

UT05.-INTRODUCCION A LA POO.

Índice

- 1.- Introducción a la orientación a objetos
 - 1.1.- Principios básicos de la orientación a objetos
 - 1.2.- Ventajas de la orientación a objetos.
 - 1.3.- Desventajas de la orientación a objetos.
 - 1.4.- Lenguajes orientados a objetos.
- 2.- Conceptos de orientación a objetos
 - 2.1.- Clases, atributos y métodos.
 - 2.2.- Control de acceso a los miembros de la clase.
 - 2.3.- Objetos. Estado, comportamiento e identidad.
 - 2.4.- Modificador static.
 - 2.5.- Constructores.
 - 2.6.- Inicialización de variables.
 - 2.7.- Almacenamiento de datos.
 - 2.8.- Modificador this
 - 2.9.- Sobrecarga de métodos.
 - 2.10.- Atributos finales
 - 2.11.- Argumentos de longitud variable
- 3.- Tipos enumerados
 - 3.1.- Características.
 - 3.2.- Métodos de java.lang.Enum.
 - 3.3.- Constructores, métodos, variables de instancia.
- 4.- Tipos Envoltorio o Wrappers
- 5.- Garbage Collector.

1. Introducción a la orientación a objetos.

Introducción.

- **Programación Estructurada**

- El programa se modela como un conjunto de procedimientos (acciones) que intercambian información → **Datos y lógica por separado.**
- Se centra en los **procedimientos**.

- **Programación Orientada a objetos.**

- Entiende un programa como un conjunto de **objetos** que tienen unas características determinadas e interaccionan entre sí a través de **mensajes** para producir un resultado.
- Simula los elementos de la realidad asociada al problema de la forma más cercana posible.
- Los **objetos** (no los procesos) **son la parte central del modelo**, así como los métodos de dichos objetos.
- El mantenimiento de los programas orientados a objetos, generalmente, es más fácil de realizar.

Principios básicos de la orientación a objetos

- ▣ **Una Clase es: Abstracción** de un conjunto de objetos → plantilla para crear objetos.
Formada por:
 - Conjunto de datos o atributos.
 - Conjunto de operaciones o métodos.
- ▣ **Un Objeto es: Encapsulación de datos** (propiedades o estado del objeto) y **métodos** (comportamiento del objeto).
 - **Mensaje.** Se envía sobre los objetos y son una orden de ejecución de una operación determinada sobre un objeto. Un mensaje está asociado a un método.

- ▣ **Una Aplicación orientada a objetos es:**

- Conjunto de **objetos** que interaccionan entre sí a través de **mensajes** para producir resultados.
- Los objetos similares se abstraen en **clases** → se dice que un objeto es una **instancia de una clase**.



Principios básicos de la orientación a objetos II

□ **Abstracción:**

- ▣ Captura características y comportamientos similares de un conjunto de objetos → **Clases**
- ▣ **Qué** operaciones puede hacer un objeto pero **no cómo** se han implementado las mismas.

□ **Encapsulación:**

- ▣ Significa agrupar todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción → **cohesión** de los componentes del sistema.
- ▣ **Ocultar ciertos detalles de los objetos** → separar el aspecto interno del externo de un objeto → ocultar los atributos y métodos de los objetos.

□ **Modularidad:**

- ▣ Subdividir una aplicación en partes más pequeñas (módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.
- ▣ En POO, **clase** \cong **módulo más básico del sistema**

□ **Cohesión:**

- ▣ En la POO las clases tendrán alta cohesión cuando se refieran a una única entidad. Podemos garantizar una fuerte cohesión disminuyendo al mínimo las responsabilidades de una clase: si una clase tiene muchas responsabilidades probablemente haya que dividirla en dos o más.
- ▣ A mayor cohesión, mejor: el módulo en cuestión será más sencillo de diseñar, programar, probar y mantener.

Principios básicos de la orientación a objetos III

- **Acoplamiento:** dependencia de una clase de otras. Lo ideal es reducir el acoplamiento → a menor acoplamiento, mejor.
- **Polimorfismo:**
 - ▣ Consiste en reunir bajo el mismo nombre comportamientos diferentes.
 - ▣ La selección del comportamiento dependerá del objeto que lo ejecute
- **Herencia:**
 - ▣ Relación que se establece entre objetos en los que unos utilizan las propiedades y comportamientos de otros formando una jerarquía. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen.
- **Recolección de basura:**
 - ▣ Destrucción automática de los objetos → desvinculación de la memoria asociada (Heap).

Principios básicos de la orientación a objetos IV

- Ejecución de una aplicación OO:
 - ▣ Creación de objetos a medida que se necesitan.
 - ▣ Los mensajes se mueven de un objeto a otro (o del usuario a un objeto).
 - ▣ Borrado de objetos cuando ya no se necesitan → liberación de la memoria → proceso automático en Java.

Ventajas de la programación orientada a objetos

- Desarrollo de software en:
 - ▣ **menos tiempo**
 - ▣ **con menos coste**
 - ▣ **mayor calidad** ya que se favorece la reutilización de código → código reusable en otras aplicaciones (clases).
 - ▣ Aumento de la calidad de los sistemas, haciéndolos más **extensibles** → facilidad para aumentar o modificar la funcionalidad de la aplicación.
- Facilidad de **modificación y mantenimiento del código** por la modularidad y encapsulación.
- Facilita la **extensibilidad** gracias a la modularidad y herencia.
- Adaptación al entorno y el cambio con **aplicaciones escalables** → propiedad para ampliar un sistema sin rehacer su diseño y sin disminuir su rendimiento.

Desventajas de la programación orientada a objetos

- ❑ **Complejidad** para adaptarse.
- ❑ Mayor complejidad a la hora de entender el flujo de datos (pérdida de linealidad).
- ❑ Mayor **cantidad de código** para proyectos pequeños (a la larga, por la reutilización, se ahorra código).
 - ❑ **Las ventajas de la OO se obtienen a largo plazo.**
- ❑ No todos los programas pueden ser modelados con exactitud por el modelo de objetos.
- ❑ Los objetos a menudo requieren una extensa **documentación**.
- ❑ Requiere **mayor concentración** en la toma de requerimientos, análisis y diseño del software.

Lenguajes OO

□ Años 80

- C++ (1985)
 - Extensión de C con características OO
 - Muy popular, ayudó a difundir la POO
- Eiffel (1985)
 - Bertrand Meyer
 - Lenguaje orientado a objetos puro
 - Fruto de un profundo estudio del paradigma OO
 - Lenguaje “teórico” poco utilizado

□ Años 90

- Java (1995)
 - Sun Microsystems □ Oracle (2009)
 - Lenguaje OO puro
 - Se popularizó por su uso en la web
 - No tiene nada que ver con Javascript
- C# (2000)
 - C Sharp Combina C++ y Java
 - Integrado en la plataforma .NET de Microsoft
- Python (versión 3 en 2008)
 - Multiplataforma y multiparadigma (OO, imperativo y funcional)

2. Conceptos de orientación a objetos

Clases, atributos y métodos.

- Una **clase** es un **tipo de datos complejo (estructurado)** definido por el usuario.
 - ▣ Define **cómo funciona** un determinado conjunto de objetos → **concepto estático**.
- Toda clase tiene un **nombre** y **está formada** por:
 - ▣ **Atributos:** conjunto de características asociadas a una clase.
 - Definen la estructura de datos.
 - Cada atributo se define por su **nombre** y su **tipo**, que puede ser simple o estructurado, como otra clase.
 - ▣ **Métodos:**
 - Un **método** es un procedimiento o función que se invoca para actuar sobre un objeto → se denomina **mensaje** a la llamada a un método de una clase sobre un objeto.
 - Son los encargados de acceder a los atributos de la clase.
 - **Constructores:** métodos especiales para inicializar correctamente los objetos.
 - El conjunto de mensajes a los que puede responder un objeto se le conoce como **protocolo del objeto**.
- Todos los objetos creados a partir de una clase **tendrán los mismos atributos y métodos**.
- **Propósito de la clase:** definir **abstracciones** y favorecer la **modularidad**.

Clases, atributos y métodos.

□ Sintaxis para definir una clase:

```
class NombreClase{  
    cuerpo de la clase //atributos y métodos  
}
```

□ Atributos:

- Los atributos constituyen la **estructura interna** de los objetos de la clase.
- También se les denomina **variables miembro**.
- Los atributos se declaran como cualquier variable y tendrán un tipo de datos simple o complejo, como un objeto (en este caso, se dice que la variable es una **referencia a un objeto**).

```
double radio; //Declaramos una variable  
double radio = 1; //Declaramos una variable y la inicializamos  
Persona p;      //Declaramos la referencia al objeto Persona  
                //pero no estamos creando el objeto
```

Clases, atributos y métodos.

□ Métodos de una clase:

- Definen el comportamiento de los objetos y las operaciones a realizar sobre los atributos de la clase.

□ Definición de un método:

```
[modificador_acceso] <tipo_retorno> <nombreMetodo> ([<param1,param2, ...>]) {  
    cuerpo del método  
    [return valor_de_tipo_retorno;]  
}
```

(*) **tipo de retorno** será void si el método no devuelve nada, o el tipo de dato del elemento a devolver y, en ese caso debe existir la sentencia return dentro del método y se devolverá un valor de ese tipo.

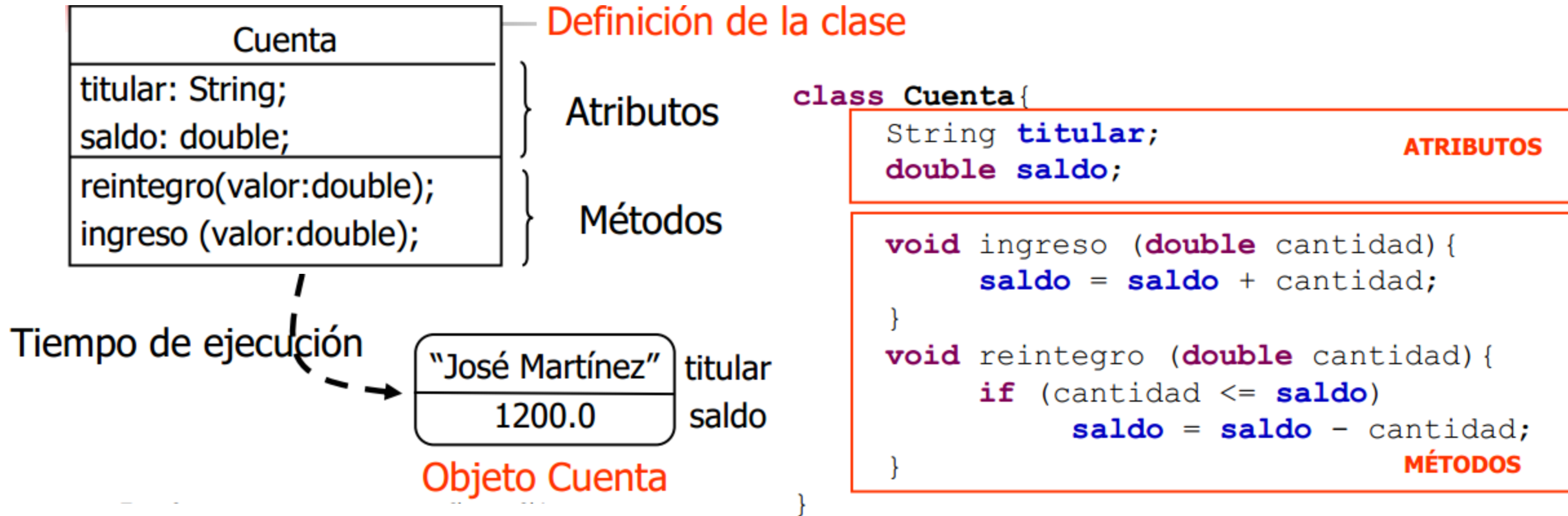
(**) **Los parámetros** en Java se **pasan siempre por valor** (en otros L.P se pueden pasar por valor o por referencia)

(***) Los parámetros son tratados como variables locales al método.

- Los métodos públicos de una clase forman lo que se denomina **Interfaz de la clase**.

Ejemplo.

□ Ejemplo: Clase Cuenta



□ Invocar un método:

c1.reintegro(200) ; //suponiendo un objeto de tipo Cuenta llamado c1

Clases, atributos y métodos.

□ Acceso a los atributos y métodos de la clase. Ejemplo:

```
class Circulo {  
    double radio;  
    double x,y;  
    double area(){  
        return Math.PI*radio*radio;  
    }  
}
```

Dentro de los métodos de la clase se tiene acceso a los atributos de la misma de forma directa

Se está accediendo al valor del atributo **radio** del objeto unCirculo. Pero veremos más adelante que evitaremos acceder así, desde fuera de la clase, a los atributos directamente

```
class AppGeometria {  
    public static void main(String[] args){  
        Circulo unCirculo = new Circulo();  
        unCirculo.radio = 4; //Acceso al atributo  
        System.out.print("El área es" +unCirculo.area()); //Acceso al método  
    }  
}
```

Ejercicio de clase.

- Crea la clase Cuenta con los atributos y métodos sugeridos en la diapositiva anterior.

Ejercicio de clase. SOLUCIÓN

```
class Cuenta {  
    double saldo;  
    String titular;  
  
    void ingreso (double cantidad) {  
  
        saldo = saldo + cantidad;  
    }  
    void reintegro(double cantidad) {  
  
        if (cantidad <= saldo)  
            saldo = saldo - cantidad;  
    }  
}
```

- El fichero debe llamarse cómo la clase: Cuenta.java

Control de acceso a los miembros de la clase

□ Principio de ocultación:

- Pretende **aislar las propiedades de un objeto** (atributos) y evitar su modificación directamente.
- El estado del objeto (sus atributos) solamente se podrá modificar mediante las operaciones definidas dentro de la clase para tal fin.
 - **Reduce la propagación de efectos colaterales** cuando se producen cambios, evitando generar código basado en la estructura interna de la clase. Esto es importante para los consumidores de bibliotecas de código.
 - **Separa la interfaz de la implementación.**
- El **concepto de clase incluye la idea de ocultación de los datos**, consistente en que no se pueda acceder a los atributos de la clase directamente, sino que haya que hacerlo a través de sus métodos.

Control de acceso a los miembros de la clase II

- Para controlar el **acceso a los miembros** de la clase Java define los siguientes **modificadores de acceso**, que definen distintos niveles de **visibilidad** para atributos y métodos:
 - ▣ **public (público)**: Accesible a cualquier clase de cualquier paquete (utilizando import)
 - ▣ **protected (protegido)**: Accesible dentro de la propia clase o desde clases derivadas (se verá más detalladamente cuando estudiemos la herencia).
 - ▣ **Cuando no se especifica nada**, decimos que el **acceso es predeterminado**. A este tipo de acceso también se le denomina acceso **“friendly”**, **default** o **de package**: en este caso, el elemento es accesible dentro de la propia clase, desde clases del mismo paquete y desde clases (o subclases) pertenecientes al mismo paquete.
 - ▣ **private (privado)**: Accesible únicamente dentro de la propia clase.

Control de acceso a los miembros de la clase III

```
public class Fecha {  
    private int anyo;  
    private int mes;  
    private int dia;  
  
    public getDia(){  
        return this.dia;  
    }  
    protected boolean esBisiesto(){  
        return ((anyo % 4==0) && (anyo % 100!=0) || (anyo % 400==0));  
    }  
    private boolean esCorrecta(){  
        ...  
    }  
}
```

Control de acceso a los miembros de la clase IV

- **Norma general**
 - ▣ Atributos (estado) → acceso private (Principio de ocultación)
 - ▣ Métodos (comportamiento) → acceso public
 - ▣ Otros métodos auxiliares → private o protected
- Una clase se divide en **dos partes**:
 - ▣ **Interfaz**: parte externa de una clase (**cómo se usa la clase**) formada por la colección de métodos públicos → los métodos private de una clase no forman parte de la interfaz.
 - ▣ **Implementación**: parte interna de la clase → representación de la abstracción, y mecanismos que conducen al comportamiento deseado (**qué hace la clase**).
- Una clase es un TDA que separa la interfaz de la implementación.

Ejercicio de clase.

- Modifica la clase Cuenta añadiendo modificadores de visibilidad a métodos y atributos.

Ejercicio de clase. SOLUCIÓN

```
class Cuenta {  
    private double saldo;  
    private String titular;  
  
    public void ingreso (double cantidad) {  
        saldo = saldo + cantidad;  
    }  
    public void reintegro(double cantidad) {  
        if (cantidad <= saldo)  
            saldo = saldo - cantidad;  
    }  
}
```

NOTA:

NO confundir los atributos de la clase (**variables miembro**) con las variables que se declaran dentro de un método (**variables locales**) que solo existen dentro del método. Las variables miembro existen mientras exista el objeto.

Control de acceso a los miembros de la clase V

□ **Métodos get y set:**

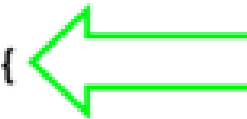
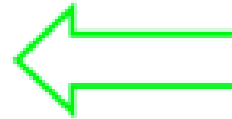
- Se definen para dar acceso a los atributos de la clase declarados como privados:
 - Si un atributo puede ser consultado se define un método de acceso (método get)
 - Si un atributo puede ser modificado se define un método de modificación (método set)
- El programador decide el nivel de acceso que proporciona a los atributos: ninguno, set, get, get/set
- Aislamos al cliente de los cambios en la estructura de los datos.

Ejercicio de clase.

- Modifica la clase Cuenta añadiendo métodos set/get para sus atributos.

Ejercicio de clase. SOLUCIÓN

```
class Cuenta {  
  
    private double saldo;  
    private String titular;  
  
    ...  
  
    public double getSaldo() {  
        return saldo;  
    }  
  
    public String getTitular() {  
        return titular;  
    }  
}
```



Control de acceso a la clase

- El **control de acceso a una clase** determina la relación que tiene una clase con otras clases de otros paquetes.
- Distinguimos dos niveles de acceso para clases:
 - ▣ **public**: una clase con este nivel de acceso es accesible a cualquier clase de cualquier paquete (desde otro paquete habrá que usar import)
 - ▣ **predeterinado o friendly o package**: una clase con este nivel de acceso solo puede ser utilizada por otras clases de su mismo paquete (ni siquiera subpaquetes).

```
package unpaquete;  
public class Persona {  
    //atributos  
    //métodos  
}
```

```
package unpaquete;  
class Persona {  
    //atributos  
    //métodos  
}
```

- ▣ Las clases con nivel de acceso private y protected se usan en clases internas que veremos más adelante.

Control de acceso a la clase II

- Dentro de un fichero fuente .java **puede haber más de una clase** definida **PERO solo una** de esas clases será **public**.
- El nombre del **fichero .java** deberá coincidir con el **nombre de la clase** public definida en el fichero.
- La **ejecución del programa** parte de la clase que tenga el método **main** que será la clase declarada como public.
- Al compilar el fichero fuente .java se generarán un fichero .class (bytecodes) para cada clase definida en el fichero .java
- Para agrupar todos los componentes de una aplicación, los ficheros .java por un lado y los .class por otro, se utiliza el concepto de **package**.

Control de acceso a la clase III: package

- Las clases se organizan en packages.
- Una **biblioteca** de clases es un conjunto de clases relacionadas.
- La **instrucción package** debe aparecer como primera línea del archivo fuente.
 - Indica a qué biblioteca de clases o paquete pertenece la clase. Esto permite agrupar código relacionado y facilita la modularidad.
 - Si una clase no tiene declaración de package se dice que pertenece al **paquete por defecto** (raíz de la unidad).
- Quien quiera utilizar una clase deberá importar el paquete donde se encuentra o emplear el nombre completo de la clase (nombre paquete + nombre clase)
- Los **nombres de los paquetes van en minúscula**, incluyendo las palabras intermedias. La primera parte del nombre del paquete es el dominio de internet invertido (se utilizan los nombres de dominio de internet, ya que son únicos). Si no disponemos de nombre de dominio tendremos que crear combinaciones de **nombres únicos**.

Control de acceso a la clase VI: package

- La división de clases en paquetes permite dividir el espacio de nombres para evitar colisiones de nombres.
- Un paquete puede contener otros paquetes. Separamos los nombres de paquetes y subpaquetes por ‘.’
- Los archivos de un paquete se ubican en un directorio.
- Ejemplo:

```
package com.mipaquete.utilidad;    <- Nombre del paquete al que pertenece la clase
    public class MiClase{
        ...
    }
```

Ahora MiClase se llama *com.mipaquete.utilidad.MiClase*.

Control de acceso a la clase V: package

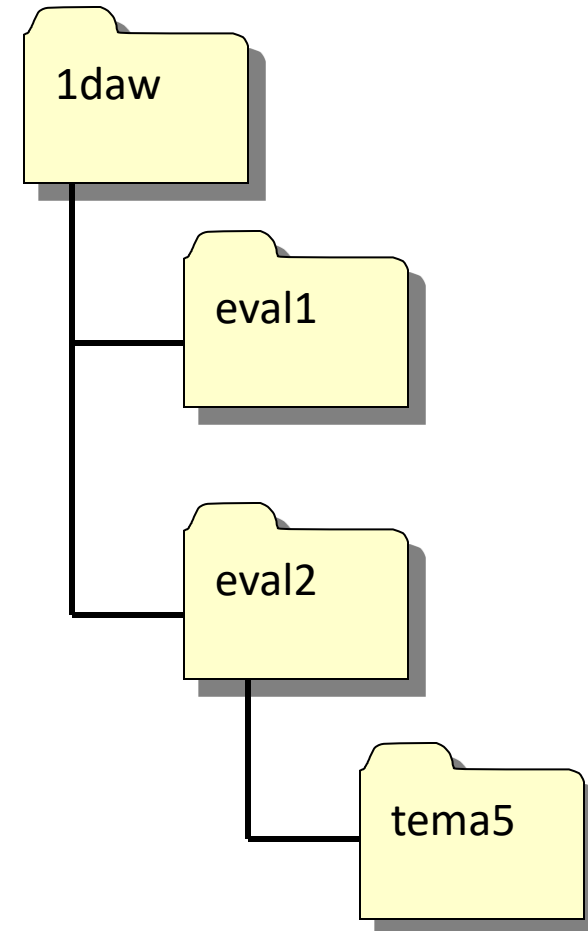
□ Ejemplo:

```
package 1daw;
```

```
package 1daw.eval1;  
class Uno;  
class Dos;
```

```
package 1daw.eval2;
```

```
package 1daw.eval2.tema5;  
class Uno;  
class Dos;
```



Control de acceso a la clase VI: package e import

- Para usar una clase que pertenece a otro paquete utilizamos la palabra **import**
- Al usar **import** podemos omitir la ruta de paquetes al nombrar una clase.
- Ejemplo:

```
import java.util.Scanner;  
public class MiClase{  
    ...  
}
```

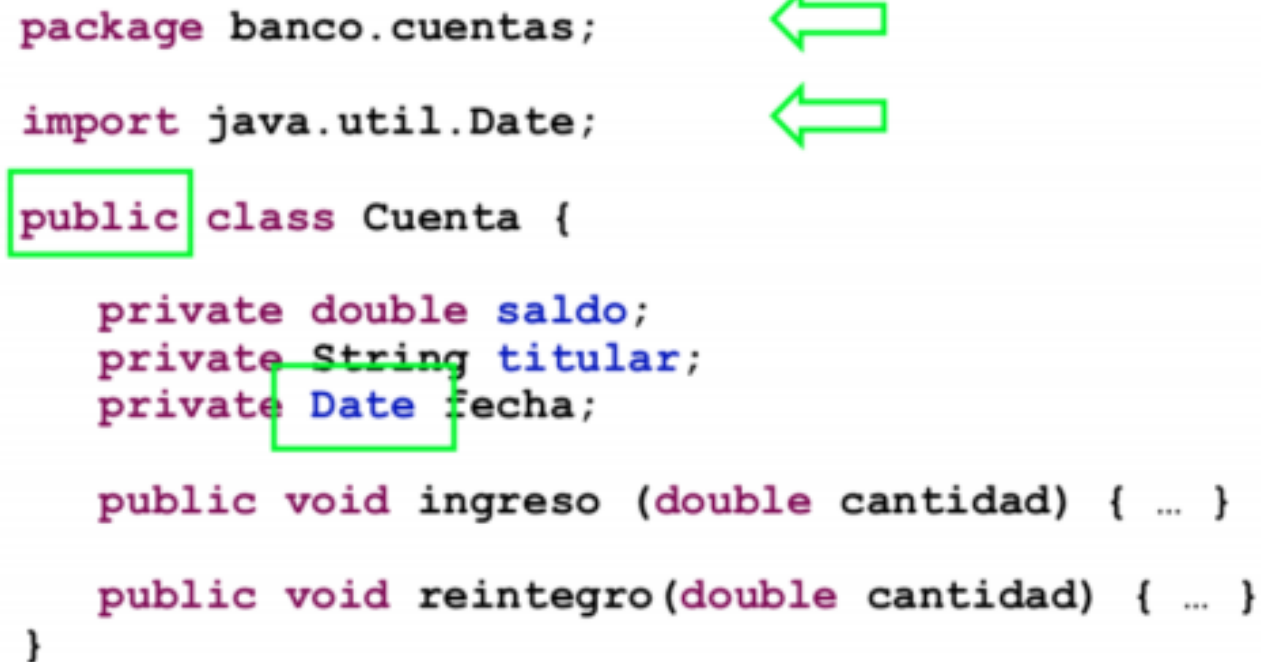
Control de acceso a la clase VII: package e import

- Una clase con nivel de acceso paquete sólo puede ser utilizada por las clases del mismo paquete (ni siquiera subpaquetes).
- Una clase pública puede ser utilizada por cualquier otra clase de otro paquete.

```
package banco.cuentas;
import java.util.Date;
public class Cuenta {
    private double saldo;
    private String titular;
    private Date fecha;

    public void ingreso (double cantidad) { ... }

    public void reintegro(double cantidad) { ... }
}
```



Clases y objetos

Programa OO

Colección estructurada
de clases

Clase

Implementación de un
Tipo Abstracto de Datos (TAD)

Objeto

Una instancia de una clase

El código Java se organiza en clases

Los objetos se comunican mediante **mensajes**



id: número de bastidor

Funciones que puede realizar:

- Ir
- Parar
- Girar a la derecha
- Girar a la izquierda
- Arrancar

Tiene las **características**:

- Color
- Velocidad
- Tamaño
- Carburante

Objetos: estado, comportamiento e identidad.

- Un **objeto** es un ente dinámico → existen en tiempo de ejecución y **ocupan memoria (heap)**.
- Un objeto debe ser creado:
 - **Instanciando** una clase → Un objeto es una **instancia de una clase** creada en tiempo de ejecución. Se les maneja mediante variables referencia → **el identificador de un objeto es una referencia al mismo**.
- Un objeto se define por:
 - **Su estado**: definido por el conjunto de valores de sus atributos (propiedades). Inicializado por el **Constructor**.
 - **Su comportamiento**: definido por los métodos públicos de su clase (tareas que puede realizar en tiempo de ejecución)
 - **Su tiempo de vida**: intervalo de tiempo a lo largo del programa en el que el objeto existe, desde su creación (**instanciación**) hasta la destrucción del objeto.
 - **Identidad**: es la propiedad de un objeto que lo lleva a distinguirse de otros.

Objetos: estado, comportamiento e identidad.

□ Declaración y construcción:

- La declaración de una variable cuyo tipo sea una clase no implica la creación del objeto.
 - Cuenta cuenta1 Declara la variable “cuenta1” que es una referencia a un objeto de la clase Cuenta. La referencia “cuenta1” no está inicializada → Si enviamos un mensaje a cuenta1 obtendremos un error en tiempo de ejecución.
 - ¡ Las referencias almacenan la dirección de memoria donde se aloja el objeto !
 - ¡ Los tipos primitivos almacenan directamente el valor !
- Los objetos se crean con el **operador new** que invoca al constructor de la clase.
 - Cuenta cuenta1 = **new** Cuenta(); Declara una referencia a un objeto de la clase Cuenta e instancia o crea el objeto. Ahora la variable cuenta1 está inicializada y apunta a un objeto de la clase Cuenta.
Se ha invocado al constructor “Cuenta()” que inicializa el objeto
 - Cuenta cuenta2 = cuenta1; Declara una variable cuenta2 que hace referencia al objeto que hemos creado antes. **!!! No se crea un nuevo objeto cuenta !!!**

Objetos: estado, comportamiento e identidad.

- Todos los objetos de una misma clase tienen los mismos elementos (métodos y atributos) pero pueden tener distintos valores en sus atributos (distinto estado).
- Por lo general para utilizar cualquier elemento de una clase (método o atributo) se necesita construir un objeto de la clase, salvo que el elemento esté declarado como **static**.
- Ejemplo donde podemos ver qué es eso de la identidad de un objeto:

```
Persona juan = new Persona("Juan", "Martinez Ruiz", 33, "52525246A");
Persona otroJuan= new Persona("Juan","Martinez Ruiz",33,"52525246A"); //los datos son idénticos
if(juan.equals(otroJuan))
    System.out.println("Son iguales ...");
else System.out.println("No son el mismo objeto aunque sus atributos sean iguales");
```

Modificador static

- ❑ **static** es un modificador de no acceso de Java que se aplica a: **bloques de código, variables, métodos y clases anidadas**.
- ❑ Cuando un miembro de una clase se declara estático (**static**), se puede acceder a él sin crear un objeto de su clase y sin referenciar a ningún objeto.
- ❑ El ejemplo más común de un miembro estático es `main()`, que se declara como estático porque la JVM debe llamarlo cuando comienza el programa. Otro ejemplo, el método `random` de la clase `Math`.
- ❑ **Fuera de la clase**, para usar un miembro estático, solo necesita especificar el nombre de la clase seguido por el operador punto y el nombre del miembro estático.

Modificador static

❑ Variables estáticas (static):

- ❑ Esencialmente son variables globales.
- ❑ Una variable o atributo static es una **variable de clase** (no de instancia, no del objeto).
- ❑ **Todas las instancias de la clase comparten la misma variable static**; en cambio cada objeto de una clase tiene su propia copia de las variables declaradas no static.
Declaramos: **static int varX;** Accedemos a ella: **NombreClase.varX;**
- ❑ Las variables static nos servirán para crear **constantes**, por ello será habitual encontrar
final static double PI = 3.1416;

❑ Métodos estáticos (static):

- ❑ Los métodos static se invocan utilizando el nombre de la clase, sin crear previamente un objeto de la misma.
- ❑ Por ejemplo, el método random() de la clase Math es estático: **Math.random()**
- ❑ Los métodos declarados como **static** tienen varias **restricciones**:
 - ❑ No pueden acceder a un miembro (atributos o métodos) no static de su clase.
 - ❑ Sí pueden acceder a miembros static de su clase.
 - ❑ Sí pueden ser accedidos desde otros métodos independientemente de que sean o no static.
 - ❑ Ellos no tienen una referencia **this**.

Ejemplo

❑ Supongamos el siguiente ejemplo:

❑ La salida del programa es:

```
C:\WINDOWS\system32\cmd.exe
C:\workspace>java Demo
Value of a = 0
Value of b = 1
Value of a = 0
Value of b = 2
```

```
public class Ejemplo {

    public static void main(String[] args) {
        Estudiante s1 = new Estudiante();
        s1.showData();
        Estudiante s2 = new Estudiante();
        s2.showData();
    }

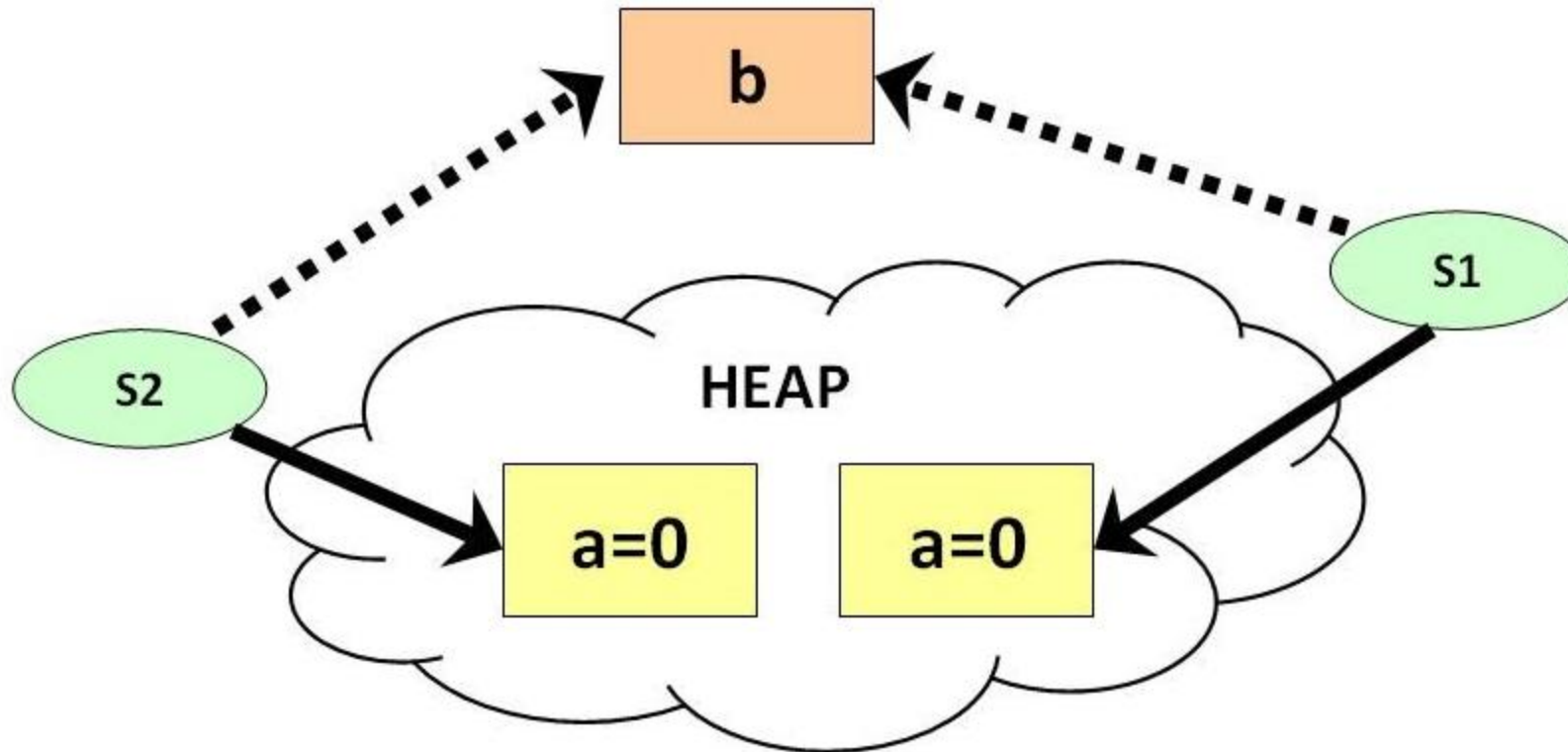
}

class Estudiante{
    int a; //variable de instancia
    static int b; //variable de clase
    /* Al ser variables miembro serán inicializadas a 0
    por el compilador. Una cuando se crea el objeto, otra
    al cargarse la clase */
    Estudiante(){ //Constructor
        b++;
    }
    public void showData(){
        System.out.println("Valor de a "+a);
        System.out.println("Valor de b "+b);
    }

}
```

Ejemplo

- La siguiente imagen muestra cómo se crean las variables de referencia y los objetos y las distintas instancias acceden a las variables estáticas.



En el ejemplo anterior se ha creado dos objetos que tienen una variable de instancia de nombre *a*, por tanto cada uno tendrá su propia variable *a*. Ambos objetos comparten una variable de clase (static) *b*.

Otro ejemplo

```
7 public class EjemploCirculo {
8     public static void main(String[] args) {
9         Circulo c1 = new Circulo(5.0F);
10        Circulo c2 = new Circulo(4.0F);
11        c1.ver();
12        c2.ver();
13        //c1.color = "Amarillo"; //No nos deja hacer esto
14        Circulo.color = "Amarillo"; //Cambiamos el color para todos
15        c2.ver();
16    }
17 }
18 class Circulo{
19     static String color="Negro";
20     float radio;
21     Circulo(float r){
22         radio = r;
23     }
24     void dibujar(){
25         this.ver();
26     }
27     void ver(){
28         System.out.println("Estamos viendo el objeto y sus propiedades son "+
29                             " color "+color + " radio "+this.radio);
30     }
31 }
```

Acceso a miembros static desde fuera de la clase Circulo.
Las variables static son compartidas

Dentro de la clase accedemos de igual forma a las variables miembro, sean static o no.
Con las variables static no podemos usar this.
Los métodos static solo pueden acceder a variables static

Ejercicio de clase.

- Modifica la clase Cuenta añadiendo un atributo static para el tipo de cuenta (Ahorro, Corriente, Nómina, por ejemplo) que compartirán todos los objetos Cuenta.
- Añade métodos set/get para este atributo.

Ejercicio de clase. Solución.

```
class Cuenta{  
    private String titular;  
    private double saldo;  
    private static String tipo;  
  
    public static void setTipo(String tipo){  
        Cuenta.tipo = tipo;  
    }  
    public static String getTipo(){  
        return tipo;  
    }  
}
```

Ejercicio propuesto.

- ❑ Crea un atributo de clase llamado “ultimoCodigo” que contenga el último código de cuenta asignado
- ❑ Crea un nuevo atributo “codigo” para la clase Cuenta que será el código único para cada objeto de tipo Cuenta
- ❑ Crea un método de clase llamado “getCodigoCuenta()” que devuelva el siguiente código de cuenta que podemos utilizar
- ❑ Finalmente realiza una prueba de creación de cuentas comprobando que se van generando los código de cuenta automáticamente

Constructores

- ❑ **Método especial** que se llama igual que la clase.
- ❑ **No devuelve nada, ni siquiera void.**
- ❑ Puede tener o no parámetros.
- ❑ Es **invocado automáticamente** por el compilador (JVM) siempre que se crea un objeto de una determinada clase.
- ❑ **Un constructor se encarga de la correcta inicialización de los objetos antes de su uso.**
- ❑ Todas las clases deben tener **al menos un constructor PERO**
- ❑ Si el programador no declara ningún constructor, el compilador incluye **un constructor por defecto (Ver diapositiva siguiente)**
- ❑ Se denomina **Constructor por defecto** al constructor sin parámetros o aquel que define automáticamente el compilador en caso de que no exista uno definido por el programador.

Constructores

- ❑ En el ejemplo de nuestra clase Cuenta no se ha declarado ningún constructor, sin embargo podemos crear objetos Cuenta. Eso es debido a que el compilador proporciona un constructor por defecto.
- ❑ **Ejemplo:**

```
public static void main(String[] args) {  
    Cuenta c1 = new Cuenta(); //Constructor por defecto|  
    System.out.println("Saldo "+c1.getSaldo());  
}
```

Constructores II

- ❑ **Suelen declararse public para poder ser invocados desde cualquier parte.**
- ❑ Podemos declarar varios constructores en la misma clase: tendrán el mismo nombre pero distinto número, tipo u orden de argumentos (**Sobrecarga de métodos**) → **en este caso si queremos disponer del Constructor por defecto deberemos declararlo explícitamente.**
- ❑ Puede haber **varios constructores**, pero el compilador solo invocará a uno en el momento de la creación de un objeto.
- ❑ **El constructor no se hereda** (veremos el significado de esto más adelante)
- ❑ **Desde el cuerpo del constructor se puede/debe asignar valores a sus atributos**, invocar a otros métodos, etc.
- ❑ Los constructores sobrecargados no se pueden distinguir en base al tipo de retorno nunca, siempre será en base a los parámetros.

Constructores III

- ❑ La construcción de un objeto consta de **tres etapas**:
 1. **Se reserva en memoria espacio para la estructura de datos** que define la clase
 2. Se realiza la **inicialización de los campos** con los valores iniciales o con los valores por defecto asociados a su tipo de datos (*)
 3. Se realiza la **llamada al constructor** que finaliza la inicialización.
- ❑ (*) Inicialización de los campos por defecto: (cuando no les damos valor explícitamente)
 - ❑ Tipos numéricos → a valor 0
 - ❑ Carácter → carácter null o '\u000'
 - ❑ Boolean → false
 - ❑ Objetos → null
- ❑ Podemos establecer el **valor de inicialización en la declaración de un atributo.**

Ejercicio de clase.

1. Crea una clase "Principal.java" dentro del paquete donde tienes la clase "Cuenta.java".
2. La clase "Principal.java" tendrá un método "main" para que se ejecute por defecto.
3. Dentro de ese método main debes crear varios objetos Cuenta y comprobar los valores de los atributos a través de sus métodos. Verás que han sido inicializados a un valor por defecto.
4. A continuación, añade un constructor para Cuenta que reciba como parámetros el titular y el saldo inicial.
5. Crea un objeto Cuenta utilizando el nuevo constructor.
6. Comprueba los valores de los atributos tras este último cambio.
7. Sobrecarga el constructor y pruébalo.

Ejercicio de clase. Solución

```
class Cuenta{  
    private String titular;  
    private double saldo;  
    private static String tipo;  
    Cuenta() {  
    }  
    Cuenta (String titular, double saldo){  
        this.titular = titular;  
        this.saldo = saldo;  
    }  
    Cuenta (String titular, double saldo,String tipo){  
        this.titular = titular;  
        this.saldo = saldo;  
        Cuenta.tipo = tipo;  
    }  
    ...  
}
```

```
public static void main(String[] args) {  
    Cuenta c1 = new Cuenta(); //Constructor por defecto  
    System.out.println("Saldo "+c1.getSaldo());  
    Cuenta c2 = new Cuenta("Pepe López",2000);  
    System.out.println("Saldo "+c2.getSaldo());  
    ...  
}
```

Inicialización de variables.

- Tipos de variables:
 - **Atendiendo a su ámbito:**
 - **Variables miembro:** son las declaradas dentro de la clase pero fuera de todo método.
 - Su ámbito es toda la clase.
 - Pueden ser **static**: en ese caso la variable pertenece a la clase, no al objeto, es decir, la variable static será compartida por todos los objetos de la clase.
 - **Variables locales a un método:** declaradas dentro de un método. Su ámbito es el método, fuera de él no son accesibles.
 - **Atendiendo a su tipo de dato:**
 - Tipos simples: int, long, double, boolean, char ...
 - Tipos compuestos como clases → **variables referencia**
- Cuando se define una variable, el compilador reserva memoria para su ubicación, le asigna un valor y procederá a su destrucción cuando el flujo de ejecución vaya fuera del ámbito de la variable.
- Cuando se instancia un objeto, el compilador invoca al constructor de la Clase, el cual inicia a valores por defecto cada variable si no se especifica un valor concreto en el constructor. Los constructores se invocan recursivamente .
- El compilador igualmente invoca al destructor de un objeto para realizar las tareas de limpieza y liberar recursos.

Inicialización de variables.

- ❑ Las **variables locales** deben ser inicializadas por el programador u obtendremos un error de compilación.
- ❑ Las **variables miembro** sino son inicializadas por el programador serán inicializadas por el compilador a los **valores por defecto**:
 - ❑ Variables numéricas son inicializadas a 0
 - ❑ Variables carácter, las inicializa a carácter null o '\u000' o \0
 - ❑ Variables boolean a False
 - ❑ Las referencias a objetos son inicializadas a null
- ❑ El **orden de inicialización** es de arriba abajo, primero se inicializan las variables (se inicializa primero todos los elementos **static** (campos y código) y después el resto de **campos** en orden de aparición a sus valores por defecto o al valor indicado) y después se ejecutan los constructores. En este orden de inicialización la máquina virtual de Java primero carga la clase y después instancia los objetos.

Inicialización de variables static

- ❑ Las **variables static** son inicializadas **solo una vez al inicio de la ejecución del programa**, cuando la clase es cargada por primera vez, mientras que los atributos no static son inicializados para cada objeto en el instante de su creación.
- ❑ Estas variables se inicializarán primero, antes de la inicialización de cualquier variable de instancia.
- ❑ Normalmente se utiliza el constructor de la clase para inicializar las variables, pero **si la clase tiene atributos static** el constructor solo es adecuado cuando la inicialización de esos atributos no sea requerida antes de que se cree un primer objeto. Si no es así será necesario añadir un **iniciador estático** (bloque) cuya sintaxis es:

```
static{  
    //inicialización de variables static  
}
```

- ❑ Un iniciador static es un método anónimo sin parámetros, no retorna ningún valor, y es invocado automáticamente por el sistema cuando se carga la clase por primera vez.
- ❑ En un bloque static solo se puede inicializar variables static, no variables de instancia.

Inicialización de variables. Orden de inicialización.

- ❑ En este ejemplo, hay dos variables: a y b. Cuando se instancia un objeto, PRIMERO se ejecutan los inicializadores de arriba a abajo, y DESPUÉS el constructor.

```
1  class A {  
2      int a = 0;  
3      public A() {  
4          System.out.println("valor de b:"+b);  
5      }  
6      char b = 'X';  
7  }
```

- ❑ Por ello, cuando se instancia este objeto, mostrará:

```
1  valor de b:X
```

- ❑ No importa si el inicializador está después de un constructor.

Inicialización de variables. Bloques de inicialización.

❑ Bloques de inicialización:

- ❑ Otra forma de inicializar las propiedades de un objeto es usando el bloque de inicialización.
- ❑ Se trata de un bloque sin nombre.
- ❑ Cuando instanciamos este objeto, el resultado será.

```
1 valor de c:25
```

```
1  class B {  
2      int a = 15;  
3      int b = 10;  
4      int c;  
5      public B() {  
6          System.out.println("Valor de c:" + c);  
7      }  
8  
9      { //bloque de inicialización  
10         c = a + b;  
11     }  
12 }
```

- ❑ Recordemos que los inicializadores se ejecutan de ARRIBA A ABAJO, y ANTES del Constructor.
- ❑ Por tanto, el orden de los inicializadores es importante.

Inicialización de variables. Bloques de inicialización.

- ❑ Por ejemplo, si modificamos la clase B de la siguiente manera: **Se produce un error!!**

```
1  class B {  
2      int a = 15;  
3      int c;  
4      public B() {  
5          System.out.println("Valor de c:" + c);  
6      }  
7      { //bloque de inicialización  
8          c = a + b; //error de compilación  
9      }  
10     int b = 10;  
11 }
```

- ❑ Además, puede haber varios bloques de inicialización. Y cumplen la misma regla: Se ejecutan de ARRIBA a ABAJO.

Inicializadores de clase.

- ❑ Inicializadores estáticos o de clase:
 - ❑ Así como hay miembros (propiedades y métodos) de clase [static], también existen los inicializadores *static*.
 - ❑ Cumplen las mismas reglas, pero **se ejecutan solo cuando la clase se carga en la memoria del JVM.**

Primero se ejecutarán los inicializadores de clase (01), los de instancia (02) y al final el constructor.

```
1  class C {  
2  
3      static int a; //1  
4      int b; //2  
5  
6      public C() {  
7          System.out.printf("Valor de a:%1s\nValor de b:%2s\n", a, b);  
8      }  
9  
10  
11      { //2  
12          b = a * 2;  
13      }  
14  
15  
16      static { //1  
17          a = 15;  
18      }  
19  }
```

Almacenamiento de datos.

- ❑ La JVM divide la memoria en zonas:
 - ❑ Zona de pila o Stack
 - ❑ Zona Montón o Heap
 - ❑ Zona estática o static
 - ❑ Zona para código (bytecodes)
- ❑ **Memoria de Pila o STACK:**
 - ❑ Memoria ubicada en la RAM. El procesador apunta a esta memoria mediante el puntero de pila, el cual es movido hacia abajo para crear una nueva variable o nuevo espacio de memoria y hacia arriba para liberar ese espacio. La memoria de pila siempre se referencia en el orden de último en entrar primero en salir.
 - ❑ Memoria más rápida y eficiente que el Heap.
 - ❑ En la pila se almacena las **variables locales** y **variables de referencia** (No son objetos, apuntan a ellos)
 - ❑ En cuanto finalice el método invocado , las referencias a objetos, las variables y todo lo relacionado con el método desaparece de la pila.

Almacenamiento de datos

❑ Memoria montón o HEAP:

- ❑ Zona única de memoria, también en la RAM y contiene instancias de objetos.
- ❑ Es un espacio de memoria dinámico, es decir, aumenta de tamaño según la necesidad.
- ❑ También puede contener variables de referencia a otros objetos. Las variables de instancia se crean en el montón.
- ❑ A diferencia de la Pila el compilador no necesita saber cuánto espacio necesita en el Heap ni por cuánto tiempo. Por el contrario, es más costoso en tiempo almacenar en el Heap que en la Pila.
- ❑ Mientras que cada thread en la JVM tiene un Stack privado, el Heap es un espacio de memoria dinámica único que se crea al inicio de la máquina virtual.
- ❑ El administrador del Heap es el **Garbage Collector** o sistema de administración de almacenamiento automático.

❑ Memoria estática o static:

- ❑ Zona de memoria estática, también en la RAM y contiene elementos declarados como static.
- ❑ Datos que están disponibles en cualquier momento de la ejecución de un programa.

[Ampliar información](#)

this

```
public class Ejemplos{
    static void main(String[] args){
        Circulo c = new Circulo(5);
        System.out.println(
            "Diametro "+c.calculaDiametro());
    }
}

class Circulo{
    float radio;
    float diametro;
    Circulo(float radio) {
        this.radio = radio; //(1)
    }
    float calculaDiametro(){
        diametro = radio*2;
        return this.diametro; //(2)
    }
}
```

- ❑ Hace referencia al objeto actual.
- ❑ this solo puede ser usado dentro de un método.
- ❑ Recuerda que siempre que se llama a un método éste se ejecuta sobre un determinado objeto. El objeto que recibe el mensaje para ejecutar un determinado método es el objeto actual.
- ❑ **Ejemplos de uso de this:**
 - ❑ (1) Para referirse a las variables de instancia del objeto actual: **se usa para resolver ambigüedades**
 - ❑ (2) Devolver el valor de una propiedad

this

```
class Circulo{
    float radio;
    float diametro;
    Circulo (){
        this(5,10); // (3)
        System.out.println("En el constructor
        por defecto");
    }
    → Circulo(float radio,float diametro){
        this.radio = radio;
        this.diametro = radio*2;
        System.out.println("En el constructor
        parametrizado");
    }
    //Método que devuelve la instancia de la clase actual
    Circulo get(){
        return this; // (4)
    }
}
```

■ Ejemplos de uso de this:

- (3) Invocar desde un constructor a otro constructor: debe ser la primera instrucción y solo se puede invocar a uno de los constructores desde otro.
- (4) Devolver el objeto actual

this

```
class Circulo{
    float radio;
    float diametro;
    void dibujar(){
        this.ver(); // (5)
    }
    void ver(){
        System.out.println("Viendo el objeto");
    }
}

public class UsaCirculo{
    public static void main(String[] args){
        Circulo c = new Circulo();
        c.dibujar();
    }
}
```

❑ Ejemplos de uso de this:

- ❑ (5) Invocar a un método sobre el objeto actual

Sobrecarga de métodos

- Se conoce como sobrecarga de métodos a la posibilidad de declarar dentro de una clase diferentes métodos con el mismo nombre pero distinta lista de parámetros (distinto número de parámetros o distinto tipo de dato, orden de los parámetros ...)

- Ejemplo:

```
int metodo1();  
int metodo1(int x);  
int metodo1(float x);  
int metodo1(int x, int y);  
int metodo1(int x, float y);
```

- La sobrecarga de un método no se puede basar únicamente en el tipo de retorno de un método:

```
int metodo1(int x);  
int metodo1(float x);
```

} válido

```
int metodo1(int x);  
void metodo1(float x);
```

} válido

```
int metodo1(int x);  
void metodo1(int x);
```

} NO válido

Atributos finales.

- ❑ Cuando una variable se declara con la palabra clave **final**, su valor no se puede modificar. Se utiliza para declarar variables constantes.
- ❑ Las constantes no se consideran atributos, no tiene sentido crearles métodos set y get.
- ❑ Las variables o atributos finales deben ser inicializados en la declaración o en un constructor.
- ❑ Es una buena práctica representar las variables finales en **mayúsculas**, utilizando guiones bajos para separar las palabras.
- ❑ Si la variable final es una referencia, esto significa que la variable no se puede volver a vincular para hacer referencia a otro objeto, pero el estado interno del objeto apuntado por esa variable de referencia se puede cambiar, es decir, puede agregar o eliminar elementos de la matriz final o colección final, etc.
- ❑ Ejemplos:

`final int LIMITE=5;`

`final int LIMITE;`

`static final double PI = 3.141592653589793;`

Atributos finales.

❑ Ejemplo:

```
public class Cuenta {  
    private double saldo ;  
    private final String titular;  
    public Cuenta(String nombre) {  
        titular = nombre;  
    }  
    ...  
}
```

Argumentos de tamaño variable

- ❑ A veces podemos necesitar que un método reciba un número variable de argumentos. En ese caso podemos sobrecargar el método o pasarle un array.
- ❑ A partir del J2SE 5.0 (JDK 5) se incluyó una característica llamada **argumentos de longitud variable** o **varargs**.
- ❑ Un método que toma una cantidad variable de argumentos se denomina **método varargs**.
- ❑ La lista de parámetros para un método o constructor varargs no es fija, sino que tiene una longitud variable, es decir, puede tomar una cantidad variable de argumentos.
- ❑ **Sintaxis:** Un argumento de longitud variable se especifica por tres puntos (...)
- ❑ Dentro del método el argumento se utiliza como un array.
- ❑ Solo puede haber un argumento de este tipo en el método y se debe colocar al final de la lista de argumentos.

Argumentos de tamaño variable

□ Ejemplo:

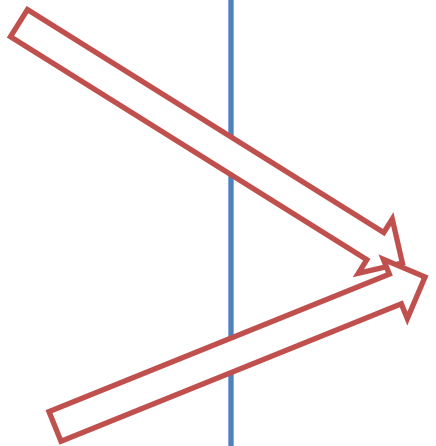
```
public class Ejemplo {  
    public static void main(String ... args) {  
        metodo(20,21,22,23);  
    }  
    public static void metodo (int ... v) {  
        for(int i=0;i<v.length;i++) {  
            System.out.println("v "+v[i]);  
        }  
    }  
}
```

Ejercicio de clase.

- Modifica la clase Cuenta y sobrecarga los métodos ingreso y reintegro en métodos **varargs**, es decir, que admitan parámetros de tamaño variable. En el método ingreso sigue comprobando que no se pueda reintegrar una cantidad que supere el saldo de la cuenta.

Ejercicio de clase. Solución.

```
public void ingreso (double... cantidad){
    for (int i=0;i<cantidad.length;i++){
        this.saldo += cantidad[i];
    }
}
public void reintegro(double... cantidad){
    for (int i=0;i<cantidad.length;i++){
        if(cantidad[i]<=this.saldo){
            this.saldo -= cantidad[i];
        }else this.saldo=0; // esta es una opción
    }
}
```



```
Cuenta c = new Cuenta();
c.ingreso(200,300,400,500);
c.reintegro(100,200,300);
c.ingreso(); //permitido!!
```


3. Tipos enumerados

Tipo enumerado

- ❑ Los **enumerados** permiten definir una lista de valores constantes que definen un nuevo tipo de datos.
- ❑ Un enumerado es una clase especial que limitan la creación de objetos a los especificados. Ejemplos: lista de planetas, meses del año, días de la semana, ...
- ❑ Los tipos **enum** son útiles siempre que necesite definir un **conjunto de valores que represente una colección fija de elementos**.
- ❑ En **Java** el tipo enumerado se define con **enum**.

```
// Un simple ejemplo donde se declara enum
// fuera de cualquier clase (Nota la palabra enum en lugar
// de la palabra class)
enum Color
{
    ROJO, VERDE, AZUL;
}

public class Test
{
    // El método
    public static void main(String[] args)
    {
        Color c1 = Color.ROJO;
        System.out.println(c1);
    }
}
```

- ❑ La primera línea dentro de **enum** debe ser una lista de **constantes** y luego otras cosas como métodos, variables y constructores.
- ❑ De acuerdo con las convenciones de nomenclatura de Java, se recomienda que nombremos las constante con mayúsculas.

Tipo enumerado

- Las constantes que se definen dentro de un enum se denominan **constantes de enumeración**:
 - Cada una se declara implícitamente como un miembro público (**public**), estático (**static**), constante (**final**) .
 - El tipo de las constantes de enumeración es el tipo del enumerado, por lo tanto, en Java, estas constantes se llaman **auto-tipado**.
 - **Cada constante enum representa un objeto de tipo enum** → esto es así porque **Java implementa las enumeraciones como tipos de clases** → las constantes de enumeración son implementadas internamente como **objetos**.
 - Podemos agregarle variables, métodos y constructores.
 - Se pueden definir dentro de una clase o fuera, pero no dentro de un método.
 - Un objeto de tipo **enum** únicamente puede contener un valor definido en la lista.

Tipo enumerado

□ Internamente:

```
/* internamente enum Color se convierte en  
class Color  
{  
    public static final Color ROJO = new Color();  
    public static final Color AZUL = new Color();  
    public static final Color VERDE = new Color();  
}*/
```

- Una vez definido un enumerado podemos crear una variable de ese tipo y se declaran y se utilizan **como si fuera un tipo primitivo**.
- Se pueden comparar dos constantes de enumeración utilizando el operador relacional ==.
- Un valor de enumeración también se puede usar para controlar una sentencia switch → todas las declaraciones de case deben usar constantes de la misma enumeración que la utilizada por la expresión de switch.

Tipo enumerado

- Ejemplo de uso de switch:

```
enum Color
{
    ROJO, VERDE, AZUL;
}

public class Enumerados {
    public static void main(String ... args) {
        Color c1 = Color.ROJO;
        System.out.println("Valor de c1 "+c1);
        switch(c1) {
            case ROJO:
                System.out.println("Soy Rojo");
                break;
            case VERDE:
                System.out.println("Soy Verde");
                break;
            case AZUL:
                System.out.println("Soy Azul");
                break;
        }
    }
}
```

Ejercicio de clase.

- ❑ Define un tipo enumerado llamado **EstadoCuenta** para reflejar los siguientes estados en los que puede estar una Cuenta: operativa, cerrada, inmovilizada, numeros_rojos.
- ❑ Añade un atributo **estado** a Cuenta de este tipo enumerado.
- ❑ En el Constructor/es de la clase se deberá inicializar este nuevo atributo a estado operativa.
- ❑ Define métodos set/get para este nuevo atributo.
- ❑ Desde el programa principal crea una Cuenta, modifica su estado y muestra en pantalla ambos estados, antes y después de su modificación.

Ejercicio de clase. Solución

```
public enum EstadoCuenta {  
    OPERATIVA, INMOVILIZADA, NUMEROS_ROJOS;  
}  
  
public class Cuenta {  
    ...  
    private EstadoCuenta estado;  
  
    public Cuenta(Persona persona) {  
        estado = EstadoCuenta.OPERATIVA;  
        ...  
    }  
}
```

```
public EstadoCuenta getEstadoCuenta(){  
    return this.estado;  
}  
  
public void setEstadoCuenta(EstadoCuenta e){  
    this.estado = e;  
}
```

```
Cuenta c1 = new Cuenta();  
System.out.println(c.getEstadoCuenta());  
c.setEstadoCuenta(EstadoCuenta.NUMEROS_ROJOS);  
System.out.println(c.getEstadoCuenta());  
|
```

Tipo enumerado. Métodos de java.lang.Enum

- El hecho de que Java defina los enumerados como una clase permite que se le puedan añadir constructores, variables, métodos, etc.
- Los enumerados heredan de **java.lang.Enum** y por tanto disponen de los siguientes métodos heredados:
 - El método implícito **values**: *public static enumConstant [] values()* -> devuelve un array con todos los valores definidos dentro del tipo enumeado. Ej. `Color array[] = Color.values();`
 - El método **ordinal**: *public final int ordinal()* -> devuelve un entero con la posición de la constante según ha sido declarada. Ej. `uncolor.ordinal();`
 - El método **valueOf**: *public static enumConstant valueOf(String s)* -> devuelve la constante enum que coincide con el valor dado. Ej. `Color.valueOf("ROJO");`
 - **toString**: *public String toString()* -> devuelve el nombre de la constante enum. Ej. `uncolor.toString();`
 - **compareTo**: *public final int compareTo(Enum other)* -> permite comparar dos constantes de la misma enumeración. Ej. `unColor.compareTo(otroColor);`

Tipo enumerado. Métodos de java.lang.Enum

```
// Programa Java para demostrar el funcionamiento de values(),
// ordinal() y valueOf()
enum Color
{
    ROJO, VERDE, AZUL;
}

public class Test
{
    public static void main(String[] args)
    {
        // Llamando a values()
        Color arr[] = Color.values();

        // enum con bucle
        for (Color col : arr)
        {
            // Llamando a ordinal() para encontrar el índice
            // de color.
            System.out.println(col + " en el índice "
                               + col.ordinal());
        }

        // Usando valueOf(). Devuelve un objeto de
        // Color con la constante dada.
        // La segunda línea comentada causa la excepción
        // IllegalArgumentException
        System.out.println(Color.valueOf("ROJO"));
        // System.out.println(Color.valueOf("BLANCO"));
    }
}
```

Bucle **for-each** (JDK5): permite recorrer una colección o array de objetos evitando el uso de iteradores.

Sintaxis:

for(TipoBase obj: arrayTipoBase){

...

//obj permite el acceso a cada elemento
}

Tipo enumerado. Constructores, métodos, variables de instancia.

- Dado que cada constante de enumeración es un objeto, podemos definir:
 - **Constructor:** cuando se define un constructor, éste será llamado una vez para cada constante de enumeración en el momento de la carga de la clase enum.
 - **El constructor debe ser privado para que** No podamos crear objetos enum explícitamente → no se puede invocar al constructor directamente.
 - Podremos sobrecargar el constructor, como en cualquier clase.
 - **Variables de instancia:** cada constante de enumeración tiene su propia copia de cualquier variable de instancia definida en la enumeración.
 - **Métodos de instancia:** cada constante de enumeración puede llamar a cualquier método definido en la enumeración.

Tipo enumerado. Constructores, métodos, variables de instancia.

```
//Uso de un constructor, una variable de instancia y un método.
enum Transporte{
    COCHE(60), CAMION(50), AVION(600), TREN(70), BARCO(20);
    private int velocidad; //velocidad típica de cada transporte

    //Añadir un constructor
    Transporte(int s){velocidad=s;}
    //Añadir un método
    int getVelocidad(){return velocidad;}
}

class Enumerados {
    public static void main(String[] args) {
        Transporte tp;
        //Mostrar la velocidad de un avión
        System.out.println("La velocidad típica para un avión es: "+
            Transporte.AVION.getVelocidad()+" millas por hora.\n");

        //Mostrar todas las velocidades y transportes
        System.out.println("Todas las velocidades de transporte: ");
        for (Transporte t:Transporte.values())
            System.out.println(t+": velocidad típica es "+t.getVelocidad()+" millas por hora.");
    }
}
```

El **constructor** se llama una vez para cada constante enum. Es preferible que se declare privado.

Los argumentos para el constructor se especifican entre paréntesis después de cada constante.

4. Clases Envoltorio o Wrappers

Tipos envoltorio o Wrappers Class

- En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, etc.) como objetos.
- Para ello Java incorpora las **clases envoltorio** que dotan a los tipos primitivos un envoltorio que permita tratarlos como objetos:

Tipo primitivo	Clase envoltorio
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Char

- Estas clases proporcionan métodos que permiten administrar estos tipos de datos primitivos.

Tipos envoltorio o Wrappers Class

- ❑ La siguiente tabla muestra un resumen de los métodos disponibles para la clase Integer:

Método	Descripción
<code>Integer(int valor)</code> <code>Integer(String valor)</code>	Constructores a partir de int y String
<code>int intValue() / byte byteValue() / float floatValue() .</code>	Devuelve el valor en distintos formatos, int, long, float, etc.
<code>boolean equals(Object obj)</code>	Devuelve true si el objeto con el que se compara es un Integer y su valor es el mismo.
<code>static Integer getInteger(String s)</code>	Devuelve un Integer a partir de una cadena de caracteres. Estático
<code>static int parseInt(String s)</code>	Devuelve un int a partir de un String. Estático.
<code>static String toBinaryString(int i)</code> <code>static String toOctalString(int i)</code> <code>static String toHexString(int i)</code> <code>static String toString(int i)</code>	Convierte un entero a su representación en String en binario, octal, hexadecimal, etc. Estáticos
<code>String toString()</code>	
<code>static Integer valueOf(String s)</code>	Devuelve un Integer a partir de un String. Estático.

- ❑ El API de Java contiene una descripción completa de todas las clases envoltorio en el package java.lang.

5. Garbage Collector

Garbage Collector.

- ❑ El Heap es un espacio de memoria dinámico, es decir, aumenta de tamaño según sea necesario. Cuando el Heap se queda sin espacio, existen dos posibilidades, o bien aumentar el tamaño del mismo, o bien liberar espacio, y es en esta última solución donde aparecen los recolectores de basura o Garbage Collector (GB).
- ❑ El GC es una aplicación que se ejecuta dentro de la JVM y se encarga de gestionar el Heap, liberando automáticamente la memoria ocupada por aquellos objetos que no van a ser utilizados en un programa.
- ❑ Cuando un objeto ya no está en uso por el programa es eliminado de memoria por el garbage collector. Si bien, antes de ser eliminado de la memoria, el garbage collector finaliza el objeto ejecutando el método `finalize()`.
- ❑ El método `finalize()` permite ejecutar las acciones pertinentes sobre el objeto antes de que sea totalmente eliminado. Por ejemplo, podemos pensar en casos como gestión de ficheros o bases de datos, en los cuales podremos cerrar las conexiones sobre dichos recursos.
- ❑ Más adelante veremos cómo sobrescribir este método para ejecutar las acciones que estimemos necesarias.

Garbage Collector.

Ventajas de uso del GC:

- ❑ Se quita la responsabilidad de liberar y destruir los objetos a los programadores.
- ❑ Garantiza la interoperabilidad entre distintas APIs, librerías o frameworks.
- ❑ Facilita el uso de grandes cantidades de memoria.

Desventajas podemos encontrar:

- ❑ Consumo adicional de recursos.
- ❑ Impacto en el rendimiento de aplicaciones.
- ❑ Posibles paradas en la ejecución de las aplicaciones.
- ❑ Incompatibilidad con la gestión manual de la memoria.