J2SE
P.O.O en Java

Antonio Espín Herranz

Repaso de la P.O.O.

- Se rompe la dependencia con la máquina y se piensa mas como en el mundo real.
- Sigue siendo imperativa.
- Es más fácil realizar modificaciones, así como la integración de las distintas partes.

Conceptos y Características

CONCEPTOS

- Clase
- Objeto
- Instancia
- Método
- Propiedad / Atributo
- Colección

CARACTERÍSTICAS

- Herencia
- Polimorfismo
- Abstracción
- Encapsulación
- Reutilización
- Modularización

Ejemplo de la P.O.O.

- Supongamos una fábrica de coches. En esta se define todas las características y funcionamiento de los coches que se va a construir → **Clase**.

- Cada Coche (**Objeto**) se creará con la definición de la clase → se instanciará con una serie de valores. Tendrá una serie de propiedades.

- Propiedades del Coche → **Atributos**.
  - Color, número de puertas, potencia.
  - Las ruedas: Son todas iguales, que tendrán una presión, modelo, marca. Se pueden agrupar en una **Colección**. Colección de Objetos Rueda.

- Y una serie de servicios / comportamiento → **Métodos**
  - frenar(), arrancar(), parar(), acelerar()

Definiciones

- Clase: Plantilla que define las características de un objeto.
- Objeto: Es un elemento que pertenece a una clase que tiene una serie de propiedades y un comportamiento propio.
- Instancia: Representa la creación de un objeto a partir de una clase. Objeto = Instancia.
- Propiedad: Es una característica de un objeto.
- Método: Es un servicio o comportamiento que nos ofrece un objeto.
- Colección: Un conjunto de objetos que en nuestro caso pertenecerán a la misma clase o a clases derivadas.

Objetos y Clases

- Definición de una clase en Java:

```
class NombreClase {
    // Definición de atributos.
    // Definición e implementación de métodos
}
```

// Seguir las convecciones de Java, el nombre de la clase en Mayúsculas.
// Los nombres de las variables y objetos en minúscula.

Para definir objetos de una clase:
NombreClase nombreObjeto;

Atributos y métodos

- Visibilidad de atributos y métodos:
  - **public**: Se permite el acceso a la parte pública de un objeto desde cualquier parte de la aplicación.
  - **private**: Solo se permite el acceso a la parte privada desde la clase que lo define.
  - **protected**: Se permite el acceso a la parte protegida desde la clase que lo define y subclases, así como las clases que pertenecen al mismo paquete.
  - **static**: Se accede a los métodos de una clase sin crear una instancia de la misma. Para hacer referencia a un método estático se precede el nombre de la clase:
    - NombreClase**.**nombreMétodoEstatico(…);
  - **Acceso por defecto**: si no indicamos ningún modificador, el acceso será a nivel de paquete. Al atributo / método declarado de esta forma podría acceder desde cualquier clase que se encuentre en el mismo paquete.

Definición de atributos

ámbito tipo identificador;

private int edad;

Podemos usar los tipos primitivos o una clase.

public Punto2D p;

// Mantener las convecciones: Usar letras minúsculas y para palabras compuestas, separar mediante mayúsculas. Ejemplos: edad, fechaNacimiento.

- Tener en cuenta lo de antes, el **ámbito no es obligatorio** → acceso package.

Definición de métodos

ámbito tipo nombreMétodo([argumentos])

```
{
    sentencia1;
    sentencia2;
    …
}
```

// **Seguir las mismas normas para nombrar a los métodos que en los atributos.**

- **Tipo:** indica el tipo que devuelve el método. Puede ser un tipo primitivo o un objeto de una clase.
- **Lista de argumentos:** el tipo y el nombre de cada uno de los argumentos separados por comas.

Procedimientos vs Funciones

- **Procedimiento:** conjunto de sentencias que no devuelven nada. En Java lo representamos por el tipo **void**.

- **Función**: similar al procedimiento pero nos devuelven un valor, para devolver el valor usaremos la palabra reservada **return**.

Argumentos por Valor y por Referencia

- **Por valor**: se modifica dentro del método pero fuera de este su valor no se ve alterado. En realidad se obtiene una copia que es la que se modifica, el valor original no se altera. Los tipos primitivos van siempre por valor.

- **Por referencia**: se modifica su valor dentro y esta modificación también se realiza fuera, los tipos referenciados siempre se pasan por referencia.

Constructores

- Sirven para crear objetos de una clase.
- Estos métodos inicializan los objetos.
- Son métodos públicos.
- No devuelven nada, ni siquiera void.
- Su nombre coincide con la clase.
- Se pueden sobrecargar.
- Si no se implementa, el compilador crea el constructor por defecto.
- El constructor se ejecuta al crear el objeto con **new**.

Uso de new

- Ejemplo:

```
public class Punto2D {
    private int x, y;

    public Punto2D(){
      x = 0;
      y = 0;
    }
}

// Desde otra clase:
Punto2D p = new Punto2D();

// También es válido:
Punto2D p;
p = new Punto2D();
```

Sobrecarga de métodos

- Dentro de una misma clase podemos definir varios métodos con el mismo nombre. Podemos sobrecargar constructores y cualquier otro método.

- El compilador los resuelve por el número y tipo de los argumentos.
  - public Punto2D(); // Por defecto.
  - public Punto2D(int x, int y); // Con parámetros.
  - public Punto2D(Punto2D otroPunto); // Por copia de otro objeto.
  - **C O R R E C TO**

- **NO** podemos sobrecargar métodos solo con el tipo devuelto.
  - public int calcularArea();
  - public float calcularArea();
  - **E R R O R**

Acceso a los atributos / métodos de una clase

- Tenemos que distinguir si accedemos desde el ámbito de la clase (su propia definición) o desde otra clase.

- Desde la propia clase:
  - Accedemos directamente con el nombre del atributo / método.
  - Usando la palabra reservada **this**.

- Desde fuera:
  - En caso de que sea permisible, usaremos el operador **.**
  - nombreObjeto.nombreAtributo = …;
  - nombreObjeto.nombreMétodo();

Ejemplo de Acceso

```
class ClaseA {
    public int n;
    private int m;
    public void metodo1(){ … }
}
```

Desde la clase:

- Directamente → public void metodo1(){ n = 10; }
- con **this** → public void metodo1(){ this.n = 10; }

Desde fuera:

```
ClaseA objA = new ClaseA();
objA.n = 25;          // C O R R E C TO
objA.m = 15;          // E R R O R
objA.metodo1();       // C O R R E C T O
```

Uso típico de this

- Dentro de los constructores:

```
public class Punto2D {
    int x, int y;

    public Punto2D(int x, int y){
      // Puedo utilizar nombres iguales pero con this se distinguen entre los
      parámetros y los atributos de la clase.

      this.x = x;
      this.y = y;
    }
}

// Desde otra clase:
Punto2D p = new Punto2D(1, 2);
```

Destructores

- Java elimina automáticamente la memoria reservada para los objetos.
- Se eliminan de la memoria cuando ya no se utilizan.
- Si queremos realizar alguna acción en este momento usaremos el método finalize().
- El método **finalize**() se encuentra en la clase **Object**, tendríamos que sobrescribirlo.

- Cabecera:

```
public void finalize() { }
```

Acceso static

- Sólo pueden llamar a otros métodos static.
- Sólo deben acceder a datos declarados como static.
- No pueden referirse a this ni a super.
- Podemos definir variables static. Solo se inicializan la primera vez.
- Un bloque static solo se ejecuta una vez.

Ejemplo de static

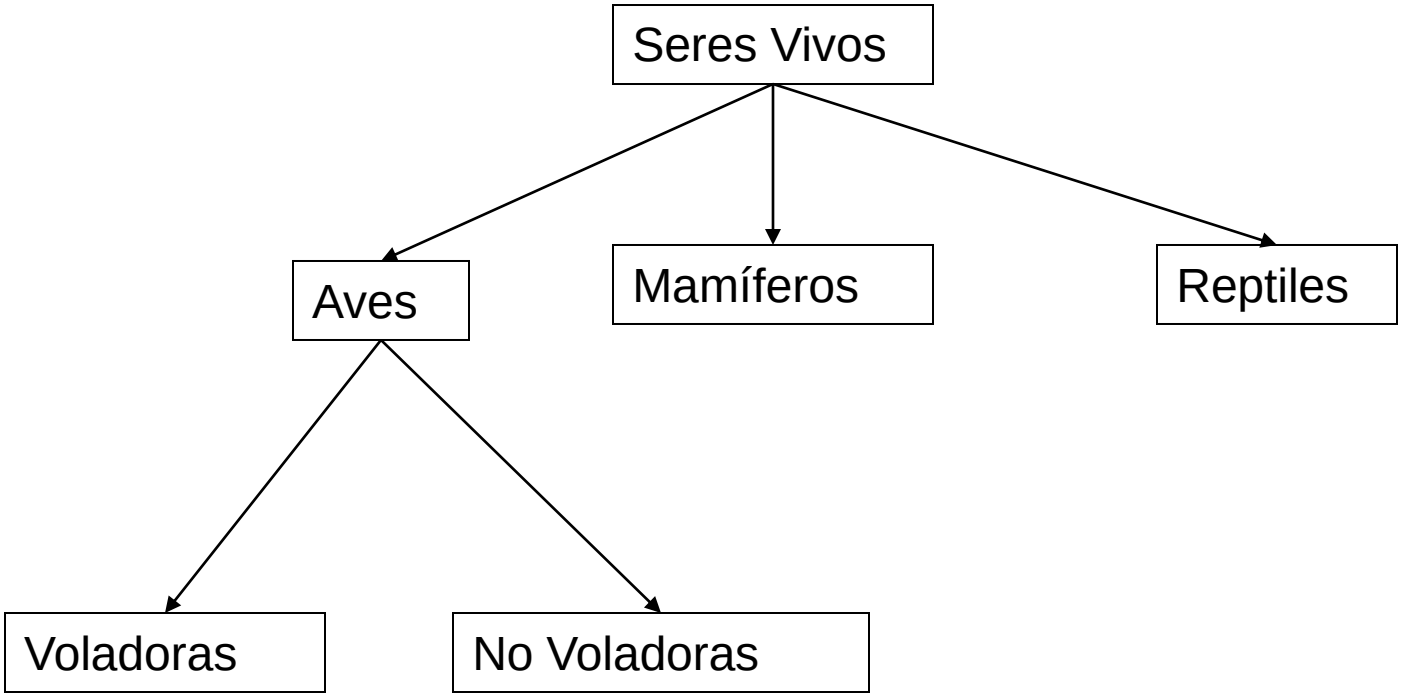
```
public class UsoStatic {

    static int a = 3;                                                    (1)
    static int b;                                                        (3)

    static void metodo(int x){
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {                                                             (2)
        System.out.println("Inicialización bloque static");
        b = a * 4;
    }

    public static void main(String args[]){                              (4)
        metodo(42);                                                      (5)
    }
}
```

public static void main(…)

- La función main es pública para poder ser accedida desde fuera.
- El cargador de clases no necesita crear una instancia de nuestra clase, llama directamente a la función main (static).
- Si llamamos a otro método desde main debe ser también static.

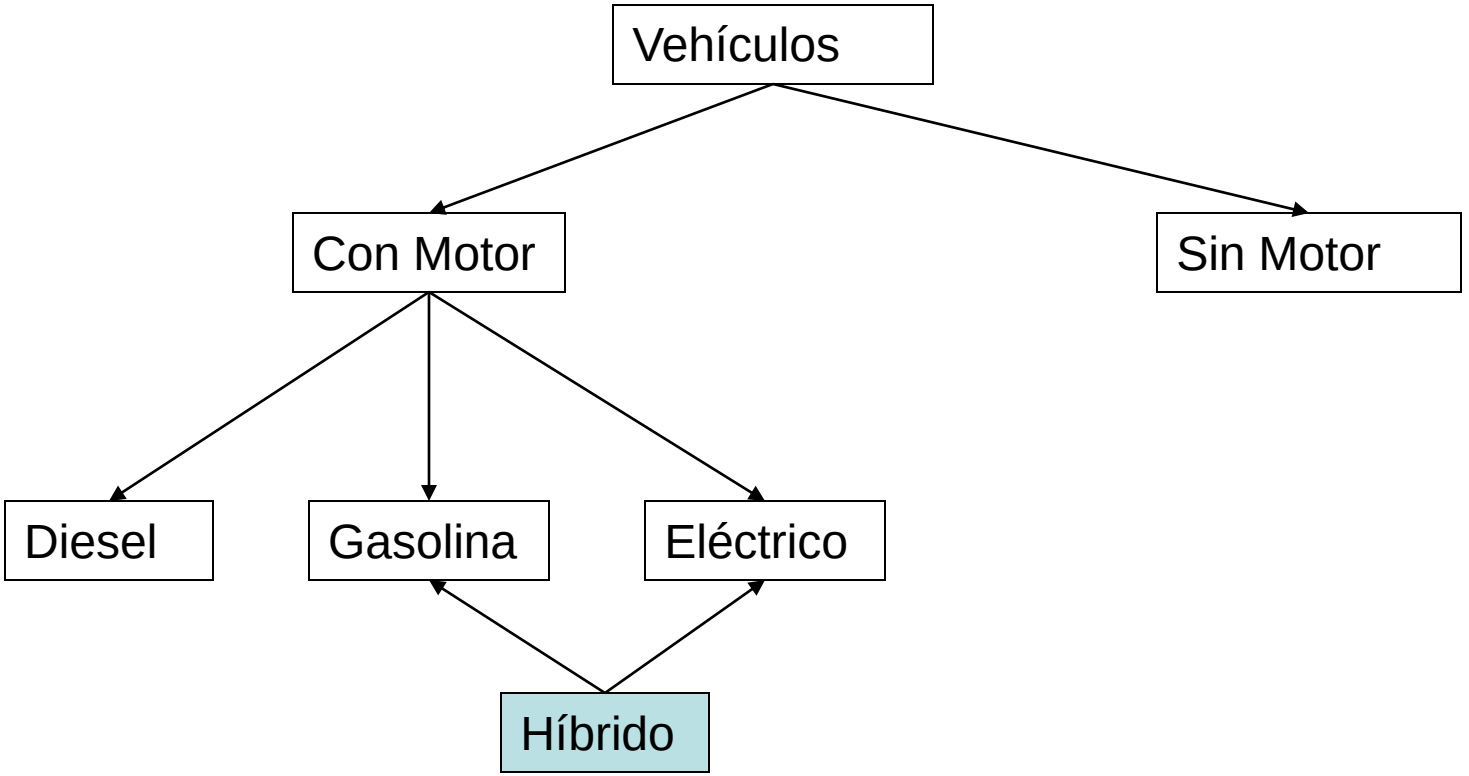
Herencia

- Con la herencia podemos representar jerarquía de clases: Herencia Simple. **Java la SOPORTA**.



Herencia

- Herencia múltiple. **Java la NO SOPORTA**.
- Híbrido depende de dos clases.



Herencia

- Permite que una clase obtenga la funcionalidad de otra clase.
- Cuando una clase hereda de otra: se hereda absolutamente TODO.
- Diremos que la clase B hereda de la clase A.
- La clase A se denomina clase Base, Padre o Superclase.
- La clase B se denomina clase Derivada, Hija o Subclase.
- Todo objeto de la clase B es un objeto de la clase A.

Herencia

- Sintaxis:

  class ClaseB **extends** ClaseA { … }

  - La ClaseB hereda toda la funcionalidad de la ClaseA (incluyendo sus atributos).

  - Podremos acceder directamente a los atributos de la ClaseA, siempre y cuando no sean privados.

  - En Java todas las clases (incluidas las que diseñemos) heredan de la clase Object.

Constructores en la Herencia

- Para llamar al constructor de la clase base usaremos la palabra clave: **super**.

- Dentro de los constructores de la clase Derivada, la primera llamada que haremos será a super() // El constructor de la superclase.

Ejemplo de Herencia

```
class ClaseA {
    private int a;

    public ClaseA(){
        a = 0;
    }

    public ClaseA(int v){
        a = v;
    }
}
```

```
class ClaseB extends ClaseA {
        private int b;

        public ClaseB(){
            super();
            b = 0;
        }
        public ClaseB(int v1, int v2){
            super(v1);
            b = v2;
        }
}
```

Redefinición de métodos

- Consiste en cambiar el comportamiento de un método heredado.
- No es lo mismo que la sobrecarga, que consiste en hacer un nuevo método.
- Ejemplo: Podemos redefinir el método toString() de la clase Object. Al llamar al método toString() se selecciona el de la clase derivada, no el de la superclase aunque con super los podemos llamar.

Polimorfismo

- Es la capacidad que tienen los objetos de comportarse de múltiples formas.

- Un objeto de una clase puede comportarse como lo que es, como un objeto de la clase a la que pertenece o como cualquiera de la clases bases.

Ejemplo de Polimorfismo

- Partimos de la siguiente relación de herencia:

  Clase Persona (nombre, apellidos, dirección)

  Clase Empleado Hereda de Persona

  (añade los atributos de departamento y código)

  Podemos decir que TODO objeto de la clase Empleado ES UN objeto de la clase Persona.

Codificación clase Persona

```
public class Persona {
    private String nombre;
    private String apellidos;
    private String direccion;

    public Persona(String nombre, String apellidos, String direccion){
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.direccion = direccion;
    }
    public String toString() { … }
}
```

Codificación clase Empleado

```
public class Empleado extends Persona {
    private String departamento;
    private String codigo;

    public Empleado(String nombre, String apellidos, String dir, String dpo, String cod){
        super(nombre, apellidos, dir);
        this.departamento = dpo;
        this.codigo = cod;
    }

    public String toString(){
        …
    }
}
```

Uso de ambas clases

```
class TestPolimorfismo {
    public static void main(String args[]){

        // ¿Son válidas ambas declaraciones?
        Persona objetos[] = new Persona[5];
        Persona objetos[] = new Empleado[5];

        objetos[0] = new Empleado(, , , );
        objetos[1] = new Persona(, , ,);

        for (i = 0 ; i < objetos.lenght ; i++ ){
                objetos[i].toString();
        }
    }
}
```

Encapsulamiento

- Se proporciona únicamente como público aquello que solo se desea que se pueda utilizar desde el exterior.

- Los objetos se ven como cajas negras que solo se pueden manipular a través de su interfaz, es decir, a través de los métodos que proporciona su clase.

- Las variables de instancia en la gran mayoría de los casos se declaran privadas y se definen una serie de métodos **get** / **set** para manipularlas pero no se permite que se trabaje con ellas libremente, siempre a través de los métodos de acceso a datos.

Ejemplo de Encapsulamiento

```
public class Punto2D {
    private int x, y;

    public Punto2D(){
       x = 0;
       y = 0;
    }

    public getX(){ return x; }
    public getY(){ return y; }

    public void setX(int x){ this.x = x; }
    public void setY(int y){ this.y = y;}
}
```

– No podemos manipular directamente los atributos del punto, solo a través de los métodos **set** y **get** definidos en la clase.

Superclases y Subclases

- Dentro de la jerarquía de clases del Empleado y la Persona.
- Empleado hereda de Persona.
- Diremos que Persona es la superclase de Empleado.
- Y que Empleado es la subclase de Persona.

Clases abstractas

- Tienen una funcionalidad definida pero no se puede implementar dicha funcionalidad → método.

- El método definido pero no implementado recibe el nombre de abstracto.

- Cabeceras de clases y métodos abstractos:

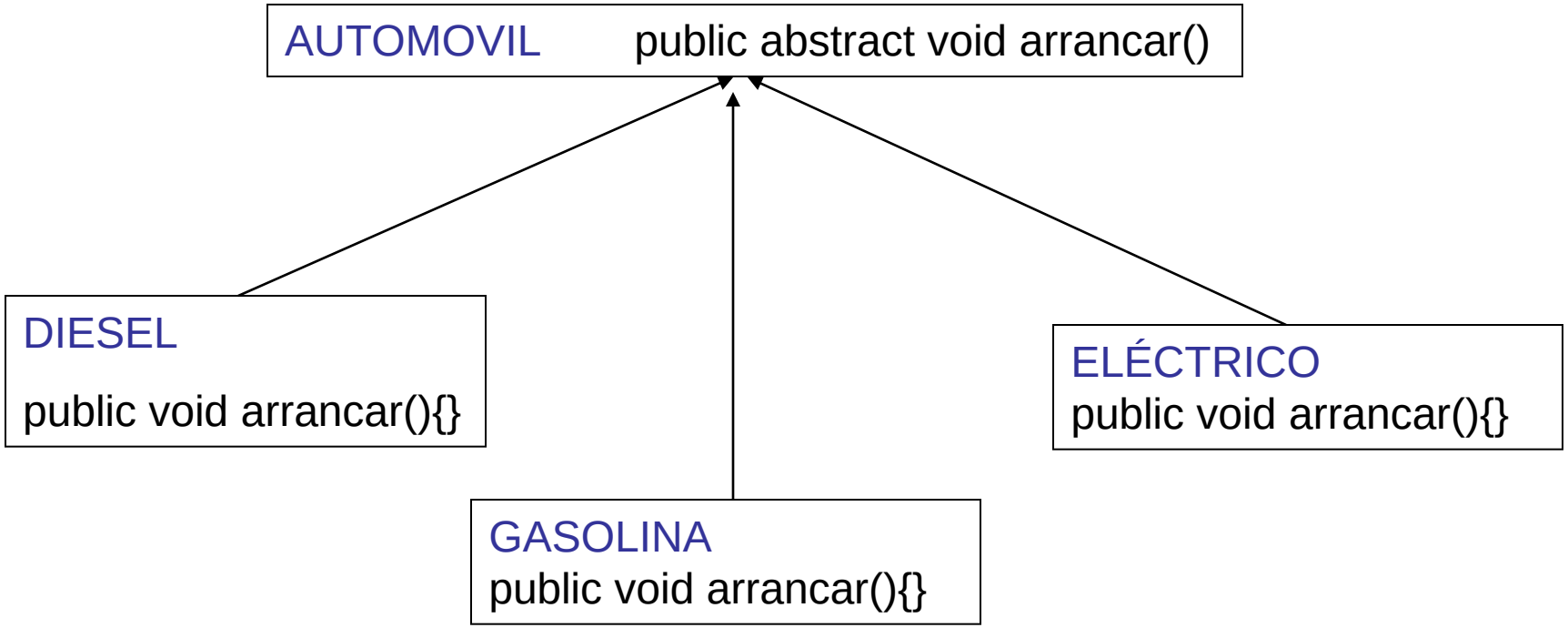
```
public abstract class ClaseAbstracta { …
        public abstract tipo metodoAbstracto();
}
```

Clases abstractas

- Una clase abstracta no se puede instanciar.
- Dentro de un árbol de herencia una clase abstracta nunca será una hoja.
- Los métodos abstractos también se heredan.
- Las subclases que heredan de una clase abstracta tienen que redefinir los métodos o propagarlos como abstract pero que sus subclases los redefinan.

Ejemplo de clase abstracta

Jerarquía de clases:



Ejemplo de clase abstracta

- La clase Automóvil define el método arrancar porque todos los coches tienen la funcionalidad de hacerlo.

- Lo que ocurre que cada tipo de coche lo hace de una forma distinta, de ahí que el método sea genérico a todas y cada subclase (Diesel, Gasolina, Eléctrico, …) sea una especialización.

Interfaces

- Es una forma que tiene de solucionar Java el problema de la Herencia Múltiple.
- Un interface define un comportamiento.
- Solo declara métodos no los implementa.
- Las clases que implementen un interface deberán codificar todos los métodos. Si no será una clase abstracta.
- No podemos instanciar un objeto de un interface.

Sintaxis de los interfaces

```
public interface NombreInterface {
    public tipo metodoUno( [argumentos]);
    public tipo metodoDos( [argumentos]);
    …
}
class MiClase implements NombreInterface { … }
class MiClase implements NombreInterface1, NombreInterface2 { . . . }
```

Se puede heredar de una clase e implementar un interface.

Un típico uso es en el caso de Applets y Trheads.

```
public class MiApplet extends Applet implements Runnable { … }
```

Clases Internas

- Una clase interna es una clase que está definida dentro de otra clase.
- Las clases internas se utilizan sobretodo en los eventos.
- Cuando se compilan estas clases la máquina virtual genera un class con el siguiente nombre: ClaseContenedora$ClaseInterna.class

Clase Interna

- Estas clases tienen la siguiente estructura:

```
class ClaseContenedora {
    …
    class ClaseInterna {
        …
    }
}
```

Tipos de Clases Internas

- Clases internas estáticas.
- Clases internas miembro.
- Clases internas locales.
- Clases anónimas.

Clases internas estáticas

- Se conocen también como clases anidadas (nested classes).
- Podemos tener clases o interfaces internos.
- Se definen fuera de cualquier método pero dentro de la clase externa.
- Se definen utilizando la palabra static.

Clases internas estáticas

```
public class Clase1 {

        static class Clase2 {
            …
        }

        // métodos de Clase1 …
}
```

Clases internas estáticas

- Las clases internas pueden ver los miembros (atributos y métodos) de la clase externa.

- Para declarar un objeto de una clase interna, hay que precederlo del nombre de la clase externa con un punto.
  - ClaseExterna.ClaseInterna objeto;

Ejemplo

```
public class ClaseExterna {

        private int a, b;
        private static int N = 0;

        public ClaseExterna(int a, int b){
        this.a = a;
        this.b = b;
        }
        static class ClaseInterna {

                private int c;

                public ClaseInterna(int c){
                this.c = c;
                }

                public int getC() {
                return c;
                }

                public void setC(int c) {
                this.c = c;
                }

                public void pruebas(){
                System.out.println(ClaseExterna.N);
                }
        }

}
```

```
public static void main(String[] args) {
ClaseExterna ce = new ClaseExterna(1, 2);
ClaseExterna.ClaseInterna ci = new
      ClaseExterna.ClaseInterna(3);

System.out.println(ci.getC());
}
```

Clases internas miembro

- Las clases internas o clases internas miembro, son clases definidas dentro de otra clase, pero fuera de cualquier método.

- No pueden contener miembros estáticos (ni atributos, ni métodos u otras clases internas static).

Clases internas miembro

- Un objeto de la clase interna está asociado a un único objeto de la clase externa.
- Un objeto de la clase Externa se puede asociar a varios objetos de la clase interna.
  - Las funciones de la clase interna puede ver los miembros de la clase externa (los privados también).
  - Las funciones de la clase Externa no pueden tocar directamente los miembros de la clase Interna, necesitan un objeto de esa clase.

Clases internas miembro

- Las clases internas pueden ser de acceso private o protected mientras que las otras son: public o package.
- Una clase interna miembro podría contener a otras clases internas.
- En la clase interna this hace referencia a la propia clase.
- Para referenciar a la clase externa:
  - ClaseExterna.this

- Para crear un objeto de la clase interna, se utiliza una referencia a la clase Externa.
  - ClaseExt **objExt** = new ClaseExt();
  - ClaseExt.ClaseInt obj = **objExt**.new ClaseInt();

Ejemplo

```
public class ClaseExterna {

private int a, b;
private static int N = 0;

public ClaseExterna(int a, int b){
this.a = a;
this.b = b;
}

        private class ClaseInterna {

        private int c;

        public ClaseInterna(int c){
        this.c = c;
        }

        public int getC() {
        return c;
        }

        public void setC(int c) {
        this.c = c;
        }

        public void pruebas(){
        System.out.println(ClaseExterna.N);
        System.out.println(ClaseExterna.this.a + " " +
              ClaseExterna.this.b);
        }


}
```

```
public static void main(String[] args) {
ClaseExterna ce = new ClaseExterna(1, 2);
ClaseExterna.ClaseInterna ci = ce.new
      ClaseInterna(3);

System.out.println(ci.getC());
}
```

Clases Internas Locales

- Este tipo de clases internas se definen dentro de un método.
- Su acceso es local, solo al método que las contienen.
- Se utilizan en la definición de eventos de las interfaces gráficas.
- Los objetos que creamos de estas clases se deben definir dentro del método que contiene a dicha clase.

Clases Internas Locales

- Las clases locales pueden trabajar con parámetros y variables definidas dentro del método, pero deben ser declaradas como final (no se pueden cambiar).
- No pueden tener el mismo nombre que sus clases contenedoras.
- No pueden definir métodos, variables y clases static.
- Tienen acceso a los atributos de la clase externa.

Ejemplo

```
public class ClaseExterna {

private int a, b;
private static int N = 0;

public ClaseExterna(int a, int b){
this.a = a;
this.b = b;
}

public void interna(){

        class ClaseInterna {

                private int c;

                public ClaseInterna(int c){
                this.c = c;
                }

                public int getC() {
                return c;
                }

                public void setC(int c) {
                this.c = c;
                }

                public void pruebas(){
                System.out.println(ClaseExterna.N);
                System.out.println(ClaseExterna.this.a + " " +
                        ClaseExterna.this.b);
        }
}

ClaseInterna ci = new ClaseInterna(6);
ci.pruebas();


}
```

```
public static void main(String[] args) {
ClaseExterna ce = new ClaseExterna(1, 2);
ce.interna();
}
```

Clases Internas Anónimas

- Son similares a las clases Internas pero no llevan nombre.
- En estas el proceso de definición de la clase y la creación de un objeto de la misma van a la par.
- Se definen dentro de una expresión new.
- Al igual que las anteriores tienen acceso a los atributos de la clase Externa.

Ejemplo

```
bCargar = new JButton();
getContentPane().add(bCargar);
bCargar.setText("Cargar");
bCargar.setBounds(379, 446, 87, 21);
bCargar.addActionListener(new ActionListener() {

    @Override
    public void actionPerformed(ActionEvent e) {
        // Instrucciones …

    }
});
```

Convenciones en la codificación en Java

- Clases: La primera en mayúscula.

  class AccountBook

  class ComplexVariable

- Interfaces: La primera en mayúscula.

  Interface Account

- Paquetes: La primera en minusculas.

  package paquete;

- Métodos: La primera en minúscula, segundas palabras en mayúsculas.

  balanceAccount ()

  addComplex ()

- Variables: Idem del anterior, pero sin paréntesis.

  int numero;

  String nombreApellidos;

Relaciones entre Clases

- Se dan 3 tipos de relaciones entre clases en un diseño OO:
  - Relación de **Herencia**:
    - La clase Empleado hereda de Persona.
  - Relación de **Composición**:
    - Podemos tener atributos dentro de una clase que sean objetos de otra clase.
      - Un Coche se compone de una Radio, un Motor, etc. Siendo Radio y Motor otras clases previamente diseñadas.
  - Relación de **Asociación** o de Uso:
    - Los tipos de Relación son duraderas en el tiempo, en este caso sólo se establece en un momento puntual en el tiempo.
    - Este tipo de relación las vamos a identificar con objetos que entran como parámetros en las llamadas a los métodos de otra clase:
    - Ejemplo:

```
Coche miCoche = new Coche();
Surtidor s = new Surtidor();

miCoche.repostar(s);
```