

Programación

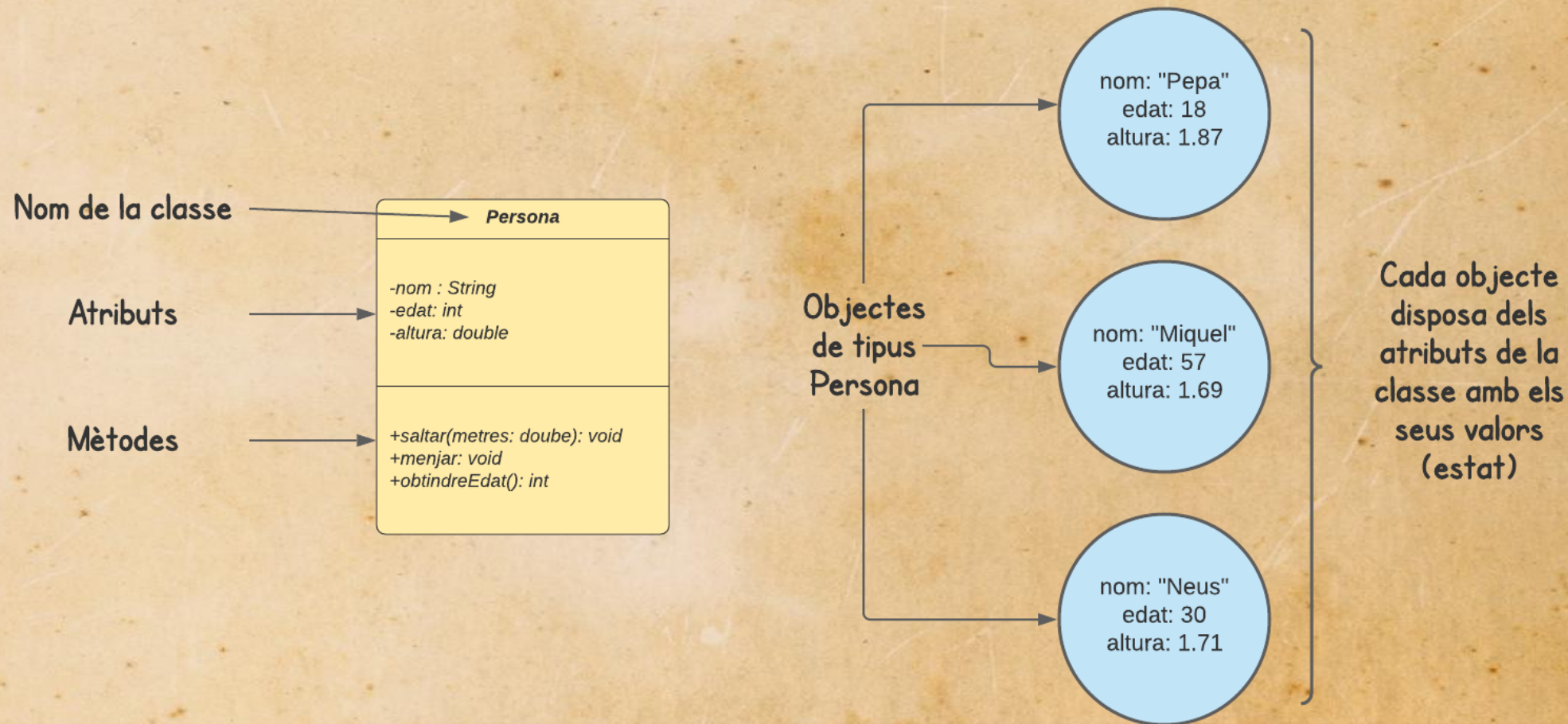
5.2 POO en J ava

Definiendo clases y objetos

Una clase en Java utiliza variables para definir los atributos y métodos para definir las operaciones.

Adicionalmente, existen unos métodos especiales conocidos como constructores que son invocados normalmente para crear **inicializar un objeto que acabamos de crear**.

Definiendo clases y objetos



Referencias en Java

- El **comportamiento** de los objetos en la memoria del ordenador y sus operaciones elementales (creación, asignación y destrucción) **es idéntico del s arrays**. Esto es por lo que tanto objetos como arrays utilizan referencias.
- Antes de construir un objeto **declarar una variable** de la cual sea su clase. La diferencia es que mientras que la variable de tipo primitivo enmaga directamente un valor, la de tipo referencia **almacena la referencia a un objeto**.

Persona p; //p es una variable de tipo persona (referencia)

Operador *new*

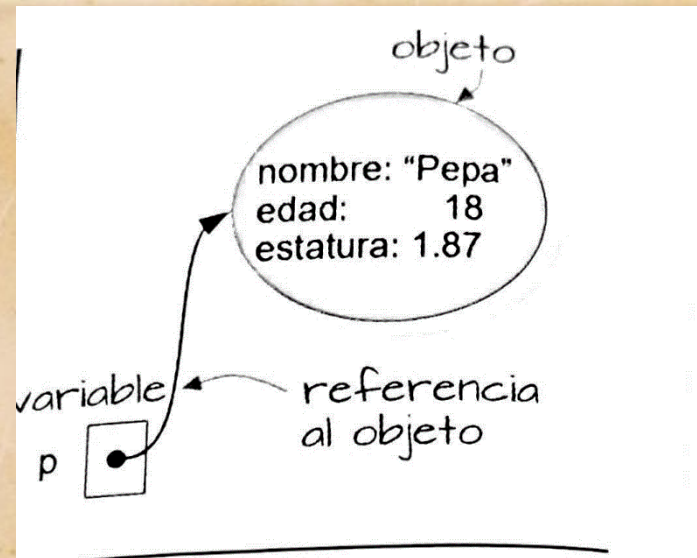
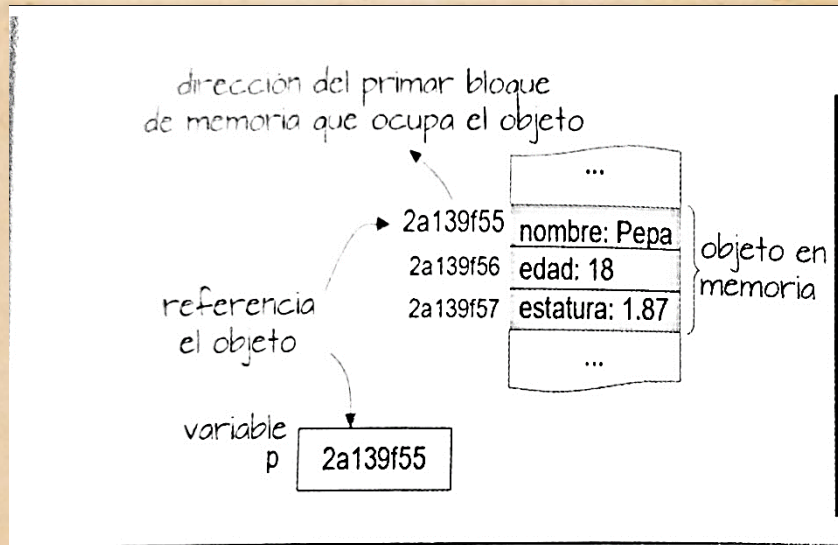
- La forma de **crear objetos**, como para los arrays, es utilizando el operador ***new***.

```
p = new Persona();
```

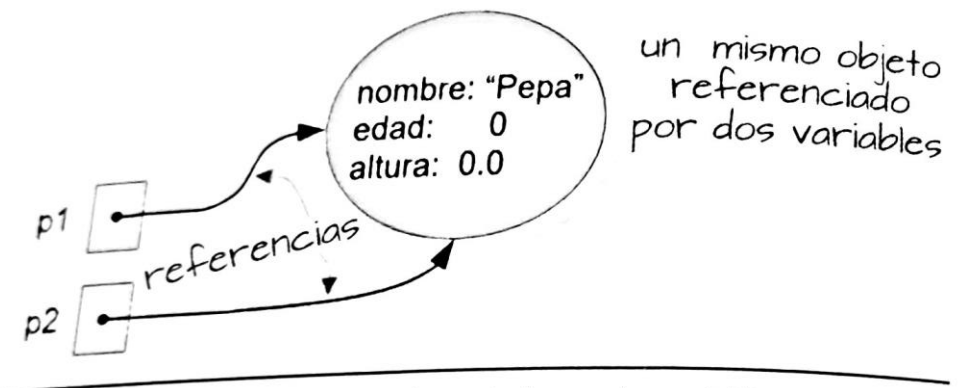
- En su caso **se crea un objeto** de tipo persona, **y se asigna su referencia a la variable *ep***. El operador ***new*** busca en memoria un espacio disponible para construir el objeto, ocupa los sitios de memoria ya que necesita del espacio libre y finalmente devuelve la referencia ya que el objeto recientemente creado, que se asigna a la variable *ep*.

¿Qué imprimiría `System.out.println(p)` ?

Operador *new*



```
Persona p1, p2;  
p1 = new Persona(); p2 =  
p1;  
p2.número = "Pepa"
```



Referencia a *null*

- El valor le tera el “null” es una referencia a nula. Dicho de otra manera, una referencia a ningún bloque de memoria.
- Cuando decimos una varilla y la referencias inicializa por defecto a nulo.
- Si intentamos acceder a los miembros de una referencia a nula se producirá un error y acabará el ejecución del programa:

Persona p; // Se inicializa por defecto en null p.obtenerEdad(); // Error de tipo Null Pointer Exception

Es til cuando el me que una va riable de e deferrefer éncia a un objeto:

Persona p = new Persona(); // p referencia a un objeto

p. p = null; // p no referencia nada.

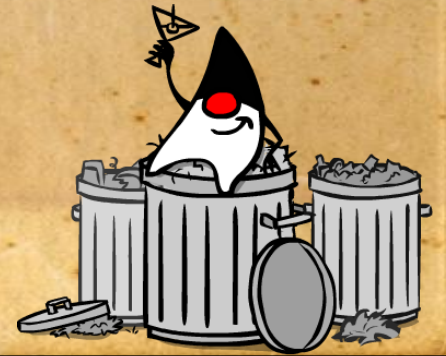
Recolector de memoria basura.

Hay tres formas de conocer que un objeto no esté referenciado:

- Crear un objeto y no asegurarlo en ningún tipo de abuelo. `new Persona();`
- Asignar a nulo una variable y bien que tenga una referencia a un objeto.
`Persona p = new Persona(); p = null;`
- Asignar un objeto distinto a una variable.
`Persona p = new Persona(); p = new Persona();`

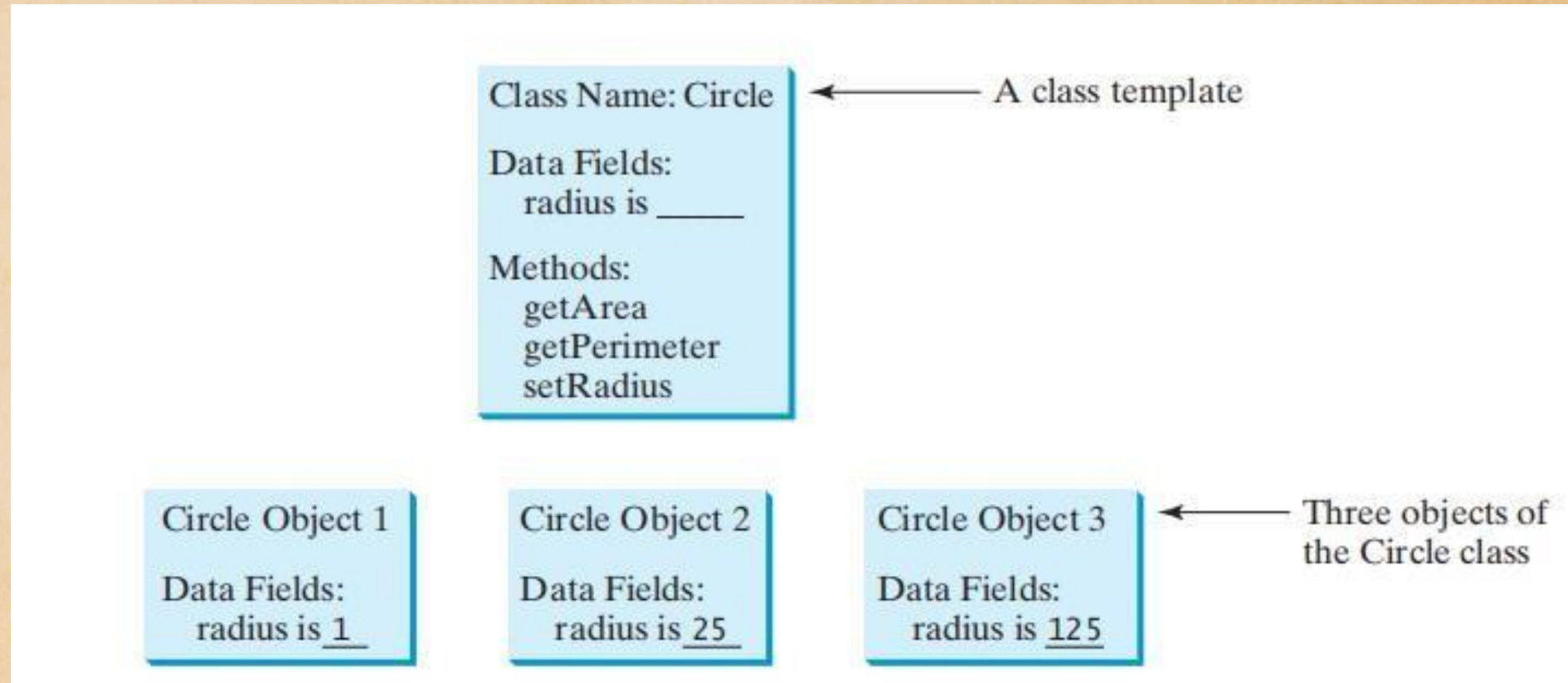
Recolector de memoria basura.

- En todos esos casos el objeto **queda perdido en memoria**, es decir, no hay forma de poder acceder a ellos y se quedan consumiendo memoria en .
- Para evitar este problema, Java llama a un mecanismo de un recolector de basura (garbage collector) que se ejecuta periódicamente de forma transparente al usuario destruyendo los objetos que no se tienen referenciados, liberando la memoria ya que ocupan.



Ejemplo de clase y objetos:

Circulo



Ejemplo de definición de una clase

```
public class SimpleCircle {  
    private double radius;  
    /** Construct a circle with radius 1 */  
    public SimpleCircle() {  
        radius = 1;  
    }  
    /** Construct a circle with a specified radius */  
    public SimpleCircle(double newRadius) {  
        radius = newRadius;  
    }  
    /** Return the area of this circle */  
    public double getArea() {  
        return radius * radius * Math.PI;  
    }  
    /** Return the perimeter of this circle */  
    public double getPerimeter() {  
        return 2 * radius * Math.PI;  
    }  
    /** Set a new radius for this circle */  
    public void setRadius(double newRadius) {  
        radius = newRadius;  
    }  
}
```

Diagram illustrating the components of the `SimpleCircle` class definition:

- Data field:** `private double radius;`
- Constructors:** `SimpleCircle()` and `SimpleCircle(double newRadius)`
- Method:** `getArea()`, `getPerimeter()`, and `setRadius(double newRadius)`

Creación de objetos de clases propias

Tal y como hemos venido haciendo con las clases del API de Java, podemos crear objetos de clases definidas por nosotros mismos.

Como hemos dicho antes, haciendo uso del operador **new** se crea un objeto invocando al **constructor** se inicializan sus datos (le damos un estado al objeto).

Creación de objetos para probarlos

```
public static void main(String[] args) {  
    // Create a circle with radius 1  
    SimpleCircle circle1 = new SimpleCircle();  
    System.out.println("The area of the circle of radius "  
        + circle1.radius + " is " + circle1.getArea());  
  
    // Create a circle with radius 25  
    SimpleCircle circle2 = new SimpleCircle(25);  
    System.out.println("The area of the circle of radius "  
        + circle2.radius + " is " + circle2.getArea());  
  
    // Create a circle with radius 125  
    SimpleCircle circle3 = new SimpleCircle(125);  
    System.out.println("The area of the circle of radius "  
        + circle3.radius + " is " + circle3.getArea());  
  
    // Modify circle radius  
    circle2.radius = 100;  
    System.out.println("The area of the circle of radius "  
        + circle2.radius + " is " + circle2.getArea());  
}
```


Práctica 1

Escribe la clase SimpleCircle en un archivo Java y crea en el paquete de test del proyecto el método main. TestSimpleCircle que contendrá el main planteado en la diapositiva anterior.

Prueba su funcionamiento.

Creación de métodos de una clase

```
public <tipus1> <nomMètode>(<tipus2> <parametre>, ...)
```

donde el **tipo1** indica el tipo del valor devuelto, que puede ser:

- void (nada)
- <tipo/Clase> (un valor de tipo primitivo o una **referencia** a un objeto)
- <tipo/Clase>[] (una referencia a un vector de valores de tipo primitivo o de referencias a objetos)
- <tipo/Clase>[][] (una referencia a una matriz de valores de tipo primitivo o de referencias a objetos) ...
 - donde <nombreMétodo> empezará en minúscula
 - donde el tipo2 puede ser cualquier tipo (excepto void)
 - donde todos los parámetros (0 a N) son pasados por valor

Otro ejemplo

Supongamos que necesitamos crear una clase que tenga las características de un televisor para un programa que necesitará de ese tipo de objetos.

Después de un análisis previo llegamos a la conclusión de que las propiedades de un televisor que nos interesan e identifican en el televisor en nuestro problema son:

- **el canal que se está viendo**
- **el nivel de volumen**
- **encendido o apagado**

Otro ejemplo

Las operaciones que puede realizar el televisor son:


- **encender**
- **apagar**
- **subir el volumen**
- **bajar el volumen**
- **cambiar al canal siguiente**
- **cambiar al canal anterior**


Práctica 2

Siguiendo la estructura de la clase SimpleCircle, define la claseTVy crea un archivoTestTV.javapara probar su funcionamiento.

Creardos **televisores**, realiza diferentes operaciones con ellos y comprueba que el estado después de esas operaciones es el esperado.

Sobrecarga de métodos



- Funcionalidad que permite tener **métodos distintos con el mismo nombre**.
 - La lista de argumentos (parámetros de entrada) debe ser diferente para **identificar inequívocamente** cada método que se llame igual.
 - No se puede sobrecargar un método en función de su parámetro de salida.
- 

Sobrecarga de métodos. Ejemplo

- Dada una clase Personase pueden definir estos dos métodos:

```
public void caminar(int pasos) {  
    //Aquí implementaría el método pasos(int)  
}  
  
public void andar(double pasos) {  
    //Aquí implementaría el método pasos(double)  
}
```

```
public int andar(int numPasos) {  
    //ERROR
```



Construyendo objetos y usando constructores

Los constructores **son un tipo especial de métodos** que tienen TRE S particularidades:

- Tienen que tener **el mismo nombre que la clase**
- No tienen **tipos de dato de retorno** (ni void)
- Los constructores **los invocamos usando el operador new** (que es lo que crea el objeto) cuando queremos inicializarlo.

Construyendo objetos y usando constructores

Otras misiones de los constructores pueden ser:

- Recoger los valores para tenerlos en cuenta a la hora de crear el objeto.
- Gestionar los errores que puedan aparecer en su fase de inicialización.
- Aplicar procesos, más o menos complicados, en los que puede intervenir todo tipo de sentencias (condicionales o repetitivas).

Si el constructor realiza más de una tarea (complejo) es conveniente modularlo.

Declaración de Constructores.

- Una misma clase puede disponer de **varios constructores**. La diferencia entre ellos la determina el orden y tipos de los parámetros de entrada que le proporcionamos
- Si no se define ningún constructor se autodefine un **constructor por defecto** (que será un constructor vacío, sin parámetros).
- Si se define algún constructor, **no se proporcionará el constructor vacío por defecto** (se debería definir expresamente).

Ejemplo de un posible constructor de la clase Persona

```
public Persona (String unDni,  
String unNombre, int unaEdad) {  
    dni = unDni;  
    nombre = unNombre;  
    if (unaEdad >= 0 && unaEdad <= 150)  
        edad = unaEdad;  
}
```

.....

**// Después se definirán los
sus métodos**



Ejemplo de TestPersona.java

```
public static void main(String args[]) {  
    Persona p1 = new Persona("000000000", "Pepe Gotera", 33);  
    Persona p2 = new Persona(); //Aquí habrá un error  
    System.out.println("Visualización de persona p1:");  
    p1.visualizar();  
    System.out.println(" Visualización de persona p2:");  
    p2.visualizar(); // Imaginemos que hemos creado el constructor vacío.  
}
```


Ejemplo: Clase Persona (v1.0)

Visualización de persona p1:

Dni.....:00000000

Nombre.....:Pepe Gotera

Edad.....:33

Visualización de persona p2:

Dni.....:null

Nombre.....:null

Edad.....:0

Práctica 3

- Crea un nuevo proyecto que contendrá la clase `Person` y define el método `visualizar` que mostrará por pantalla los datos de una persona.
- Crea la clase `TestPerson` donde escribirás el método `main` de la diapositiva anterior. Aparecerá un error en una de las líneas del método `main`.
- Visualiza los datos de ambas personas y razona por qué aparecen los resultados que aparecen.

Ejercicio "CuentaCorrente" (I)

Diseña la clase *CuentaCorrente* que almacene los datos: DNI y nombre del titular, así como el saldo. Las operaciones típicas con una cuenta corriente son:

- **Crear una cuenta:** Se necesita el DNI y nombre del titular. El saldo inicial será 0.0
- **Sacar dinero:** el método debe indicar si ha sido posible llevar a cabo la operación (si existe saldo suficiente)
- **Ingresar dinero:** Se incrementa el saldo.
- **Mostrar información:** Muestra la información disponible de la cuenta corriente (nombre, DNI y saldo de la cuenta)

Pregunta

¿Qué estoy haciendo en cada línea?

1. `int x;`
2. `Intervalo r;`
3. `Intervalo r = new Intervalo();`
4. `int[] t;`
5. `t = new int[100];`
6. `Intervalo[] tt;`
7. `Intervalo[] tt = new Intervalo[100];`
8. `for (int i = 0; i < tt.length; i++) { tt[i] = new Intervalo();`
- `}`

Referenciación a objetos

```
// Declaración de la variable X de referencia obj1 no inicializada (valdrá null)  
obj1;
```

```
// Creación del objeto al que se podrá acceder vía la variable de referencia obj1
```

```
obj1 = new X(...);
```

```
// Declaración de la variable de referencia obj2 y creación del objeto al que se podrá  
acceder vía la variable de referencia obj2
```

```
X obj2 = new X(...);
```

```
// Declaración de la variable de referencia obj3 no inicializada (valdrá null)
```

```
X obj3;
```

```
// La variable obj3 hace referencia al mismo objeto que hace referencia a la variable obj1
```

```
obj3 = obj1;
```

```
// Declaración de variable de referencia que hace referencia al mismo objeto que // referencia la  
variable obj2
```

```
X obj4 = obj2;
```


Valor null y NullPointerException

Recuerda que las variables de referencia no inicializadas por defecto tienen el valor null.

Acceder a una propiedad (con el operador.) de una variable null provocará que aparezca la excepción

NullPointerException

```
public class ShowErrors {  
    public void method1() {  
        Circle c;  
        System.out.println("What is radius "  
            + c.getRadius());  
        c = new Circle();  
    }  
}
```


Pregunta

¿Por qué falla este programa?

```
class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.print();  
    }  
}  
  
class A {  
    String s;  
  
    public A(String newS) {  
        s = newS;  
    }  
  
    public void print() {  
        System.out.print(s);  
    }  
}
```


Definición de atributos

- Si te has dado cuenta, los **atributos de una clase** serán lo que hasta ahora conocíamos como variables globales, no dejando de ser globales en clase por este hecho.
- Es muy importante definir correctamente los modificadores para mantener los principios de abstracción y encapsulación.

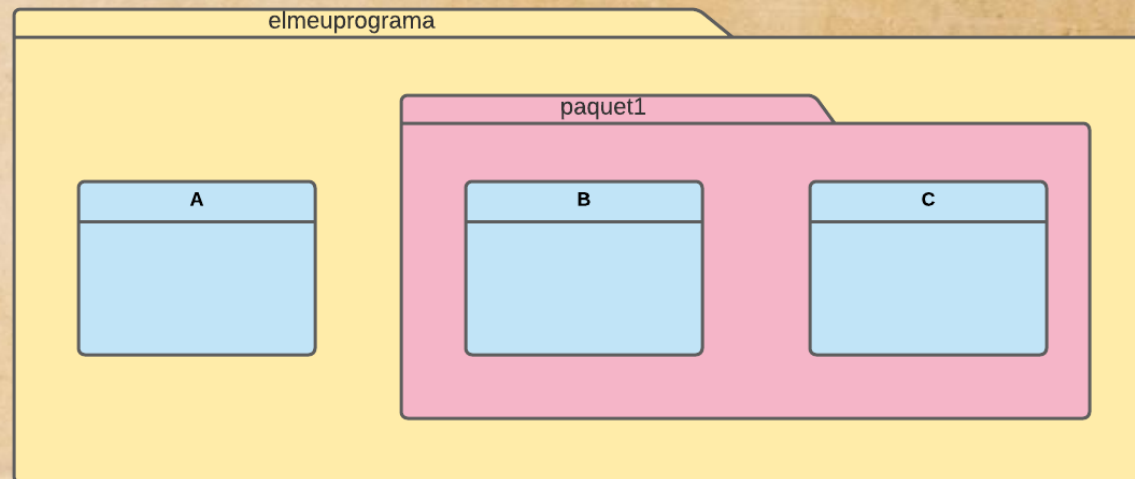
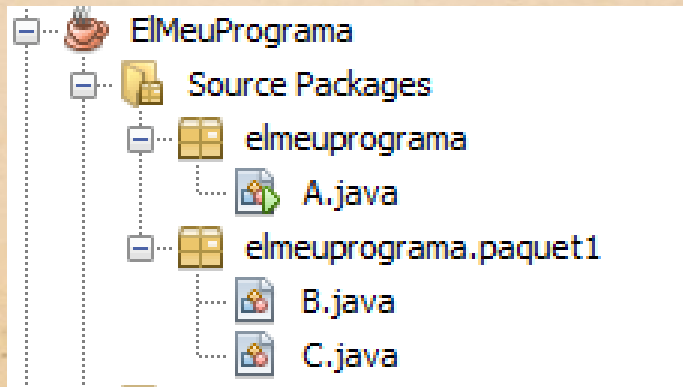
Modificadores de acceso o visibilidad

- Una clase será visible por otra, o no, dependiendo de si se encuentran en el mismo **paquete** y de los **modificadores de acceso** que utilice.
- Estos modificadores **cambian su visibilidad** entre clases, permitiendo que se muestre u oculte.
- Al igual que podemos modificar la visibilidad **entre clases**, también se puede modificar la visibilidad **entre miembros** de distintas clases, es decir, qué atributos y métodos son visibles para otras clases.

Modificadores de acceso para clases

Debido a la estructura de clases de Java, organizadas en paquetes, dos clases cualquiera pueden definirse como:

- **Clases vecinas:** Cuando ambas pertenecen al mismo paquete.
- **Clases externas:** Cuando se han definido en paquetes distintos.



B i C són
classes veïnes

A és externa
al paquet1

Modificadores de acceso para clases

Podremos encontrar dos tipos de visibilidades para las clases:

- **Visibilidad por defecto:** Cuando definimos una clase sin utilizar ningún modificador de acceso. Esa clase **sólo** será visible para sus **clases vecinas**.
- **Visibilidad total:** Cuando definimos una clase utilizando el modificador de acceso "**public**". Esa clase será **visible para cualquier otra clase**, sea vecina o no. Si la clase es externa será necesario que **import** la clase a la que desea tener acceso.

```
package elmeuprograma.paquet1;

class B { // Sense modificador d'accés.
    /*
        Segons la nostra estructura, B
        serà visible per a C (veïna) pero
        B no serà visible per a A (externa)
    */
}
```

```
package elmeuprograma.paquet1;

public class B { // Modificador public.
}
```

```
package elmeuprograma;
import elmeuprograma.paquet1.B;
class A {
    // ...
}
```


Modificadores de acceso para clases

	Visible desde ...	
	Clases vecinas	Clases externas
Sin modificador	✓	X
público	✓	✓

Modificadores de acceso para miembros

- De la misma forma que es posible modificar la visibilidad de una clase, podemos regular la visibilidad de sus miembros.
- Que un atributo sea visible significa que podemos acceder a él, tanto para leer como modificarlo. Que un método sea visible significa que puede ser invocado.
- Para que un miembro sea visible es indispensable que su clase lo sea también.
- Cualquier miembro es siempre visible dentro de su propia clase, independientemente del modificador de acceso que tenga.

```
package elmeuprograma;  
  
public class A { // Classe pública  
    int dada; // El seu àmbit és tota la classe.  
    // L'atribut 'dada' és accessible des de qualsevol lloc d'A  
}
```


Modificadores de acceso para miembros

Podremos encontrar los siguientes tipos de visibilidad para miembros:

- **Por defecto (default o package):**Cuando definimos un miembro sin especificar ningún modificador de acceso. Estos miembros serán visibles sólo desde las clases vecinas.
- **Pública (public):**Cuando definimos a un miembro con el modificador de acceso "public". Será visible para cualquier otra clase, sea o no vecina.
- **Privada (private):**Cuando definimos un miembro con el modificador de acceso "private", será visible sólo para la propia clase que los define (ni vecinas ni externas).
- **Protegida (protected):**Cuando definimos a un miembro con el modificador de acceso "protected". Esos miembros serán visibles para sus clases vecinas y para las sus subclases aunque sean externas.

Modificadores de acceso para miembros

	Visible desde ...		
	La propia clase	Clases vecinas	Clases externas
private	✓	X	X
sin modificador	✓	✓	X
protected	✓	✓	Sólo clases derivadas o subclases
público	✓	✓	✓

Ejercicio “CuentaCorrente” (II)

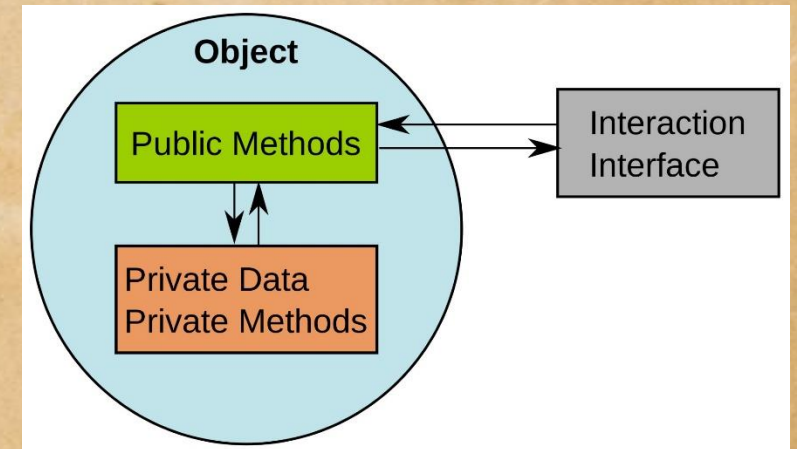
Modifica la visibilidad de la clase “CuentaCorrente” para que sea visible desde las clases externas y la visibilidad de sus miembros sea:

- **saldo:** no será visible para otras clases.
- **nombre:** será visible para cualquier otra clase.
- **dni:** sólo será visible para las clases vecinas.
- Los constructores y otros métodos serán visibles para cualquier clase.

Métodos *get* y *set*

Un atributo público puede ser leído y modificado desde cualquier clase lo que plantea graves inconvenientes:

- Eliminamos el **principio de ocultación**, dejando al descubierto los detalles de implementación (no nos interesa que sea visible toda la estructura del objeto, sólo cierta parte de nuestra encapsulación).
- - No podemos **controlar los valores asignados** a estos atributos, que pueden tener o no oído. Por ejemplo, se puede asignar a la edad un valor negativo.



Métodos *get*/*set*

Por ese motivo existe una **convención** entre programadores que consiste en **ocultar los atributos**, y en su lugar crear dos métodos públicos para cada uno de los atributos a los que queramos dar visibilidad (habrá atributos con ambos, sólo uno, o ninguno de los métodos get/set).

- El primer método, **set**, permitirá asignar un valor a un atributo, pudiendo validar o manipular este valor antes de ser asignado al atributo.
- El segundo, **get**, devuelve el valor del atributo o bien alguna transformación del mismo (o copia defensiva)

Métodos *get*/*set*

Por convención estos métodos se identifican con set/get seguido del nombre del atributo. Por ejemplo para el atributo "edad":

```
public class Persona {  
    private int edad;  
  
    public int getEdad() {  
        return edad;  
    }  
  
    public void setEdad(int edad) {  
        if (edad >= 0) {  
            this.edad = edad;  
        }  
    }  
}
```


Clase Circle con atributos privados

```
public class CircleWithPrivateDataFields {  
    /** The radius of the circle */  
    private double radius = 1;  
  
    /** The number of objects created */  
    private static int numberOfObjects = 0;  
  
    /** Construct a circle with radius 1 */  
    public CircleWithPrivateDataFields() {  
        numberOfObjects++;  
    }  
  
    /** Construct a circle with a specified radius */  
    public CircleWithPrivateDataFields(double newRadius) {  
        radius = newRadius;  
        numberOfObjects++;  
    }  
}
```

```
}  
  
    /** Return radius */  
    public double getRadius() {  
        return radius;  
    }  
  
    /** Set a new radius */  
    public void setRadius(double newRadius) {  
        radius = (newRadius >= 0) ? newRadius : 0;  
    }  
  
    /** Return numberOfObjects */  
    public static int getNumberOfObjects() {  
        return numberOfObjects;  
    }  
  
    /** Return the area of this circle */  
    public double getArea() {  
        return radius * radius * Math.PI;  
    }  
}
```


Paso de objetos como parámetros

- Es posible pasar objetos como parámetros en los métodos planteados.

```
public class Test {  
    public static void main(String[] args) {  
        // CircleWithPrivateDataFields is defined in Listing 9.8  
        CircleWithPrivateDataFields myCircle = new  
            CircleWithPrivateDataFields(5.0);  
        printCircle(myCircle);  
    }  
  
    public static void printCircle(CircleWithPrivateDataFields c) {  
        System.out.println("The area of the circle of radius "  
            + c.getRadius() + " is " + c.getArea());  
    }  
}
```


Ejercicio “CuentaCorrente” (III)

Modifica la clase para ocultar todos sus atributos. Después debes implementar los métodos necesarios para poder leer los valores de los atributos “saldo”, “nombre” y “dni” y para poder modificar los atributos “nombre” y “dni” (asegúrate de que el DNI contenga 9 letras)

Pregunta

¿Cuál es la salida de este código?

```
public class Test {  
    public static void main(String[] args) {  
        Count myCount = new Count();  
        int times = 0;  
  
        for (int i = 0; i < 100; i++)  
            increment(myCount, times);  
  
        System.out.println("count is " + myCount.count);  
        System.out.println("times is " + times);  
    }  
  
    public static void increment(Count c, int times) {  
        c.count++;  
        times++;  
    }  
}
```

```
public class Count {  
    public int count;  
  
    public Count(int c) {  
        count = c;  
    }  
  
    public Count() {  
        count = 1;  
    }  
}
```


Pregunta

¿Cuál es la salida de este código?

```
public class Test {  
    public static void main(String[] args) {  
        Circle circle1 = new Circle(1);  
        Circle circle2 = new Circle(2);  
  
        swap1(circle1, circle2);  
        System.out.println("After swap1: circle1 = " +  
            circle1.radius + " circle2 = " + circle2.radius);  
  
        swap2(circle1, circle2);  
        System.out.println("After swap2: circle1 = " +  
            circle1.radius + " circle2 = " + circle2.radius);  
    }  
  
    public static void swap1(Circle x, Circle y) {  
        Circle temp = x;  
        x = y;  
        y = temp;  
    }  
}
```

```
    public static void swap2(Circle x, Circle y) {  
        double temp = x.radius;  
        x.radius = y.radius;  
        y.radius = temp;  
    }  
}  
  
class Circle {  
    double radius;  
  
    Circle(double newRadius) {  
        radius = newRadius;  
    }  
}
```


Pregunta

Qué muestra lo siguiente
código?

```
public class Test {  
    public static void main(String[] args) {  
        int[] a = {1, 2};  
        swap(a[0], a[1]);  
        System.out.println("a[0] = " + a[0]  
            + " a[1] = " + a[1]);  
    }  
  
    public static void swap(int n1, int n2) {  
        int temp = n1;  
        n1 = n2;  
        n2 = temp;  
    }  
}
```

(a)

Pregunta

Qué muestra lo siguiente
código?

```
public class Test {  
    public static void main(String[] args) {  
        int[] a = {1, 2};  
        swap(a);  
        System.out.println("a[0] = " + a[0]  
            + " a[1] = " + a[1]);  
    }  
  
    public static void swap(int[] a) {  
        int temp = a[0];  
        a[0] = a[1];  
        a[1] = temp;  
    }  
}
```

(b)

Pregunta

Qué muestra lo siguiente
código?

```
public class Test {  
    public static void main(String[] args) {  
        T t = new T();  
        swap(t);  
        System.out.println("e1 = " + t.e1  
            + " e2 = " + t.e2);  
    }  
  
    public static void swap(T t) {  
        int temp = t.e1;  
        t.e1 = t.e2;  
        t.e2 = temp;  
    }  
}  
  
class T {  
    int e1 = 1;  
    int e2 = 2;  
}
```

(c)

Pregunta

Qué muestra lo siguiente
código?

```
public class Test {  
    public static void main(String[] args) {  
        T t1 = new T();  
        T t2 = new T();  
        System.out.println("t1's i = " +  
            t1.i + " and j = " + t1.j);  
        System.out.println("t2's i = " +  
            t2.i + " and j = " + t2.j);  
    }  
}  
  
class T {  
    static int i = 0;  
    int j = 0;  
  
    T() {  
        i++;  
        j = 1;  
    }  
}
```

(d)

Pregunta

Qué muestra lo siguiente
código?

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = null;
        m1(date);
        System.out.println(date);
    }

    public static void m1(Date date) {
        date = new Date();
    }
}
```

(a)

Pregunta

Qué muestra lo siguiente
código?

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date(1234567);
        m1(date);
        System.out.println(date.getTime());
    }

    public static void m1(Date date) {
        date = new Date(7654321);
    }
}
```

(b)

Pregunta

Qué muestra lo siguiente
código?

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date(1234567);
        m1(date);
        System.out.println(date.getTime());
    }

    public static void m1(Date date) {
        date.setTime(7654321);
    }
}
```

(c)

Pregunta

Qué muestra lo siguiente
código?

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date(1234567);
        m1(date);
        System.out.println(date.getTime());
    }

    public static void m1(Date date) {
        date = null;
    }
}
```

(d)

Ocultación de atributos

- Sabemos que dos variables declaradas en ámbitos nidados no pueden tener el mismo identificador ya que esto genera un error.

```
public static void main(String[] args){  
    int valor = 3;  
    for(int i = 0; i < 100; i++){  
        double valor = i;  
    }  
}
```

- Sin embargo, existe una excepción cuando una variable local en un método tiene el mismo identificador que un atributo de la clase. En este caso, dentro del método, la variable local tiene prioridad sobre el atributo . que la variable local oculta en el atributo

```
public class Persona {  
    int edat; // atribut enter  
  
    void metode() {  
        double edat; // variable local  
        edat = 8.2; // oculta l'atribut "edat"  
        System.out.println(edat);  
    }  
}
```


La palabra reservada this

- Especialmente útil para tratar la manipulación de atributos.
- Tiene dos finalidades:
 - **Dentro de los métodos no estáticos:** para hacer referencia al objeto sobre el que se ejecuta el método (Es recomendable siempre utilizarlo para evitar conflictos)

Por ejemplo: `this.edad` o `this.nombre`.

- **Dentro del método constructor:** para invocar a otro constructor de igual clase (constructor sobrecargado).

Por ejemplo: `this(edad, nombre)` invocaría al constructor `Persona(int edad, String nombre);`

Ejemplo

- Creamos un constructor de Persona que se creará a partir de otra (constructor copia):

```
public Persona (Persona p) {  
    this(p.dni, p.nombre, p.edad);  
}
```


Ejemplo

- Supongamos que creamos un método que crea una persona a partir de otra llamada clonar:

```
public Persona clonar() {  
    return new Persona(this);  
}
```



Ejemplo


```
public static void main(String args[]) {  
    Persona p1 = new Persona("000000000","Pepe Gotera",33);  
    Persona p2 = new Persona(p1);  
    Persona p3 = p1.clonar();  
    System.out.println("Visualización de persona p2:");  
    p2.visualizar();  
    System.out.println("Visualización de persona p3:");  
    p3.visualizar();  
}
```


¿p1, p2 y p3 hacen referencia al mismo objeto u objetos distintos?

Práctica 4

- Modifica la clase Persona, anteponiendo el acceso de cualquiera de sus atributos, la palabra reservada this.

Práctica 5



- Amplía el método main que tienes en este momento, para que se cree una persona a partir de otra, haciendo uso del método "clonar" y comprueba que funciona correctamente mostrando los valores de la nueva persona creada.
 - Crea a una nueva persona haciendo uso directamente del "constructor copia" y muestra los atributos de la persona creada.
- 

Práctica 6

- Crea un nuevo constructor en Persona, que creará una nueva persona a partir de otros dos. En su caso, el dni y el nombre de la nueva persona creada serán los de la persona1 y la edad la de la persona 2.

Práctica 7

- Por último, crea una nueva versión del método clonar que utilizará el constructor creado anteriormente. En este caso, la persona1 que necesita este constructor, será el objeto que efectúa la operación y la persona2 se recibirá como parámetro.
- Invoca este método desde el método main y comprueba que funciona correctamente.

Ejercicio “CuentaCorrente” (IV)

Sobrecarga la clase CuentaCorrent para poder crear objetos con:

- El DNI, el nombre del titular y el saldo inicial .
- El DNI del titular y un saldo inicial.

Escribe un programa que compruebe el funcionamiento de los métodos.

() Utiliza también “this” para acceder a los atributos de instancia.*

Ejercicio "CuentaCorrente" (V)

Existen gestores que administran las cuentas bancarias y atienden a sus propietarios . Cada cuenta tiene un único gestor (o ninguno) .**Gestor**de la que nos interesa guardar su nombre, teléfono, y el importe máximo autorizado con el que puede operar. Respecto a los gestores existen las siguientes restricciones:

- Un gestor tiene siempre un nombre y un teléfono.
- Si no se asigna, el importe máximo autorizado por operación será de 10000 euros.
- Un gestor, después de ser creado no podrá cambiar su número de teléfono y todo el mundo podrá consultarlo.
- El nombre del gestor será público y el importe máximo sólo será visible para clases vecinas.

Modi fi ca la classe CompteCorrent para que pueda disponer de un objeto Gestor. Escr y los métodos necesarios.