . . . }
        . . .
        private Object[] elementData;
}
```


Dolor 1

```
ArrayList files = new ArrayList();
. . .
String filename = (String) names.get(0);
```

Dolor 2

```
files.add(new File(". . ."));
```

# Generics II

Clases genéricas

```
ArrayList<String> files = new ArrayList<String>();
```

```
String filename = files.get(0);
```

```
public class Pair<T> {
private T first;
private T second;

public Pair() {
      first = null;
      second = null;
}
public Pair(T first, T second) {
      this.first = first;
      this.second = second;
}
public T getFirst() {
      return first;
}
public T getSecond() {
      return second;
}
public void setFirst(T newValue) {
      first = newValue;
}
public void setSecond(T newValue) {
      second = newValue;
}
}
```

```
Pair<String> aPair = new Pair<String>();
```

```
String first = aPair.getFirst();
```

# Generics III

Si tenemos

```
Pair<Empleado, Integer>
```

y Empleado hereda de Persona, eso NO es un subtipo de

```
Pair<Persona, Persona>
```

# Generics IV

## Métodos genéricos

```
class ArrayAlg
{
    public static <T> T getMiddle(T[] a) {
        return a[a.length / 2];
    }
}
```

# Generics VI

## Type bounds

Solución

```
public static <T extends Comparable> T min(T[] a) . .
```

```
public static <T extends Comparable & Serializable> T min(T[] a)
```

# Generics V

## Type bounds

Cuál es el problema con este código?

```
class ArrayAlg {
public static <T> T min(T[] a)
{
        if (a == null || a.length == 0)
                return null;
        T smallest = a[0];

        for (int i = 1; i < a.length; i++)
                if (smallest.compareTo(a[i]) > 0)
                        smallest = a[i];
        return smallest;
}
}
```

# Generics VII

## Wildcard Types

Pair<? extends Employee>

Pair<? super Manager>

Pair<?>

```
public static void printBuddies(Pair<? extends Employee> p) {
    Employee first = p.getFirst();
    Employee second = p.getSecond();
    System.out.println(first.getName() + " and " + second.getName());
}
```

# Generics VIII

## Wildcard Types

No

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Si

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

# Tipos Enumerados I

```
public enum Size {
SMALL, MEDIUM, LARGE, EXTRA_LARGE
};
```

```
Size s = Size.MEDIUM;
```

```
enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
    private Size(String abbreviation) { this.abbreviation = abbreviation; }
    public String getAbbreviation() { return abbreviation; }
    private String abbreviation;
}
```

```
Size s = Size.MEDIUM;
s.getAbbreviation();
```

# Autoboxing I

En Java teníamos tipos primitivos:

```
int, long, float, double, short, byte, char, boolean
```

Pero muchos métodos esperan objetos, para trabajar, no tipos primitivos:

Inválido

```
ArrayList<int> list = new ArrayList<int>();
```

Entonces, cómo hacemos?

# Autoboxing II

Java provee tipos wrapper para los tipos primitivos:

```
Integer, Long, Float, Double, Short, Byte, Character, Boolean
```

**autoboxing**

el compilador convierte automáticamente el tipo primitivo en el wrapper:

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

```
list.add(3);
```

funciona porque el compilador lo traduce a

```
list.add(new Integer(3));
```

# Autoboxing III

...y el tipo wrapper en el tipo primitivo


```
int n = list.get(i); //recordar que la lista era de Integer
```

funciona porque el compilador lo traduce a

```
int n = list.get(i).intValue();
```

Otro ejemplo:

```
Integer n = 3;
n++;
```

# Ejercicio

## Ejercicio X de la guía

Muchas gracias!