

TEMA 1

CONCEPTOS DE JAVA PARA ESTRUCTURAS DE DATOS
Herencia y polimorfismo

Objetivos

El objetivo general de este primer tema es alcanzar los conceptos básicos de la Programación Orientada a Objetos y cómo son soportados por el lenguaje *Java*:

- ❑ Avanzar en la descripción del propio lenguaje: clases abstractas e interfaces
- ❑ Señalar el concepto de polimorfismo y sus ventajas
- ❑ Programación generica usando herencia
- ❑ Profundizar en el uso de clases envoltorio
- ❑ Iniciar el desarrollo de librerías propias

Índice

1.- Conceptos básicos

- Tipos primitivos vs. referencias

2.- Objetos y clases

- Diseño de una clase: atributos y métodos
- Paquetes
- Modificadores de acceso

3.- Herencia

- Clases bases y derivadas
- Polimorfismo y enlace dinámico
- Clases abstractas

4.- Interfaces

1.- CONCEPTOS BÁSICOS

Tipos primitivos

- Los tipos primitivos de *Java* son similares a los de los demás lenguajes de programación:
 - **boolean**: *true* o *false*
 - **char**: caracteres unicode de 16 bits
 - **byte**: enteros de 8 bits con signo
 - **short**: enteros de 16 bits con signo
 - **int**: enteros de 32 bits con signo
 - **long**: enteros de 64 bits con signo.
 - **float**: reales de 32 bits
 - **double**: reales de 64 bits
- En los tipos primitivos se almacena directamente el valor

1.- CONCEPTOS BÁSICOS

Referencias

- El resto de tipos de datos son referencias:
 - Vectores
 - Objetos: instancias de clases
- Las variables de tipo referencia almacenan direcciones y no valores directamente
- Una referencia es la dirección de un área en memoria, la cual se reserva con el operador *new*

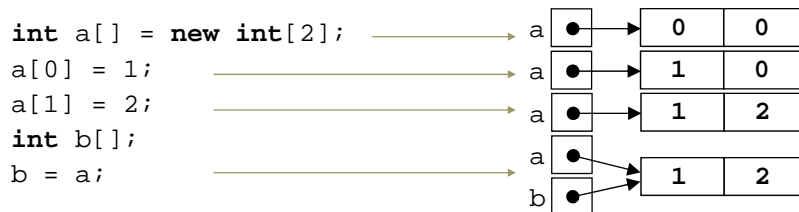
1.- CONCEPTOS BÁSICOS

Ejemplo: tipos primitivos vs. referencias

Ejemplo con tipos primitivos:



Ejemplo con referencias:



1.- CONCEPTOS BÁSICOS

Ejercicio

- Ejercicio 1: Indica el contenido de los vectores a y b tras las siguientes instrucciones:

```
int a[];
int b[] = new int[3];
a = new int[3];
a[0] = 1;
b[1] = a[0];
a = b;
b[0] = a[1];
```

2.- OBJETOS Y CLASES

Definición

- Una *clase Java* está formada por un conjunto de *atributos* que almacenan datos y un conjunto de *métodos* que definen su funcionalidad
- Un *objeto* de una clase *Java* contiene un tipo de información que puede ser manipulada (establecida, modificada o consultada) a través del conjunto de métodos de la clase a la que pertenece

2.- OBJETOS Y CLASES

Diseño de una clase: *selección de atributos*

- Se quiere diseñar una clase para poder representar objetos de tipo círculo
- Primero deben definirse los **atributos** de la clase:
 - Modela la relación TIENE UN(A)

Ejemplo: ¿Qué tiene un círculo?

⇒ Un **radio** y un **color**

```
public class Circulo {
    /** Atributos */
    private double radio;
    private Color color;
    ...
}
```

2.- OBJETOS Y CLASES

Diseño de una clase: *constantes*

- Para definir constantes dentro de una clase se usan los siguientes modificadores de acceso:
 - **final**: indica que el valor de un atributo no puede ser modificado
 - **static**: indica que el valor del atributo es el mismo para todos los objetos de la misma clase

```
public class Circulo {
    /** Atributos */
    private double radio;
    private Color color;
    private static final double RADIO_POR_DEFECTO = 3.0;
    private static final Color COLOR_POR_DEFECTO = Color.black;
    ...
}
```

2.- OBJETOS Y CLASES

Diseño de una clase: *definición de métodos*

- Los métodos definen la funcionalidad del objeto:
 - **Constructores**: inicializan los atributos del objeto
 - **Consultores**: consultan información sin modificar el objeto
 - **Modificadores**: modifican el estado del objeto
 - **toString()**: devuelve una cadena con una descripción del objeto actual
 - **equals(Object x)**: compara el objeto *x* con el actual, indicando si son iguales

2.- OBJETOS Y CLASES

Diseño de una clase: *métodos constructores*

- Si no se define un constructor, los atributos se inicializan con su valor por defecto: *radio = 0* y *color = null*
- Un constructor tiene el nombre de la clase y no devuelve nada.

Ejemplos:

```
public Circulo(double radio, Color color) {
    this.radio = radio;
    this.color = color;
}

public Circulo() {
    this(RADIO_POR_DEFECTO, COLOR_POR_DEFECTO);
}
```

El constructor sin parámetros invoca al de dos parámetros

2.- OBJETOS Y CLASES

Diseño de una clase: *consultores y modificadores*

- Ejemplo de método *consultor*:

```
public double getRadio() {  
    return radio;  
}
```

- Ejemplo de método *modificador*:

```
public void setRadio(double radio) {  
    this.radio = radio;  
}
```

- Ejercicio 2: Implementar los métodos *area* y *perimetro*, haciendo uso del siguiente atributo de la clase *Math*:

```
public static final double PI
```

2.- OBJETOS Y CLASES

Diseño de una clase: *el método toString()*

- Devuelve una cadena que muestra el estado del objeto
- Es un método predefinido de la clase *Object* (clase base de todas las clases)
- Definir el método *toString()* implica la sobrescritura o redefinición del método *toString()* de la clase *Object*

```
public String toString() {  
    String res = "Círculo de radio " + radio;  
    res += " y color " + color + "\n";  
    return res;  
}
```

2.- OBJETOS Y CLASES

Diseño de una clase: *el método equals()*

- ❑ Comprueba si dos objetos de una misma clase son iguales
- ❑ Es también un método predefinido de la clase **Object**

```
public boolean equals(Object x) {  
    Circulo cX = (Circulo) x;  
    return (radio == cX.radio) && (color.equals(cX.color));  
}
```

- ❑ La diferencia entre *equals* y el operador == radica en que éste último comprueba la igualdad de referencias, no de objetos

Ejemplo:

```
Circulo c1 = new Circulo();  
Circulo c2 = new Circulo();
```

- Se cumple que `c1 != c2` y, sin embargo, `c1.equals(c2) == true`

2.- OBJETOS Y CLASES

Paquetes en *Java*

- ❑ El mecanismo básico para organizar un grupo de clases que guardan alguna relación entre sí es el **package**
- ❑ Los paquetes *Java* estándar son:
 - **java.lang**: contiene las clases básicas (*Integer*, *Math*, *String*, etc.)
 - **java.util**: contiene diversas utilidades (*Random*, *Date*, *StringTokenizer*, *Scanner*, etc.)
 - **java.io**: clases para manipular ficheros y canales de E/S
 - **java.awt**: contiene clases para diseñar interfaces gráficas de usuario (*Color*, *Graphics*, etc.)

2.- OBJETOS Y CLASES

Paquetes en *Java*: *utilización*

- Una clase *C* en un paquete *p* se especifica como *p.C*
 - Ejemplo: crear un objeto de la clase *Color* (que se encuentra en el paquete *java.awt*)

```
java.awt.Color color;           // Declaramos la variable
color = new java.awt.Color(0,0,0); // Reserva de memoria
```

- Para simplificar se utiliza la directiva ***import***:

```
import java.awt.*;           // O, también, java.awt.Color
...
Color color = new Color(0,0,0);
```

2.- OBJETOS Y CLASES

Paquetes en *Java*: *creación*

- Para indicar que una clase *C* es de un paquete *P*, se deben de seguir los siguientes pasos:
 - Las clases en *Java* se implementan en un fichero con el mismo nombre y con extensión *.java*
 - La primer línea del fichero *C.java* es la instrucción:
package *P*;
 - El fichero *C.java* debe estar en el subdirectorio *P*
 - El subdirectorio *P* debe estar en la variable de entorno ***CLASSPATH***
 - Todas las clases que no forman parte de un paquete, pero se pueden alcanzar a través de la variable ***CLASSPATH***, se consideran parte del mismo paquete

2.- OBJETOS Y CLASES

Principio de ocultación de la información

- Para las clases, sus métodos y sus atributos hay que especificar los modificadores de acceso oportunos. Para ello se debe seguir el principio de **ocultación de la información**:
 - **Especificación**: descripción de lo que se puede hacer sobre un objeto
 - **Implementación**: descripción de los detalles internos de cómo se satisface la especificación (no es necesario conocer los atributos ni la implementación de una clase para utilizarla)
- Por este motivo, los atributos de una clase suelen ser, por regla general, ser **privados** y sus métodos **públicos**

2.- OBJETOS Y CLASES

Modificadores de acceso

- Los siguientes modificadores son aplicables a clases, métodos y atributos:
 - **friendly**: cuando no se especifica ningún modificador, es visible dentro del paquete en el que se define
 - **private**: visible sólo desde la propia clase
 - **protected**: visible en las clases derivadas (subclases) y dentro del paquete en el que se define
 - **public**: visible desde cualquier sitio
 - **final**: indica que no puede ser modificado
 - **static** (no aplicable a clases*): común a todos los objetos de una misma clase

2.- OBJETOS Y CLASES

Modificadores de acceso, tabla resumen

Modificador de visibilidad	Propia clase	Mismo paquete	Subclases	Otras clases / paquetes
private	Sí	No	No	No
friendly (sin modificador)	Sí	Sí	No	No
protected	Sí	Sí	Sí	No
public	Sí	Sí	Sí	Sí

2.- OBJETOS Y CLASES

Métodos estáticos (1/2)

- ❑ Los métodos estáticos impiden el acceso a los atributos no estáticos de la clase. Ejemplos:

```
public static Color getColorPorDefecto() {  
    return COLOR_POR_DEFECTO; ← Correcto, pues COLOR_POR_DEFECTO es  
                                estático  
}
```

```
public static double area(double radio) {  
    return Math.PI * radio * radio; ← Este método calcula el área de un  
                                     círculo dado un radio.  
                                     Ojo: no usa el atributo radio, sino el  
                                     radio que recibe como parámetro  
}
```

La versión no estática del método anterior sería: No requiere ningún parámetro pues usa
el atributo radio del objeto actual

```
public double area() {  
    return Math.PI * this.radio * this.radio;  
}
```

2.- OBJETOS Y CLASES

Métodos estáticos (2/2)

- Los métodos estáticos pueden invocarse directamente sobre la clase, pues no acceden a los atributos del objeto. Ejemplos:

```
double area = Circulo.area(10.0);
```

← Calcula el área de un círculo de radio 10. Es muy útil pues no hace falta crear ningún objeto para usar este método

Para conocer el área de un círculo de radio 10 mediante el método no estático es necesario crear un objeto de tipo *Circulo* con dicho radio:

```
Circulo c = new Circulo(10.0, Color.red);  
double area = c.area();
```

2.- OBJETOS Y CLASES

Resultado final: la clase *Circulo* (1/2)

```
package lasFiguras; // Paquete al que pertenece la clase Circulo  
import java.awt.*; // Para poder usar la clase Color  
  
public class Circulo {  
    // Atributos  
    private double radio;  
    private Color color;  
    private static final double RADIO_POR_DEFECTO = 3.0;  
    private static final Color COLOR_POR_DEFECTO = Color.black;  
    // Constructores  
    public Circulo(double radio, Color color) {  
        this.radio = radio;  
        this.color = color;  
    }  
    public Circulo() { this(RADIO_POR_DEFECTO, COLOR_POR_DEFECTO); }  
}
```

2.- OBJETOS Y CLASES

Resultado final: la clase *Circulo* (2/2)

```
// Consultores
public double getRadio() { return radio; }
public Color getColor() { return color; }
// Modificadores
public void setRadio(double radio) { this.radio = radio; }
public void setColor(Color color) { this.color = color; }
// toString() y equals()
public String toString() {
    return "Círculo de radio " + radio + " y color " + color;
}
public boolean equals(Object x) {
    Circulo cX = (Circulo) x;
    return (radio == cX.radio) && (color.equals(cX.color));
}
}
```

2.- OBJETOS Y CLASES

Ejercicio

- Ejercicio 3: Implementar una clase que permita gestionar un conjunto de círculos (como máximo 10 círculos)
 - Los círculos se guardarán en un vector (con tamaño máximo 10)
 - El constructor debe crear el vector vacío
 - Consultores: leer el número de círculos insertados y poder recuperar un círculo del vector dado su índice
 - Modificadores: insertar un círculo en el vector (si hay menos de 10)
 - Métodos *toString()* y *equals()*

3.- HERENCIA

El modelo ES UN(A)

- Si ahora se quisiera diseñar una clase que fuera una colección de rectángulos ¿Habría que crear una clase completamente nueva?
 - Mediante la herencia podemos tener una **clase base**, *Figura*, de la que se deriven varias **subclases** (*Circulo*, *Rectangulo*, etc.)
 - Las **clases derivadas** (*Circulo*, *Rectangulo*, etc.) heredan todo lo que no sea privado de la clase base
 - Puesto que un *Circulo* o un *Rectangulo* son *Figuras*, podemos definir una colección de *Figuras* que nos servirá para almacenar cualquier tipo de figura

3.- HERENCIA

Definición de la clase base

```
public class Figura {
    protected String tipo;
    protected Color color;
    protected static final Color COLOR_POR_DEFECTO = Color.black;
    public Figura(String tipo, Color color) {
        this.tipo = tipo; this.color = color;
    }
    public String toString() {
        return "Figura de tipo " + tipo + " y color " + color;
    }
    public boolean equals(Object x) {
        Figura fX = (Figura) x;
        return (tipo.equals(fX.tipo) && color.equals(fX.color));
    }
}
```

En la clase base los atributos suelen definirse como *protected* para que las clases derivadas puedan acceder a ellos

3.- HERENCIA

Definición de la clase derivada (1/2)

```
public class Circulo extends Figura {
    /* Atributos propios de Circulo */
    protected double radio;
    protected static final double RADIO_POR_DEFECTO = 3.0;

    /* Métodos propios de Circulo */
    public Circulo(Color color, double radio) {
        super("Círculo", color);
        this.radio = radio;
    }
}
```

Llamada al constructor de la clase base →
inicialización de los atributos de la clase base.
Si no se llama a *super* se hace una
llamada automática sin parámetros

3.- HERENCIA

Definición de la clase derivada (2/2)

```
/* Sobrescritura de métodos */
public String toString() {
    return "Círculo de radio " + radio + " y color " +
        color + "\n";
}
}
```

↓

```
/* Otra opción es sobrescribir parcialmente  
el método, invocando al método de la clase base: */
public String toString() {
    return super.toString() + " y radio " +
        radio + "\n";
}
```

3.- HERENCIA

Polimorfismo

- Una variable referencia es *polimórfica* cuando su tipo de declaración no coincide con el tipo del objeto al que referencia:

```
Figura f1 = new Figura("Estándar", Color.red);
```

↑
f1 es de tipo *Figura* (tipo estático) f1 apunta a un objeto de tipo *Figura* (tipo dinámico)

```
Figura f2 = new Circulo(Color.green, 5.0);
```

↑
f2 es de tipo *Figura* (tipo estático) f2 apunta a un objeto de tipo *Circulo* (tipo dinámico)

f2 es una variable polimórfica

3.- HERENCIA

Enlace dinámico

- Cuando se llama a un método que ha sido sobrescrito, se selecciona el método correspondiente al *tipo dinámico* de la variable:

```
Figura coleccion[] = new Figura[2];
coleccion[0] = new Figura("Estandar", Color.red);
coleccion[1] = new Circulo(Color.green, 5.0);
for (int i = 0; i < 2; i++) {
    System.out.println("Figura " + i + ": " +
        coleccion[i].toString());
}
```

- Resultado de la ejecución:

Figura 0: Figura de tipo Estandar y color java.awt.Color[r=255,g=0,b=0]

Figura 1: Círculo de radio 5.0 y color java.awt.Color[r=0,g=255,b=0]

3.- HERENCIA

Enlace dinámico

- El operador *instanceof* permite averiguar el tipo dinámico de una variable:

```
if (coleccion[0] instanceof Circulo) { ... }
```

- Para convertir una variable del tipo base (*Figura*) en una de un tipo derivado (*Circulo*) hay que hacer una conversión explícita (*casting*), ya que la herencia no lo permite:

```
if (coleccion[0] instanceof Circulo) {  
    Circulo c = (Circulo) coleccion[0];  
}
```

3.- HERENCIA

Ejercicio

- Ejercicio 4: Sean las siguientes clases:

```
public class Animal {  
    public void sonido() { System.out.println("Grunt"); }  
}  
public class Muflon extends Animal {  
    public void sonido() { System.out.println("MOOOO!"); }  
}  
public class Armadillo extends Animal {}
```

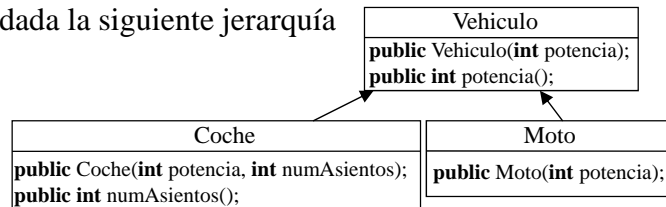
¿Qué instrucciones del siguiente programa no son correctas?

```
public class Test1Animal {  
    public static void main(String[] args) {  
        adoptaAnimal(new Armadillo());  
        Object o = new Armadillo();  
        Armadillo a1 = new Animal();  
        Armadillo a2 = new Muflon();  
    }  
    private static void adoptaAnimal(Animal a) { }  
}
```

3.- HERENCIA

Ejercicio

- **Ejercicio 5:** dada la siguiente jerarquía de clases:



Diseñar una clase **Garaje** que:

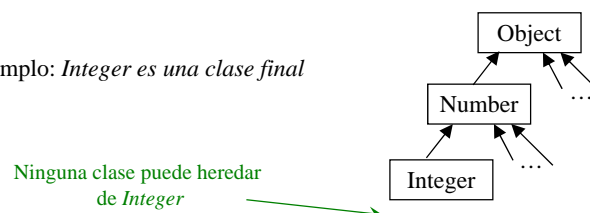
- En el constructor se indique el número total de plazas de garaje
- En cada plaza se pueda guardar tanto un coche como una moto
- Tenga una función que devuelva la cuota mensual de una plaza:
 - Si en dicha plaza hay un coche, la cuota se calcula como la potencia multiplicada por el número de asientos
 - Si en dicha plaza hay una moto, la cuota se calcula como la potencia multiplicada por 2
 - Si no hay ningún vehículo en la plaza, la cuota es 0

3.- HERENCIA

Atributos, métodos y clases finales

- La cláusula **final** se utiliza para:
 - Definir constantes (**atributos** cuyo valor no varía)
 - Especificar **métodos** que no puedan modificarse mediante herencia
 - Definir **clases** que no pueden ser modificadas mediante herencia
 - Las clases finales se ubican, por lo tanto, al final de la jerarquía de herencia

Ejemplo: *Integer es una clase final*



3.- HERENCIA

Métodos abstractos

- Un método **abstracto** es un método que no está implementado.

Ejemplo:

- En la clase *Figura* podemos incluir el método *area*, pues todas las figuras tienen un área

```
public abstract double area();
```

- Sin embargo, ¿Cómo se calcula el área de una figura? No es posible calcular el área de una figura hasta que no sepamos de qué tipo es (*Círculo*, *Rectángulo*, etc.)
- Un método **abstracto** se utiliza para definir un método común en todas las clases derivadas

3.- HERENCIA

Clases abstractas

- Si una clase tiene un método abstracto es una clase **abstracta**

- La clase *Figura* es, por lo tanto, una clase abstracta:

```
public abstract class Figura {  
    ...  
    public abstract double area();  
}
```

- No se pueden crear objetos de una clase abstracta
- Las clases derivadas (*Círculo*, *Rectángulo*, etc.) deben implementar (sobrescribir) los métodos abstractos de la clase base (*Figura*). En caso contrario serán también clases abstractas.

4.- INTERFACES

- Una interfaz es una clase en la que:
 - Todos sus métodos son todos públicos y abstractos
 - Todos sus atributos son públicos y finales
 - Una interfaz no tiene, por lo tanto, constructor ni puede ser instanciada
- ¿Para qué se utiliza?
 - Para permitir la herencia múltiple (ya que la herencia en Java sólo permite heredar de una sola clase)
 - Definen un comportamiento (o funcionalidad) genérico, ignorando los aspectos relacionados con su implementación

4.- INTERFACES

Ejemplo (1/3)

- Se quiere definir el comportamiento de un vehículo terrestre:

```
public interface VehiculoTerrestre {  
    int numRuedas();  
}  
  
public class Coche implements VehiculoTerrestre {  
    private int numeroDeRuedas;  
    public Coche(int numeroDeRuedas) {  
        this.numeroDeRuedas = numeroDeRuedas;  
    }  
    public int numRuedas() {  
        return numeroDeRuedas;  
    }  
}
```

Todas las clases que implementen la interfaz *VehiculoTerrestre* deberán sobrescribir el método *numRuedas*

La clase *Coche* implementa la interfaz *VehiculoTerrestre*

La clase *Coche* debe implementar, por lo tanto, el método *numRuedas*

4.- INTERFACES

Ejemplo (2/3)

- Se quiere definir el comportamiento de un vehículo acuático:

```
public interface VehiculoAcuatico {  
    double eslora();  
}
```

- Al igual que hemos hecho antes, podríamos definir una clase *Barco* que implementara la interfaz *VehiculoAcuatico*
- ¿Y si queremos definir una clase *Anfibio*, que es tanto un vehículo terrestre como acuático?

4.- INTERFACES

Ejemplo (3/3)

```
public class Anfibio implements VehiculoAcuatico, VehiculoTerrestre {  
    private int numeroDeRuedas;  
    private double longitudEslora;  
    public Anfibio(int numeroDeRuedas, double longitudEslora) {  
        this.numeroDeRuedas = numeroDeRuedas;  
        this.longitudEslora = longitudEslora;  
    }  
    public int numRuedas() {  
        return numeroDeRuedas;  
    }  
    public double eslora() {  
        return longitudEslora;  
    }  
}
```

EJERCICIOS

- Ejercicio 6: Corrige el código de las siguientes clases para que el método *mostrarReparto* funcione correctamente:

```
public abstract class Persona {
    private String nombre;
    public Persona(String nombre) { this.nombre = nombre; }
}
public class Actor extends Persona {
    private String pelicula;
    public Actor(String nombre, String pelicula) {
        this.nombre = nombre; this.pelicula = pelicula;
    }
}
public class Peliculas {
    public static void mostrarReparto(Actor lista[], String pelicula) {
        for (int i = 0; i < lista.length; i++)
            if (lista[i].pelicula == pelicula)
                System.out.println(lista[i].toString());
    }
}
```

EJERCICIOS

- Ejercicio 7: Diseñar la clase *OperacionesArray* que:
 - Tenga un método estático que busque un elemento en un *array* y devuelva su posición (devolverá -1 si el elemento no está en el *array*)
 - Tenga un método estático que permita eliminar un elemento dado de un *array*. Si el elemento no está no hace nada

Nota: se asume que todos los datos del *array* están al principio y el resto de posiciones (a partir del último dato válido) apuntan a *null*