

Java Inicial

(20 horas)





Temario

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. **POO en Java**
5. Relaciones entre Objetos
6. Polimorfismo, abstracción e interfaces
7. Excepciones
8. Conceptos avanzados



Tema 4

Programación Orientada a Objetos (POO) en Java



Objetivos

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. **POO en Java**
5. Relaciones entre Objetos
6. Polimorfismo, abstracción e interfaces
7. Excepciones
8. Conceptos avanzados

- Introducción
- Elementos de la POO
- Estructura de Clase
 - Visibilidad
 - Declaración atributos y métodos
 - Argumento Valor/Referencia
 - Puntero This
 - Sobrecarga
- Métodos
 - Constructores
 - De Acceso
 - Comportamiento
 - Destrucción

■ Introducción

- Viene a solventar los problemas de la TAD
 - Estructuras de Datos
 - Operaciones para modificar dichos TADs
- Modelar más fielmente la realidad
- ‘POO = TAD + Herencia + Interacción’
- Propiedades
 - Abstracción: Definir el qué pero no el cómo
 - Encapsulación: Hacer público lo que queremos
 - Reutilización: No re-implementar
 - Modularización: Divide y vencerás. Acoplamiento

■ Elementos de la POO

- Clase: Idea. Definición y declaración de la estructura de datos y operaciones. Genérica.
- Objeto: Instancia de la clase. Idea Concreta.
- Atributo: Propiedades, variables de la clase.
- Método: Operaciones de la clase para trabajar con los atributos.
- Estado: Conjunto de valores de los atributos de un objeto.

■ Objetos

- Entidades que combinan estado, comportamiento e identidad.
 - El **estado**: compuesto de datos, será uno o varios atributos a los que se habrán asignado unos valores concretos (datos).
 - El **comportamiento**: Definido por los procedimientos o métodos con que puede operar dicho objeto, es decir, qué operaciones se pueden realizar con él.

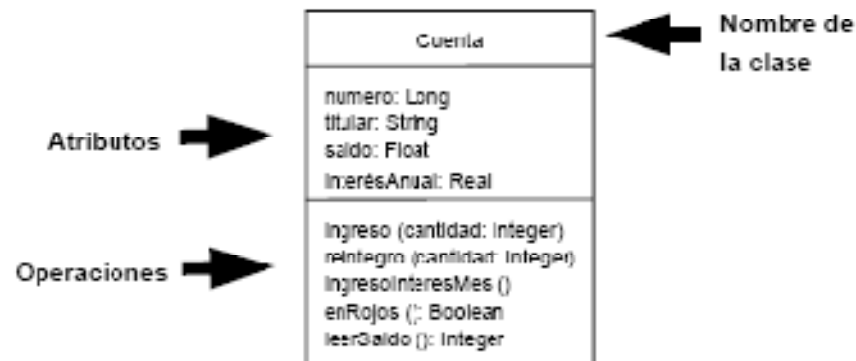
■ Programa POO

- Conjunto de objetos, que colaboran entre ellos para realizar tareas que los hace más fáciles de *escribir, mantener y reutilizar*.
- Un objeto contiene toda la información para definirlo e identificarlo exclusivamente
 - Respecto a objetos de otras clases.
 - Incluso respecto a objetos de su misma clase.
- La comunicación entre objetos → *Métodos*
 - La comunicación favorece el cambio de estado.

■ Clase de Objetos

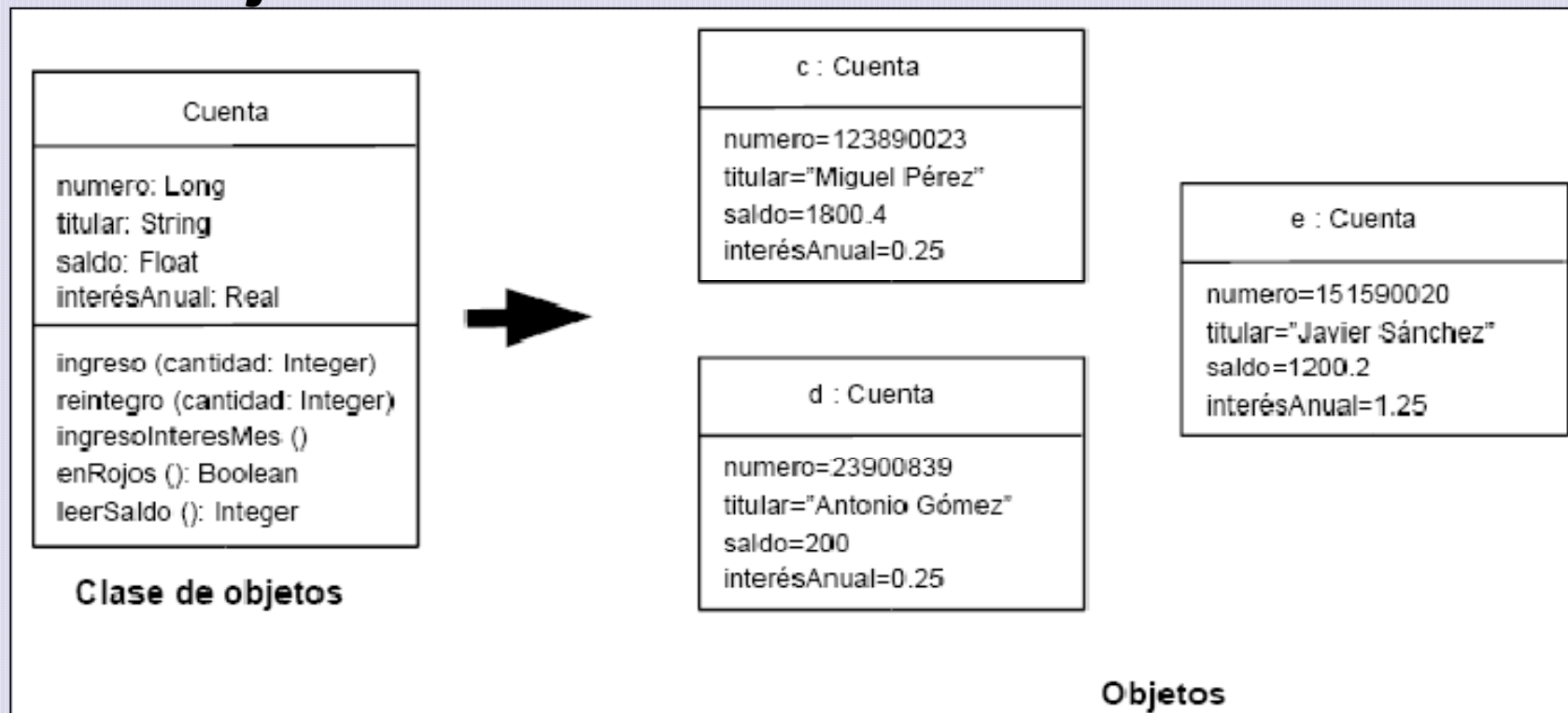
- Las clases de objetos son los elementos básicos de la programación orientada a objetos y representan conceptos o entidades significativos de un problema determinado.
- Una clase viene descrita por 2 elementos:

- + **Atributos (o variables de clase)**. Describen el estado Interno de cada objeto
- + **Operaciones (o métodos)**. Describen lo que se puede hacer con el objeto, los servicios que proporciona



■ Objetos

- Una clase de objetos describe las características comunes a un conjunto de **objetos**.



■ Estructura de una clase en Java

- La sintaxis sencilla de una clase es:

```
class NombreClase
{
    // definicion de atributos
    // definición e implementación de métodos
}
```

- Definición Atributos:
 - Todas las propiedades de la clase.
- Definición y/o implementación de Métodos
 - Todas las operaciones de la clase.
- Regla de estilo
 - ¡Nombre clase primera letra en Mayúscula!
 - ***class Cuenta***

■ Ejemplo *Clase Cuenta*

- La implementación de la clase en Java se realizaría con un fichero *Cuenta.java*

Atributos →

```
class Cuenta {  
    long numero;  
    String titular;  
    float saldo;  
    float interesAnual;  
}
```

- Los atributos pueden ser de cualquiera de los tipos básicos de Java: **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** y **double**, referencias a otros objetos o arrays de elementos de alguno de los tipos citados

■ Ejemplo *Clase Cuenta*

- La implementación de las operaciones se realiza en el interior de la definición de la clase, justo tras su declaración.

```
class Cuenta {  
    long numero;  
    String titular;  
    float saldo;  
    float interesAnual;  
  
    void ingreso (float cantidad) {  
        saldo += cantidad;  
    }  
  
    void reintegro (float cantidad) {  
        saldo -= cantidad;  
    }  
  
    void ingresoInteresMes () {  
        saldo += interesAnual * saldo / 1200;  
    }  
  
    boolean enRojos () { return saldo < 0; }  
    float leerSaldo () { return saldo; }  
}
```

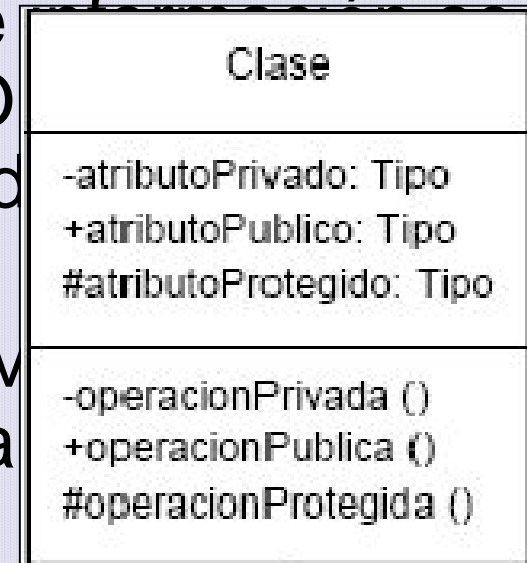
- Ejemplo *workspace*
 - Package `com.slashmobility.oo1`: clases `PruebaTiempo1.java` y `Tiempo1.java`.
 - Contiene un ejemplo de cómo abstraer el concepto de Reloj.

■ Protección de Miembros

- El principio de ocultación de información en el mundo del plasma en los lenguajes OO y los mecanismos de protección de miembros de la clase.

- UML permite asociar tres niveles de protección diferentes a cada miembro de una clase:

- **públicos (+)**. Sin ningún tipo de protección
- **privados (-)**. Inaccesibles desde el exterior de la clase
- **protegidos (#)**. Similares a los privados aunque se permite su acceso desde las clases descendientes



■ Protección de miembros

- En Java un miembro se etiqueta como público colocando el identificador **public** delante de su declaración
- Para los miembros privados utilizaremos el identificador **private**

Cuenta
numero: Long -titular: String -saldo: Float -InterésAnual: Real
+Ingreso (cantidad: Integer) +reIntegro (cantidad: Integer) +IngresoInteresMes () +enRojos (): Boolean +leerSaldo (): Integer

```

class Cuenta {
    private long numero;
    private String titular;
    private float saldo;
    private float interesAnual;

    public void ingreso (float cantidad) {
        saldo += cantidad;
    }

    public void reintegro (float cantidad) {
        saldo -= cantidad;
    }

    public void ingresoInteresMes () {
        saldo += interesAnual * saldo / 1200;
    }

    public boolean enRojos () { return saldo < 0; }
    public float leerSaldo () { return saldo; }
}

```

- Visibilidad de los atributos y métodos
 - Importante para la encapsulación.
 - Evitar el alto acoplamiento.

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, <i>through inheritance</i>	No	No
From any non-subclass class outside the package	Yes	No	No	No

■ Visibilidad 'amiga' o de paquete

```
package prueba;      package prueba;      class D {
class A {            class C {            } ...
    ...              } ...
} ...
class B {            } ...
    ...              class E {
} ...                } ...

```

Las clases A, B y C son amigas al pertenecer al mismo paquete "prueba"

Las clases D y E son amigas al pertenecer al mismo fichero fuente

- Un fichero fuente java forma en sí un paquete y por tanto todas las clases incluidas en él son amigas
- Las clases incluidas en varios ficheros fuente pueden agruparse en un único paquete indicando el nombre de paquete al principio de cada fichero mediante el indicador **package**

■ Visibilidad 'amiga' o de paquete

- En este ejemplo, las clases *Cuenta* y *Banco* son amigas al pertenecer al mismo fichero fuente. El acceso a los atributos de los objetos de la clase *Cuenta* almacenados en el vector interno de *Banco* queda así garantizado. El atributo *saldo* puede mantenerse como privado puesto que existe una operación que permite obtener su valor

```
class Cuenta {
    long numero;
    String titular;
    private float saldo;
    float interesAnual;

    public void ingreso (float cantidad) {
        saldo += cantidad;
    }

    public void reintegro (float cantidad) {
        saldo -= cantidad;
    }

    public void ingresoInteresMes () {
        saldo += interesAnual * saldo / 1200;
    }

    public boolean enRojos () { return saldo < 0; }
    public float leerSaldo () { return saldo; }
}

class Banco {
    Cuenta[] c; // vector de cuentas
    ...
}
```

■ Protección de Clases

- Por protección de clases entendemos un nivel superior de la ocultación de información, a nivel de clases.
- Especificar que clases pueden ser utilizadas y cuales no, y por quién.
- Dentro de un paquete, las clases son amigas y por tanto no existen restricciones respecto a la utilización de una clase por las otras
- Sin embargo, desde el punto de vista del exterior, únicamente podrán ser utilizadas las clases públicas del paquete, (***public***)

■ Protección de Clases: *Ejemplo*



Sólo una clase pública por fichero fuente

- *En nuestro ejemplo, si queremos que la clase Cuenta pueda ser utilizada desde el exterior del fichero **Cuenta.java** deberemos declararla como pública*

```
public class Cuenta {  
    long numero;  
    String titular;  
    private float saldo;  
    float interesAnual;  
  
    public void ingreso (float cantidad) {  
        saldo += cantidad;  
    }  
  
    public void reintegro (float cantidad) {  
        saldo -= cantidad;  
    }  
  
    public void ingresoInteresMes () {  
        saldo += interesAnual * saldo / 1200;  
    }  
  
    public boolean enRojos () { return saldo < 0; }  
    public float leerSaldo () { return saldo; }  
}
```

■ Declaración de Atributos

```
class NombreClase
{
    ambito tipo identificador;
    // definición e implementación de métodos
}
```

- **Ámbito:** Visibilidad del atributo.
 - **Tipo:** Tipo de datos del atributo.
 - **Identificador:** Nombre identificativo del atributo.
-
- Regla de estilo
 - Debe comenzar por minúscula [a..z][A..Z]
 - Ejemplo: *edad, nombre, fechaNacimiento*

■ Declaración de Métodos

```
class NombreClase
{
    ambito tipo nombreMetodo ( Lista de argumentos )
    {
        sentencia1;
        ...
        sentenciaN;
    }
}
```

- **Ámbito:** Visibilidad del método.
- **Tipo:** Tipo de datos que retorna el método.
- **nombreMetodo:** Identificador del método.
- **Lista de argumentos:** el tipo y el nombre de cada uno de los argumentos que recibe la función.
- Regla de estilo
 - El nombre del método en minúscula
 - acelerar(), sacarDinero()

■ Ejercicio: modelado de universidad

- Declarar una clase Estudiante que abstraiga el concepto de estudiante.
- Debe encapsular los siguientes conceptos:
 - Nombre, edad, dni, telefono, notas de 10 asignaturas, finalizado o no de las 10 asignaturas.
 - Operaciones para saber si el estudiante ha finalizado o no una asignatura concreta, saber la nota de una asignatura concreta y la nota media (sumar las asignaturas finalizadas / num total)
- Declarar la clase Universidad, que abstraiga el concepto de universidad
- Debe encapsular los siguientes conceptos:
 - Contener estudiantes (clases Estudiante)
 - Operaciones para consultar cuantos estudiantes tiene, la nota media de la universidad (media de los estudiantes / número de estudiantes).

■ Ejemplo workspace:

- En `com.slashmobility.PruebaPunto` se puede ver cómo abstraer el concepto de punto.
- En `com.slashmobility.PruebaCirculo` se muestra un ejemplo de abstracción del concepto de `Circulo`.

- Métodos: Procedimientos vs Función
 - **Procedimiento:** conjunto de sentencias agrupadas por un nombre (nombre del procedimiento) que realizar una serie de tareas (la ejecución de cada una de las sentencias) y devuelve un tipo **void**.
 - **Función:** Similar a un procedimiento excepto que devuelve un ÚNICO valor. Este valor será retornado con la sentencia *return*.

■ Métodos: Argumentos valor y referencia

□ Argumento por valor

- Permanece igual tras la llamada
- Se genera una copia de los datos
- Los datos primitivos se pasan por valor

□ Argumento por referencia

- Será modificado en el interior del método
- Es más eficiente para la memoria.
- Los datos referenciados se pasan por referencia para evitar que el sistema cree una copia.

```
void bar() {  
    Foo f = new Foo();  
    doStuff(f);  
}  
void doStuff(Foo g) {  
    g.setName("Boo");  
    g = new Foo();  
}
```

■ Ejemplo *workspace*

- Package `com.slashmobility.referencia`: clase `Referencia.java`.
- Contiene un ejemplo de cómo evoluciona la modificación de valores a mientras pasan por métodos de clases.

■ Métodos: Sobrecarga

- En programación tradicional el nombre de los métodos debía ser diferenciable.
 - No podíamos tener varios procedimientos o funciones con el mismo nombre .
- En POO se permite *sobrecarga de métodos*
 - Utilizar el mismo identificador para implementar múltiples métodos.

■ Métodos: Sobrecarga (II)

- Los métodos sobrecargados **DEBEN** cambiar la lista de argumento
 - Ya sea en tipo o en número
- Los métodos sobrecargados **PUEDEN** cambiar el tipo de devuelven
- Los métodos sobrecargados **PUEDEN** cambiar los modificadores de acceso
- Los métodos sobrecargados **PUEDEN** declarar nuevas excepciones lanzadas
- Ejercicio ¿qué sobrecargas son correctas?:
 - `public void metodo(int x);`
 - `public void metodo(int x, double y);`
 - `public void metodo(int y);`
 - `public void metodo(String s, int x);`
 - `public int metodo(int x, double y);`

■ Sobrecarga: Tipos

■ Distinta clase

```
class ClaseA
{
    void imprimir()
    {
        ...
    }
}

class ClaseB
{
    void imprimir()
    {
        ...
    }
}
```

■ Distintos argumentos

```
class ClaseA
{
    void metodoUno()
    {
        ...
    }
    void metodoUno(int n)
    {
        ...
    }
}
```

■ Mismos argumentos diferente tipo

```
class ClaseA
{
    void metodoUno(int a)
    {
        ...
    }
    void metodoUno(String s)
    {
        ...
    }
}
```

■ Métodos: Constructores

□ Para crear e inicializar el objeto

- Métodos con el mismo nombre que la clase
- Son públicos
- No devuelven nada (ni siquiera *void*)
- Se pueden sobrecargar

■ Tipos

□ Por defecto:

```
public ClaseA()  
{  
    //sentencias de inicializacion del objeto  
}
```

□ Con Argumento:

```
public ClaseA(tipo arg1, tipo arg2)  
{  
    //sentencias de inicializacion del objeto  
}
```

□ Copia

```
public ClaseA(ClaseA unObjeto)  
{  
    //sentencias de inicialización del objeto  
}
```

■ Constructores: *Ejemplo*

- Inicializaremos los atributos.
- El nombre coincide con el de la clase.

```
public class Cuenta {
    long numero;
    String titular;
    private float saldo;
    float interesAnual;

    Cuenta (long aNumero, String aTitular, float aInteresAnual) {
        numero = aNumero;
        titular = aTitular;
        saldo = 0;
        interesAnual = aInteresAnual;
    }

    public void ingreso (float cantidad) {
        saldo += cantidad;
    }

    // Resto de operaciones de la clase Cuenta a partir de aquí
}
```

- Métodos Constructores: *Sobrecarga*
 - Se pueden definir varios constructores posibles para una clase siempre que se diferencien en la lista de argumentos

```
// constructor para obtener los datos de la cuenta de un fichero
Cuenta (long aNumero) throws FileNotFoundException, IOException, ClassNotFoundException {

    FileInputStream fis = new FileInputStream (aNumero + ".cnt");
    ObjectInputStream ois = new ObjectInputStream (fis);
    numero = aNumero;
    titular = (String) ois.readObject ();
    saldo = ois.readFloat ();
    interesAnual = ois.readFloat ();
    ois.close ();
}
```

```
// Constructor general
Cuenta (long aNumero, String aTitular, float aInteresAnual) {
    numero = aNumero;
    titular = aTitular;
    saldo = 0;
    interesAnual = aInteresAnual;
}
```

■ Métodos Constructores por defecto

- Si no se proporciona ningún constructor, Java proporciona automáticamente un **constructor por defecto** que no toma argumentos y realiza un valor por defecto de los atributos.

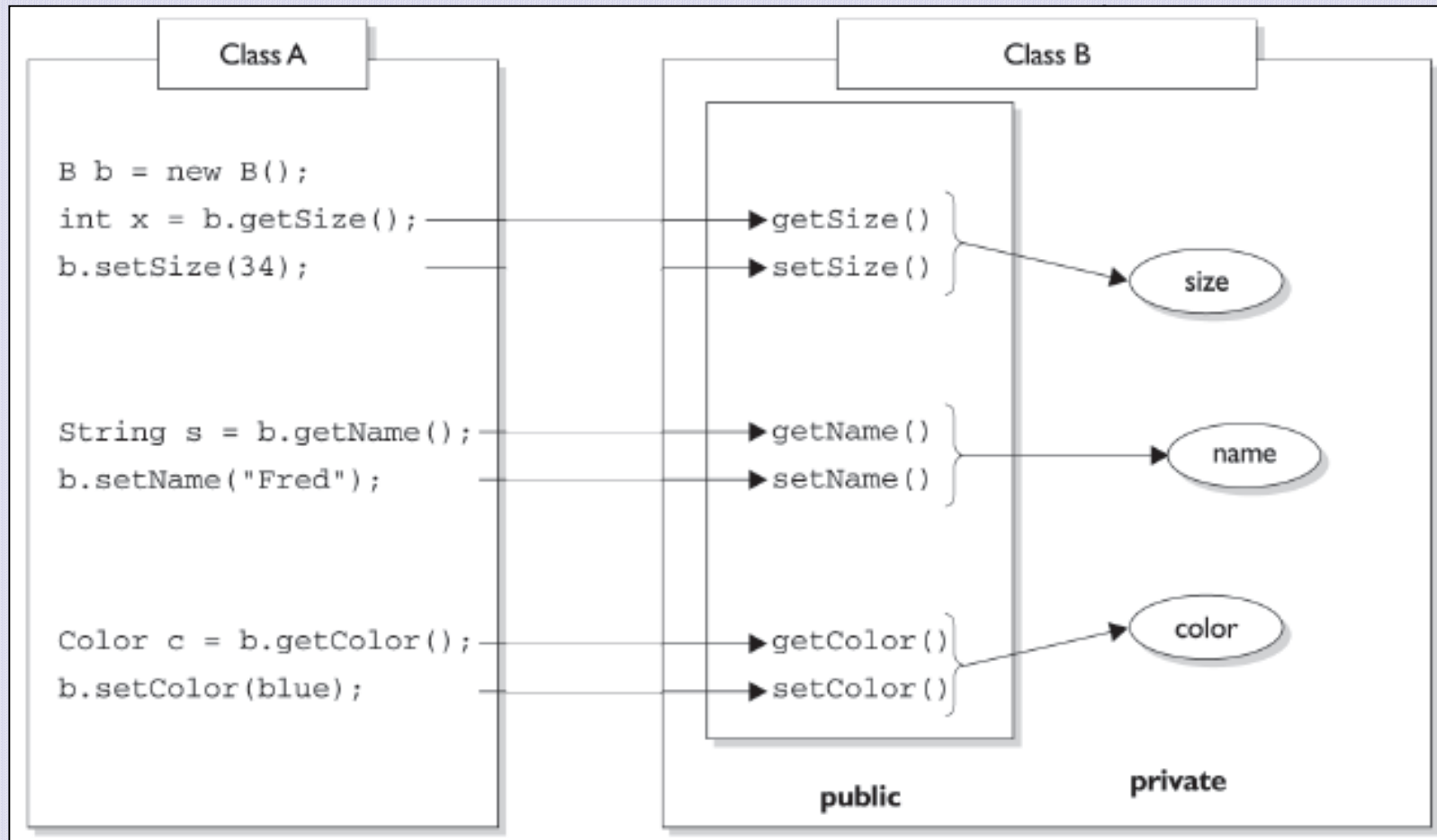
```
cuenta () {  
    numero = "000000000";  
    titular = "ninguno";  
    saldo = 0;  
    interesAnual = 0;  
}
```

- Una vez implementado un constructor propio por parte del programador, Java elimina dicho constructor, aunque puede ser definido nuevamente de manera explícita.

■ Métodos: De acceso

```
public class Ingrediente
{
    //Declaración de los atributos
    public int stock;
    public String nombre;
    ...
    public int getStock()
    {
        return stock;
    }
    public String getNombre()
    {
        return nombre;
    }
    public void setStock(int s)
    {
        stock = s;
    }
    public void setNombre(String n)
    {
        nombre = n;
    }
}
```

- Métodos: De acceso / Acoplamiento
 - Al no acceder directamente a las variables



■ Ejemplo

- Clase que calcula mi saldo en euros y pesetas

```
public class SaldoInternacional {  
  
    public double euros;  
    public double pesetas; saldoInternacional.pesetas = 30000;  
  
    /**  
     * Get the value of pesetas  
     *  
     * @return the value of pesetas  
     */  
    public double getPesetas() {  
        return pesetas;  
    }  
  
    /**  
     * Set the value of pesetas  
     *  
     * @param pesetas new value of pesetas  
     */  
    public void setPesetas(double pesetas) {  
        this.pesetas = pesetas;  
        this.setEuros(pesetas / 166.386);  
    }  
}
```

¡¡Mi saldo en euros no se actualizaría!!

■ Métodos: Comportamiento

- Dan comportamiento o funcionalidad específica a los futuros objetos de la clase.

```
public class ingrediente
{
    ...
    public void decrementarStock(int s)
    {
        stock -= s;
    }
    public void incrementarStock(int s)
    {
        stock += s;
    }
    public String toString()
    {
        return "Ingrediente = " + nombre + "\n" +
               "Stock          = " + stock + "\n";
    }
    ...
}
```

■ Métodos: Destruectores

- Cuando finaliza el uso de un objeto, es frecuente la realización de ciertas tareas antes de su destrucción, principalmente la liberación de la memoria solicitada durante su ejecución.
- Operaciones invocadas automáticamente justo antes de la destrucción del objeto
- Sin embargo, en Java la liberación de memoria se realiza de **manera automática** por parte del recolector de basura, por tanto la necesidad de este tipo de operaciones no existe en la mayor parte de los casos

■ Métodos: Destructores (II)

- Si es necesario realizar alguna tarea no relacionada con la liberación de memoria.
 - Salvar el estado de la clase en BDD
- Java permite introducir código para este fin implementando una función pública (***finalize***)
- Invocada antes de la destrucción del objeto por parte del recolector de basura.

```
public void finalize () : throws FileNotFoundException, IOException {  
  
    FileOutputStream fos = new FileOutputStream (numero + ".cnt");  
    ObjectOutputStream oos = new ObjectOutputStream (fos);  
    oos.writeObject (titular);  
    oos.writeFloat (saldo);  
    oos.writeFloat (interesAnual);  
    oos.close ();  
  
}
```

■ Métodos: Destructores (III)

- Método encargado de eliminar la memoria dinámica creada ocupada por los atributos.
- Será el último en ejecutarse.
- En Java lo hace automáticamente el proceso de baja prioridad *garbage Collector*
- Hay que sobrescribir el siguiente método

```
protected void finalize()  
{  
    //sentencias que ejecutará el método  
}
```

■ Garbage Collector

- Sin embargo no sabemos en que momento será llamada dicha operación, puesto que el recolector de basura puede decidir su eliminación en un momento indeterminado, e incluso no ser eliminado hasta el final de la ejecución de la aplicación
- Una posible solución, aunque no muy recomendable, es ordenar al recolector de basura que realice una limpieza de memoria inmediata, para asegurar la finalización de los objetos.
- Esto se realiza mediante *Runtime.getRuntime().gc()*
- Se recomienda crear en su lugar una operación ordinaria con este mismo propósito, que sea invocada cuando haya finalizado el uso del objeto

■ Creación de Objetos

- En Java los objetos se crean únicamente de forma dinámica, es decir, en el heap. Para ello se utiliza el operador `new`, de manera similar a C++
- Los objetos en Java se utilizan siempre a través de referencias.
- Por tanto los pasos a seguir en la creación de un objeto son:
 - Declarar una referencia a la clase
 - Crear un objeto mediante el operador `new` invocando al constructor adecuado
 - Conectar el objeto con la referencia

■ Creación de Objetos: *Ejemplo*

```
Cuenta c;  
float in;  
long num;  
  
in = 0.1f;  
num = 18400200;  
  
c = new Cuenta (num, "Pedro Jiménez", in);
```

e Cuenta

```
clase Cuenta  
f);
```

- Los tipos básicos (int, long, float, etc.) sí pueden ser creados directamente en la pila.
 - Java no los implementa realmente como Objetos
 - Por motivos de eficiencia (Uso más frecuente)

■ Creación de Objetos: *String*

- Las cadenas de caracteres se implementan con una clase (*String*). Sin embargo no suele ser necesaria su creación de manera explícita, ya que Java lo hace de manera automática al asignar una cadena constante

```
String s; // Una referencia a un objeto de la clase String

// Conexión de la referencia s con un objeto String
// creado dinámicamente e inicializado con la constante "Pedro"
s = "Pedro";

// Sería equivalente a:
// char[] cc = {'P', 'e', 'd', 'r', 'o'}
// s = new String (cc);
```

■ Creación de Objetos: *Array*

- Los arrays también deben ser creados dinámicamente con *new* como si fueran objetos.

```
int [] v; // Una referencia a un vector de enteros
v = new int[10] // Creación de un vector de 10 enteros
```

- Si el array es de referencias a objetos, habrá que crear además cada uno de los objetos referenciados por separado

```
Cuenta[] v; // Un vector de referencias a objetos de la clase Cuenta
int c;

v = new Cuenta [10] // Crear espacio para 10 referencias a cuentas
for (c = 0; c < 10; c++)
    v[c] = new Cuenta (18400200 + c, "Cliente n. " + c, 0.1f);
```

- Trabajando con Objetos:
 - Trabajar con un objeto Java es similar a C++, aunque las referencias permiten un uso mucho más sencillo.

```
Cuenta c1 = new Cuenta (18400200, "Pedro Jiménez", 0.1f);
Cuenta c2 = new Cuenta (18400201);

c2.reintegro (1000);
c1.ingreso (1000);

if (c2.enRojos ())
    System.out.println ("Atención: cuenta 18400201 en números rojos");
```



■ Trabajando con Objetos:

- Naturalmente el compilador producirá un error ante cualquier acceso ilegal a un miembro de la clase.

```
Cuenta c = new Cuenta (18400200, "Pedro Jiménez", 0.1f);  
c.saldo = 1000;  
  
Cuenta.java:XX: saldo has private access  
c.saldo = 1000;  
^  
1 error
```

- El acceso a un miembro estático se realiza utilizando el nombre de la clase en lugar de un objeto.

```
Cuenta c = new Cuenta (18400200, "Cliente 1", 0.1f);  
  
c.ingreso (1000);  
int pts = Cuenta.eurosAPesetas (c.leerSaldo ());  
  
System.out.println ("Saldo: " + c.leerSaldo () + "(" + pts + " pesetas");
```



Conclusiones

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. **POO en Java**
5. Relaciones entre Objetos
6. Polimorfismo, abstracción e interfaces
7. Excepciones
8. Conceptos avanzados

- Introducción
- Elementos de la POO
- Estructura de Clase
 - Visibilidad
 - Declaración atributos y métodos
 - Argumento Valor/Referencia
 - Puntero This
 - Sobrecarga
- Métodos
 - Constructores
 - De Acceso
 - Comportamiento
 - Destrucción