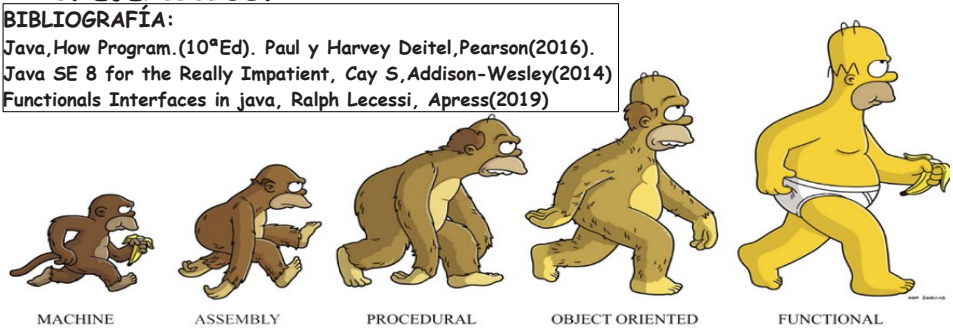
UNIDAD 5

PROGRAMACIÓN ORIENTADA A OBJETOS AVANZADA

BIBLIOGRAFÍA:

Java,How Program.(10ªEd). Paul y Harvey Deitel,Pearson(2016).
Java SE 8 for the Really Impatient, Cay S,Addison-Wesley(2014)
Functionals Interfaces in java, Ralph Lecessi, Apress(2019)



5.1 RELACIONES ENTRE CLASES.

Cuando definimos el concepto de clase, la describimos principalmente como una especie de plantilla o molde usado para instanciar (construir) un objeto: un **mecanismo de definición de objetos**.

Por tanto, a la hora de diseñar un conjunto de clases para **modelar** (representar) el conjunto de información cuyo tratamiento se desea automatizar en un programa, es importante establecer de forma adecuada las posibles relaciones que puedan existir entre unas clases y otras.

En algunos casos es posible que no exista relación alguna, pero también es habitual que sí la haya: una clase puede ser una **especialización de otra** (*relación entre dos clases donde una de ellas (**la subclase**) es una versión más especializada de la otra (**la superclase**), compartiendo características en común pero añadiendo o modificando otras características específicas que la especializan.*

*El punto de vista inverso de la especialización sería la **generalización** (relación entre dos o más clases donde una de ellas (la superclase) es una versión más genérica de las otras (las subclases), compartiendo las características comunes de las hijas pero sin las propiedades específicas que caracterizan a cada subclase (el punto de vista inverso sería la **especialización**).*

Otras relaciones consisten en que **una clase contiene en su interior objetos de otra**, o **una clase utiliza a otra**, etc. Es decir, que entre unas clases y otras habrá que definir cuál es su relación (si es que existe alguna). También hay relaciones entre objetos.

Se pueden distinguir diversos tipos de relaciones entre clases:

- **Clientelar**. Una clase utiliza objetos de otra clase (por ejemplo al pasarlos como parámetros a través de sus métodos).
- **Composición**. Alguno de los atributos de una clase son un objeto de otra clase.
- **Anidamiento**. Se definen clases en el interior de otra.
- **Herencia**. Una clase comparte determinadas características con otra (clase base), añadiéndole o modificando alguna funcionalidad específica (especialización).

La relación de **clientela** la llevas utilizando desde que has empezado a programar en Java, pues desde tu clase principal (clase con método main) has estado declarando, creando y utilizando objetos de otras clases. Por ejemplo: si utilizas un objeto **String** dentro de la clase principal de tu programa, éste será cliente de la clase **String** (como sucederá con prácticamente cualquier programa Java).

Es la relación fundamental y más habitual entre clases (la utilización de unas clases por parte de otras) y, por supuesto, la que más vas a utilizar tú también, de hecho, ya la has estado utilizando y lo seguirás haciendo.

EJEMPLO 1: La clase `C1` es cliente de la clase `String` y de la clase `System` porque las utiliza en su código.

```
public class C1 {
    public static void main(String[] args) {
        System.out.println("Hola, tengo " + args.length +
                           "argumentos");
    }
}
```

La relación de **composición** se produce cuando una clase está compuesta por (variables miembro) otros objetos en su interior, lo cual

es bastante habitual.

EJEMPLO 2: La clase `Persona` está compuesta por `String` y `LocalDate`.

```
public class Persona {
    private String nombre;
    private LocalDate nace;
    // Otros elementos...
}
```

La relación de **anidamiento** (o **anidación**) es quizás menos habitual, pues implica declarar unas clases dentro de otras (**clases internas** o **anidadas**). En algunos casos puede resultar útil para tener un nivel más de **encapsulamiento** (*ocultamiento del estado de un objeto (de sus datos miembro o atributos) de manera que sólo se puede cambiar mediante las operaciones (métodos) definidas para ese objeto. Cada objeto está aislado del exterior de manera que se protegen los datos contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos colaterales no deseados.*

Este modo de proceder permite que el usuario de una clase pueda obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de manera imprevista e incontrolada) y **ocultación** (efecto que se consigue gracias a la **encapsulación**: se evita la visibilidad de determinados miembros de una clase al resto del código del programa para de ese modo comunicarse con los objetos de la clase únicamente a través de su interfaz (métodos).

El caso de la relación de **herencia** también es de suma importancia, pues ofrece un mecanismo que aumenta la productividad al ahorrarnos reprogramar de forma innecesaria código.

Podría decirse que tanto la **composición** como la **anidación** son casos

particulares de **clientela**, pues en realidad en todos esos casos una clase está haciendo uso de otra (al contener atributos que son objetos de la otra clase, al definir clases dentro de otras clases, al utilizar objetos en el paso de parámetros, declarar variables locales utilizando otras clases, etc.).

En los siguientes apartados del tema veremos estas relaciones junto con otros elementos de la POO y de la programación funcional.

5.2 COMPOSICIÓN DE CLASES.

Para indicar que una clase contiene objetos de otra clase no es necesaria ninguna sintaxis especial. Cada uno de esos objetos no es más que un atributo (variable miembro) y debe ser declarado como tal:

```
class nombreClase {
   [modificadores] NombreClase1 nombreAtributo1;
   [modificadores] NombreClase2 nombreAtributo2;
   ...
}
```

EJEMPLO 3: La clase `Punto` define las coordenadas de un punto en un plano 2D.

```
class Punto {
   private double x;
   private double y;
   public Punto(double unX, double unY) {
      x = unX;
      y = unY;
   }
   public double getX() { return x; }
   public double getY() { return y; }
}
```

La clase **Rectangulo** define una figura en el plano 2D a partir de dos de sus vértices o esquinas que serían de tipo Punto.

- vertice1 = la esquina superior izquierda
- vertice2 = la esquina inferior derecha del cuadrado.

Tal y como hemos formalizado ahora los tipos de relaciones entre clases, parece bastante claro que aquí tendrías un caso de composición: "un rectángulo está formado por 2 puntos". Por tanto, podrías definir los atributos de la clase Rectangulo como dos objetos de tipo Punto.

EJERCICIO 1: Define la clase Rectangulo usando composición de la clase Punto e implementa además del constructor los siguientes métodos públicos:

1. `getSuperfice()` calcula y devuelve el área encerrada por la figura.
2. `getPerimetro()` calcula y devuelve la longitud del perímetro de la figura.
3. `esCuadrado()` que devuelve true si el rectángulo es un cuadrado.

PRESERVAR LA OCULTACIÓN

La relación de composición no tiene más misterio a la hora de implementarse que simplemente declarar variables miembro cuyo tipo sea de las clases que necesites.

Ahora bien, debe tener precaución con aquellos métodos que intercambien estos atributos con el exterior, ya sea porque los sacan al exterior (getters) o porque los introducen desde el exterior (setters y constructores).

Lo habitual suele ser declarar los atributos de la clase como privados (o protegidos) para ocultar la implementación. Para que otros objetos puedan acceder a la información, o al menos a una parte de ella, deberán hacerlo a través de métodos que sirvan de interfaz, de

manera que sólo se podrá tener acceso a aquella información que el creador de la clase haya considerado oportuno.

Del mismo modo, los atributos solamente serán modificados desde los métodos de la clase, así el creador de la clase controla cómo y bajo qué circunstancias deben realizarse esas modificaciones. Con esa metodología de acceso se tenía perfectamente separada la parte de manipulación interna de los atributos de la interfaz con el exterior.

Si los métodos de tipo get devuelven tipos primitivos, devuelven copias del contenido que había almacenado en los atributos, pero los atributos siguen "a salvo" como elementos privados de la clase.

Cuando vayas a devolver un objeto mutable debes ir con más precaución. Si devuelves directamente un atributo que es un objeto mutable, estarás ofreciendo directamente una referencia a un objeto que probablemente has definido como privado. **¡Estás volviendo a hacer público un atributo que querías definir como privado!**

Para evitar ese tipo de situaciones (ofrecer al exterior referencias a objetos mutables privados) puedes optar por diversas alternativas, procurando siempre evitar la devolución directa de su referencia:

- Una opción podría ser devolver solamente tipos primitivos.
- Dado que esto no es siempre posible, o como mínimo poco práctico, otra posibilidad es crear un nuevo objeto que sea una copia del objeto que quieres devolver y utilizar ese objeto como valor de retorno. Es decir, crear una copia del objeto exclusivamente para devolverlo. De esta manera, el código cliente podrá manipular a su antojo ese nuevo objeto, pues no será el original, sino un nuevo objeto con el mismo contenido.
- Por último, debes tener en cuenta que es posible que en algunos

casos sí se necesite realmente la referencia al atributo original (algo muy habitual en el caso de atributos estáticos). En tales casos, no habrá problema en devolver directamente el atributo al código cliente.

En resumen: evita devolver directamente un atributo que sea un objeto mutable (estás dando al exterior su referencia y haciéndolo visible y manipulable desde fuera), salvo que sea lo que necesites.

LOS GETTERS SON PELIGROSOS

Para entender estas situaciones un poco mejor, podemos volver al objeto Rectangulo y observar sus getters.

```
public Punto getVertice1() { return vertice1; }
public Punto getVertice2() { return vertice2; }
```

Esto funciona perfectamente, pero debes tener cuidado con este tipo de métodos que devuelven directamente una referencia a un objeto atributo que probablemente has definido como privado. Estás de alguna manera, haciendo público el atributo que se declara privado.

Para evitarlo basta con crear un nuevo objeto que sea una copia del objeto a devolver (en este caso un objeto de la clase Punto).

EJEMPLO 4: Para la clase `Rectangulo`, escribimos su método `getVertice1()` para que devuelva el vértice superior izquierdo del rectángulo (objeto de tipo `Punto`) de forma segura.

```
public Punto getVertice1 () { // Creación de un nuevo punto
   double x, y;
   Punto nuevoPunto;
   x = vertice1.getX();
   y = vertice1.getY();
   nuevoPunto = new Punto(x, y);
   return nuevoPunto;
}
```

EJERCICIO 2: Define tu el getter del vertice2, pero intenta minimizar los pasos intermedios empleados en el ejemplo 4 (que se ha escrito de esa forma para hacerlo más comprensible).

CONSTRUCTOR DE COPIA

Uno de los constructores que podemos implementar en nuestras clases para ayudar a solucionar de forma cómoda el problema de la pérdida de modificadores de acceso que aparece en la composición (si nuestra clase se usa como una parte de otras) es el constructor de copia. En este fragmento de código, la clase `Parte` se va a utilizar para componer o definir la clase `Todo` (`Todo` está compuesto de `unaParte`). El getter de `Todo` está mal implementado y al devolver directamente la referencia al objeto `unaParte`, está perdiendo el `private` (comparte objeto mutable con el exterior).

```
class Parte { ... }  // Objetos mutables
class Todo {
   private Parte unaParte;
   // Otras cosas de Todo...
   public Parte getUnaParte(){ return unaParte; }
}
```

Para implementarlo bien, debería hacer una copia del objeto y devolver la copia, no su objeto:

```
public Parte getUnaParte(){ return UNA_COPIA_DE(unaParte); }
```

Bien, pues una forma sencilla de hacer copias de objetos de la clase `Parte`, es implementar en la clase `Parte` un constructor de copia, que recibe un objeto de esta clase y crea uno nuevo que es una copia suya.

```
class Parte {
   ...
   // Este es el constructor de copia. this es el objeto copia
   public Parte(Parte original) {
```

```
            this.dato1 = original.dato1; ...
        }
    }
```

De esta manera, se devuelve un punto totalmente nuevo que podrá ser manipulado sin ningún temor por parte del código cliente de la clase pues es una copia especialmente creada para él, el rectángulo sigue controlando sus vértices y el cliente puede manipular las copias -> todos contentos.

EJEMPLO 5: Vuelve a implementar el getter del ejemplo 4 de forma segura usando el constructor de copia de la clase `Punto` que se tendrá que definir.

```
class Punto {
    ...
    // Constructor de copia
    public Punto(Punto p) { this.x = p.x; this.y = p.y; }
}

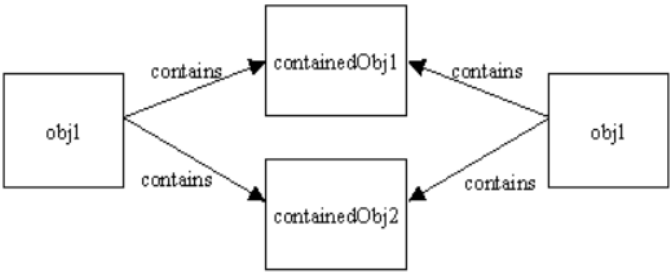
class Rectangulo {
    ...
    public Punto getVertice1 () { return new Punto(vertice1); }
}
```

EJERCICIO 3: Define tu el getter del vertice2, pero en vez de utilizar el constructor de copia, busca un método que herede de una clase padre.

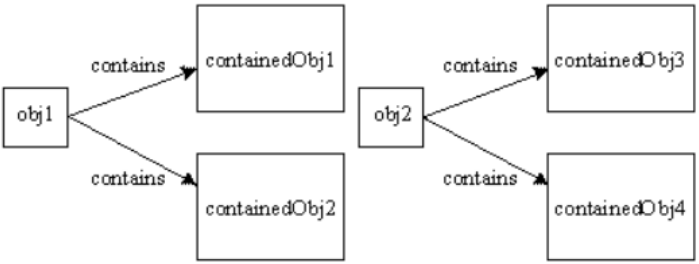
- ¿Tiene algún inconveniente?
- Si lo hay, ¿Cómo se podría solucionar?

Piensa en estas figuras que representan la parte izquierda un objeto original, y la parte derecha una copia del objeto original de la izquierda:

Copia superficial (shallowcopy):



Copia en profundidad (deepCopy):



LOS CONSTRUCTORES SON PELIGROSOS

Otro factor que debes considerar, a la hora de escribir clases compuestas de objetos de otras clases, es su comportamiento a la hora de instanciarse. Durante el proceso de creación de un objeto (constructor) de la clase contenedora habrá que tener en cuenta también la creación (llamadas a constructores) de aquellos objetos que estén contenidos. El constructor de la clase contenedora debe invocar a los constructores de las clases de los objetos contenidos.

En este caso hay que tener cuidado con las referencias a objetos que se pasan como parámetros para rellenar el contenido de los atributos.

Es conveniente hacer una copia de esos objetos y utilizar esas copias

pues si se utiliza directamente la referencia que llega del exterior como parámetro, **el código exterior de la clase tiene acceso a ella sin necesidad de pasar por la interfaz de la clase** (volveríamos a dejar abierta una puerta pública a algo que quizá esté protegido como private, package o protected).

Además, si el objeto parámetro que se pasó al constructor formaba parte de otro objeto, esto podría ocasionar un desagradable efecto colateral si esos objetos son modificados en el futuro desde el código cliente de la clase, ya que no sabes de dónde provienen esos objetos, si fueron creados especialmente para ser usados por el nuevo objeto o si pertenecen a otro objeto que podría modificarlos más tarde. Es decir, **corres el riesgo de estar "compartiendo" esos objetos con otras partes del código, sin ningún tipo de control de acceso** y con las nefastas consecuencias que eso podría tener: cualquier cambio de ese objeto afectaría a partes del programa supuestamente independientes, que entienden ese objeto como suyo (**creas código acoplado**).

En el fondo los objetos no son más que variables de tipo referencia a la zona de memoria en la que se encuentra toda la información del objeto en sí mismo. Esto es, puedes tener un único objeto y múltiples referencias a él. Pero sólo se trata de un objeto y cualquier modificación desde una de sus referencias afectaría a todas las demás, pues estamos hablando del mismo objeto.

Recuerda también que sólo se crean objetos cuando se llama a un constructor (uso de **new**). Si realizas asignaciones o pasos de parámetros, no se están copiando o pasando copias de los objetos, sino simplemente de las referencias y por tanto se tratará siempre del mismo objeto.

Se trata de un efecto similar al que sucedía en los métodos get, pero en este caso en sentido contrario (en lugar de que nuestra clase "regale" al exterior uno de sus atributos objeto mediante una referencia, en esta ocasión se "adueña" de un parámetro objeto que probablemente pertenezca a otro objeto y que es posible que en el futuro haga uso de él).

Para entender mejor estos posibles efectos podemos continuar con el ejemplo de la clase `Rectangulo` que contiene en su interior dos objetos de la clase `Punto`. En los constructores del rectángulo habrá que incluir todo lo necesario para crear dos instancias de la clase `Punto` evitando las referencias a parámetros (haciendo copias).

EJEMPLO 6: Intenta reescribir los constructores de la clase `Rectangulo` teniendo en cuenta su estructura de atributos (dos objetos de la clase `Punto`):

1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean (0,0) y (1,1).
2. Un constructor con cuatro parámetros (x1, y1, x2, y2) que cree un rectángulo con los vértices (x1, y1) y (x2, y2).
3. Un constructor con dos parámetros (punto1, punto2) que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
4. Un constructor con dos parámetros (base, altura), que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición (0,0) y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.
5. Un constructor de copia (recibe un Rectángulo y crea una copia).

Durante el proceso de creación de un objeto (constructor) de la clase contenedora (en este caso Rectangulo) hay que tener en cuenta también la creación (llamada a constructores) de aquellos objetos que son contenidos (en este caso objetos de la clase Punto). En el caso del primer constructor, habrá que crear dos puntos con las coordenadas (0,0) y (1,1) y asignarlos a los atributos correspondientes (vertice1 y vertice2):

```
public Rectangulo() {
   vertice1 = new Punto(0, 0); // El exterior no envía objetos
   vertice2 = new Punto(1, 1);
}
```

Para el segundo constructor habrá que crear dos puntos con las coordenadas x1, y1, x2, y2 que han sido pasadas como parámetros:

```
public Rectangulo(double x1, double y1, double x2, double y2) {
   vertice1 = new Punto(x1, y1); // El exterior no envía objetos
   vertice2 = new Punto(x2, y2);
}
```

En el caso del tercer constructor puedes utilizar directamente los dos puntos que se pasan como parámetros para construir los vértices del rectángulo. Ahora bien, esto podría ocasionar un efecto colateral no deseado si esos objetos de tipo Punto son modificados en el futuro desde el código cliente del constructor (no sabes si esos puntos fueron creados especialmente para ser usados por el rectángulo o si pertenecen a otro objeto que podría modificarlos más tarde).

Por tanto, para este caso quizá sea recomendable crear dos nuevos puntos a imagen y semejanza de los puntos que se han pasado como parámetros. Para ello tendrías dos opciones:

1. Llamar al constructor de la clase Punto con los valores de los atributos (x, y).

2. Llamar al constructor copia de la clase Punto, si es que se dispone de él.

Aquí tienes las posibles versiones:

Se guardan las referencias que vienen como parámetros, comparte los objetos Punto con el exterior (dejan de ser private aunque se declaren así):

```
public Rectangulo(Punto vertice1, Punto vertice2) {
   this.vertice1 = vertice1; // this se refiere a este objeto
   this.vertice2 = vertice2; // si no tiene this es el parámetro
}
```

Constructor que "extrae" los atributos de los parámetros y crea nuevos objetos (ahora ya no comparte con el exterior):

```
public Rectangulo(Punto vertice1, Punto vertice2) {
   this.vertice1 = new Punto(vertice1.getX(), vertice1.getY() );
   this.vertice2 = new Punto(vertice2.getX(), vertice2.getY() );
}
```

Constructor que crea los nuevos objetos mediante el constructor copia de la clase Punto (tampoco comparte con el exterior):

```
public Rectangulo(Punto vertice1, Punto vertice2) {
   this.vertice1 = new Punto( vertice1 );
   this.vertice2 = new Punto( vertice2 );
}
```

En este segundo caso puedes observar la utilidad de los constructores de copia a la hora de tener que clonar objetos (algo muy habitual en las inicializaciones).

Para el caso del constructor que recibe como parámetros la base y la altura, habrá que crear vértices con valores (0,0) y (0 + base, 0 + altura), o lo que es lo mismo: (0,0) y (base, altura).

```
public Rectangulo(double base, double altura) {
   this.vertice1 = new Punto(0, 0);
   this.vertice2 = new Punto(base, altura);
}
```

Quedaría finalmente por implementar el constructor copia, pero te lo dejo como ejercicio.

EJERCICIO 4: Crea el constructor de copia de la clase `Rectangulo` por si se usa como parte de otras clases.

LOS SETTERS TAMBIÉN SON PELIGROSOS

Con los setters pasa lo mismo que con los constructores. Cuando reciben referencias a objetos que van para cambiar el valor de una variable global, en vez de usar directamente la referencia, deberías usar una copia de la referencia si quieres implementar composición o si mantienes oculta la implementación de la clase.

5.2.1. TIPO DE COMPOSICIONES.

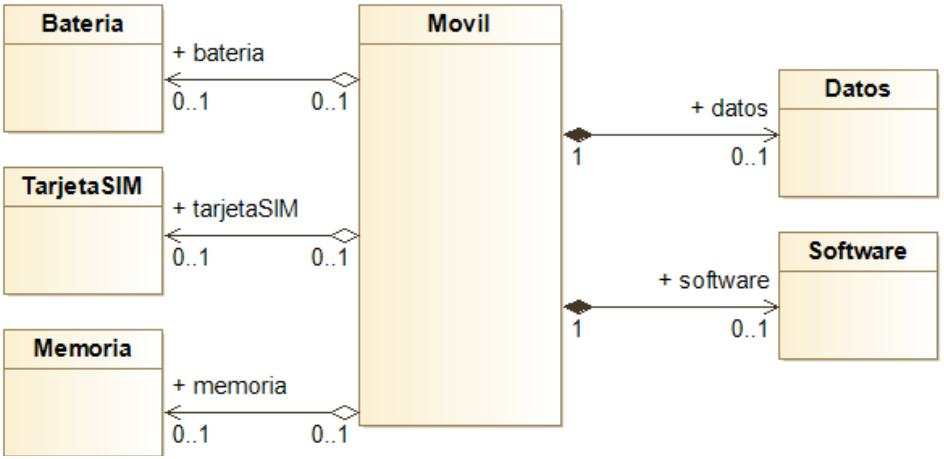
Cuando tenemos que una clase A está formada por otra clase B (A contiene a B y por tanto B es una parte A) podemos diferenciar entre **composición** y **agregación**, dos relaciones entre clases con diferencias muy sutiles.

- **Agregación:** Si el objeto de la clase A deja de existir, el objeto de la clase B puede seguir existiendo. Es una asociación débil. B no necesita a A para existir.
- **Composición: Si A desaparece (el todo), el objeto contenido B (la parte) también deja de existir.** Es una asociación fuerte. Las partes deben ser privadas.

Si quieres implementar composición, debes asegurarte que cuando se destruya el objeto, se destruyan también sus partes (eso implica no

compartirlas con el exterior) --> constructores, getters y setters no intercambian referencias directas a sus variables miembro cuando sean objetos mutables y los modificadores de acceso a private.

EJEMPLO 7: En UML la agregación se representa como un diamante hueco y la composición como un diamante sombreado. En el diagrama de clases siguiente:



Queremos expresar que el móvil está compuesto de batería, memoria y SIM pero de manera ligera (si el móvil se destruye, la tarjeta SIM, la memoria y la batería no lo necesitan para existir de manera independiente (relación de agregación). Pero el móvil también está compuesto de datos y software instalado. Si el móvil se destruye, estos elementos también (si el todo desaparece, esas partes también).

Al implementarlo, en algún lado, dentro de la clase Movil, deben estar escritas las líneas `new Datos()` y `new Software()` y no debe dejar accesibles las referencias a estos objetos desde el exterior, como mucho dejará acceder a copias. Si lo implementamos de esta manera:

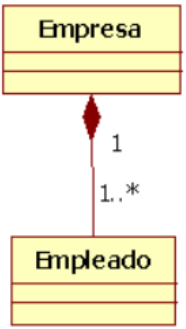
```
public class Movil {
    private Datos datos = new Datos(); // Compuestos
    private Software software;
    private Bateria b;                 // Agregados
    private Memoria ram;
    private TarjetaSIM ts;

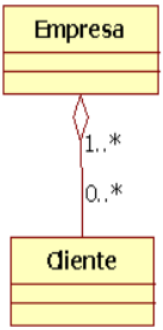
    public Movil(){ software = new Software(); ... } // Compuesto
    public void resetSoftware(){ software = new Software(); ... }
    ...
    // Si hago esto, pongo en peligro la composición
    // datos se debe crear y destruir desde dentro de la clase
    public Datos getDatos() { return datos; }
}
```

Pero en caso de hacer cosas como las marcadas de amarillo:

```
public static void Main() {
    Movil m = new Movil();
    Datos d = m.getDatos;
    m = null;          // El móvil se destruye pero...
    // Más código... los datos del móvil no, los referencia d
}
```

EJERCICIO 5: Observa estos dos gráficos UML e indica dónde se usa la composición y donde la agregación. Intenta implementarlas indicando constructores que reflejen las relaciones.





5.3 HERENCIA Y JERARQUÍAS DE CLASES.

El mecanismo que permite crear clases aprovechando otras que ya existen es conocido como **herencia**. Java implementa la herencia mediante la palabra reservada **extends**.

El concepto de herencia es algo bastante simple y sin embargo muy potente: cuando se desea definir una nueva clase y ya existen clases que, de alguna manera, implementan parte de la funcionalidad que se necesita, es posible crear una nueva clase derivada de la que ya tienes. Al hacer esto, se reutilizan todos los atributos y métodos de la clase previa o **clase base** (**clase padre** o **superclase**), sin necesidad de escribirlos de nuevo. Una **subclase** (**derivada** o **hija**) hereda todos los miembros no estáticos de su clase padre (atributos, métodos y clases internas). **Los constructores no se heredan**, aunque se pueden invocar desde la subclase.

Pero no tiene sentido utilizarla cuando en el mundo real la clase hija no es un tipo especializado de la clase padre. Algunos ejemplos:

- Un coche es un tipo de vehículo (hereda atributos como la velocidad máxima o métodos como parar y arrancar).
- Un empleado es una persona (hereda atributos como el nombre o la fecha de nacimiento).
- Un rectángulo es una figura geométrica en el plano (hereda métodos como el cálculo de la superficie o de su perímetro).
- Un cocodrilo es un reptil (hereda atributos como por ejemplo el número de dientes).

En este caso el concepto lógico que puedes usar para plantearte si el tipo de relación entre dos clases A y B es de herencia podría ser la clase hija "**es un/una**" clase padre: "la clase A **es un** tipo específico de la clase B" (especialización), o visto de otro modo: "la clase B **es un**

caso general de la clase A" (generalización).

Por este motivo, si tienes una clase llamada Paloma (esos bichos tan bonitos que representan la paz y que nos obsequian a menudo con sus excrementos) y quieres hacer otra nueva clase llamada F15 (un caza del ejército), el hecho de querer ahorrarte programar el método vuela() aprovechando que ya lo tienes programado en Paloma no sería un motivo suficiente para extender F15 de Paloma, salvo que un F15 sea un tipo de Paloma, o una Paloma sea un tipo más general de Caza.

En Java, la clase **Object** (dentro del paquete java.lang) define e implementa el comportamiento común a todas las clases (incluidas aquellas que tú escribas). **En Java cualquier clase deriva en última instancia de la clase Object**.

Todas las clases tienen una clase padre, que a su vez también posee una superclase, y así sucesivamente hasta llegar a la clase **Object**. De esta manera, se construye lo que habitualmente se conoce como una **jerarquía de clases** (como un árbol familiar) que en el caso de Java tendría a la clase Object en la raíz.

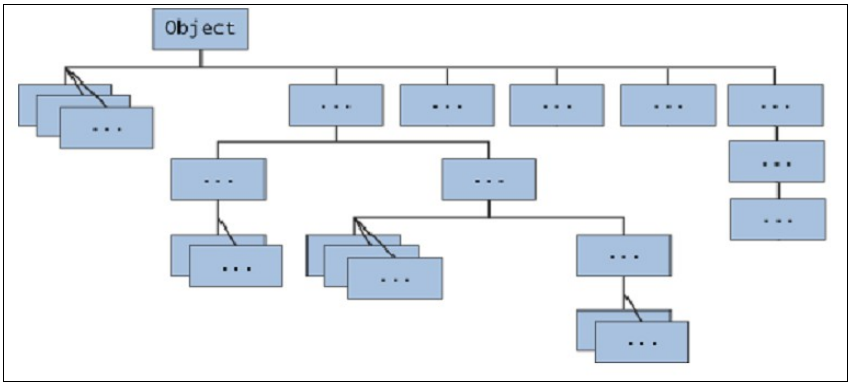


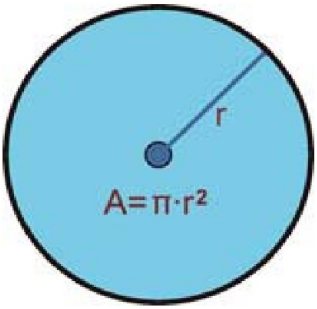
Figura 5: Jerarquía de clases en Java.

5.3.1 CUANDO USAR HERENCIA O COMPOSICIÓN.

Cuando escribas tus propias clases, debes tener claro en qué casos utilizar composición y cuándo herencia:

- **Composición:** cuando una clase está compuesta (tiene partes) de objetos de otras clases. En estos casos se incluyen objetos de esas clases, pero no necesariamente se comparten características con ellos (no se heredan características de esos objetos, sino que directamente se utilizan sus atributos y sus métodos). Esos objetos incluidos no son más que atributos miembros de la clase que se está definiendo.
- **Herencia**: cuando una clase tiene todas las características de otra (ES la otra). En estos casos, la clase derivada es una especialización (o particularización, extensión o restricción) de la clase base -> "La clase hija ES UN tipo de la clase padre" Desde otro punto de vista se diría que la clase base es una generalización de sus clases derivadas (recoge lo común a todas ellas).

Por ejemplo, imagina que trabajas con la clase Punto y decides definir una nueva clase llamada Circulo. Dado que un punto tiene como atributos sus coordenadas en el plano (x, y), decides que es buena idea aprovechar esa información e incorporarla en la clase Circulo que estás escribiendo.



Para ello utilizas la herencia, de manera que al derivar la clase Círculo desde la clase Punto, tendrás disponibles los atributos x e y. Ahora solo faltaría añadirle algunos atributos y métodos más como por

ejemplo el radio del círculo, el cálculo de su área y su perímetro, etc.

En principio parece que la idea puede funcionar, pero es posible que más adelante, si continúas construyendo una jerarquía de clases, observes que puedes llegar a conclusiones incongruentes al suponer que un círculo es una especialización de un punto (un tipo de punto). ¿Todas aquellas figuras que contengan uno o varios puntos deberían ser tipos de punto? ¿Y si tienes varios puntos? ¿Cómo accedes a ellos? ¿Un rectángulo también tiene sentido que herede de un punto? No parece muy buena idea.

Parece que en este caso habría resultado mejor establecer una relación de composición. Analízalo detenidamente: ¿cuál de estas dos situaciones te suena mejor?

1. "Un círculo es un punto (su centro)", y por tanto heredará las coordenadas x e y que tiene todo punto. Además tendrá otras características específicas como el radio o métodos como el cálculo de la longitud de su perímetro o de su área.
2. "Un círculo contiene un punto (su centro)", junto con algunos atributos más como el radio y métodos para el cálculo de su área o de la longitud de su perímetro.

Parece que en este caso la composición refleja con mayor sentido de la realidad la relación que existe entre ambas clases. Normalmente suele ser suficiente con plantearse las preguntas:

"¿A es un tipo de B?" -> Herencia.

"¿A contiene elementos de tipo B?" -> Composición

En Java la herencia se indica mediante la palabra reservada **extends**:

[modificadores] **class** ClasePadre { /* Cuerpo de la clase ... */ }

```
[modificadores] class ClaseHija extends ClasePadre { /* Cuerpo...*/ }
```

El día a día de la programación (especialmente de programadores que trabajan con objetos) consiste en crear clases y subclases: adaptar una clase que ya existe haciendo algunos cambios y añadiendo algo es una situación más común que crear clases desde cero. Para crear una nueva clase a partir de otra se utiliza esta sintaxis:

```
public class clase_nueva extends clase_existente {
    // Elementos cambiados y añadidos...
}
```

Por ejemplo, imagina que queremos implementar el juego del Blackjack usando las clases **Carta**, **Mano** y **Mazo**.

Estas 3 clases ya las tenemos pero no para el juego del BlackJack, habrá que hacer cambios a la clase **Mano** porque las reglas del juego necesitan poder calcular la puntuación de una mano en cierto momento, que se calcula sumando los valores de las cartas de la mano (el valor de una carta numérica es su valor numérico, y el valor de una sota, caballo y rey es 10 y el valor de un As puede ser tanto 1 como 11. Cuenta como 11 salvo que la suma de todas pase de 21, eso supone que el segundo, tercer y cuarto As de una mano valgan siempre 1).

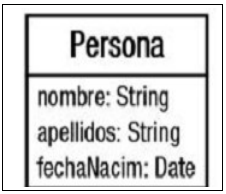
Una forma de alcanzar este objetivo es extender la clase **Mano** añadiendo un método que calcule su valor en el Blackjack. Ej:

```
public class BlackjackMano extends Mano {
   /**
   * Calcula el valor de la mano en el BlackJack.
   * Añadimos este método y reutilizamos todos los de Mano
   */
   public int getBlackjackValor() {
      int valor;     // El valor calculado
      boolean hayAs; // es true si la mano tiene un as
```

```
            int cartas;     // Nº de cartas de la mano
            valor = 0;
            hayAs = false;
            cartas = cuentaCartas();     // método de la clase Mano
            for ( int i = 0; i < cartas; i++ ) { // Sumar cartas de mano
               Carta carta;  // La i-ésima carta
               int v;        // valor de la carta
               carta = dameCarta(i);
               v = carta.getValor(); // El valor normal
               if (v > 10)
                  v = 10;   // para caballo y rey
               if (v == 1)
                  hayAs = true;   // Al menos hay un As
               valor += v;
            }
            // Ahora hay que comprobar si no pasamos de 21 y podemos
            // contar el as como 11
            if ( hayAs && valor + 10 <= 21 )
               valor = valor + 10;
            return val;
         } // fin getBlackjackValor()
      } // fin clase BlackjackMano
```

Así que hemos tenido que esforzarnos en implementar un método nuevo, para tener una clase con 8 métodos y variables (que no tenemos que volver a programar porque los heredamos), reutilizando una clase que ya existía.

Imagina ahora que tienes una clase **Persona** que contiene atributos como nombre, apellidos y fecha de nacimiento:

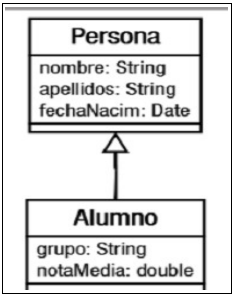



```
public class Persona {
   String nombre;
   String apellidos;
   LocalDate fechaNacim;
   …
}
```

Es posible que más adelante, necesites una clase **Alumno** que tenga

esos atributos (dado que todo alumno es una persona, pero con algunas características específicas).

En tal caso tienes la posibilidad de crear una clase **Alumno** que repita todos esos atributos o bien heredarlos de Persona:




```
public class Alumno extends Persona {
   String grupo;
   double notaMedia;
   …
}
```

A partir de ahora, un objeto de la clase Alumno contendrá los atributos grupo y notaMedia (propios de la clase Alumno), pero también nombre, apellidos y fechaNacim (propios de su clase base Persona y que por tanto ha heredado).

Una clase derivada puede ser a su vez clase padre de otra que herede de ella y así sucesivamente dando lugar a una jerarquía de clases, excepto aquellas que estén en la parte de arriba de la jerarquía (sólo serán clases padre) o en la parte de abajo (sólo serán clases hijas).

Una clase hija no tiene acceso a los miembros privados de su clase padre, tan solo a los públicos (como cualquier parte del código tendría) **y los protegidos** (a los que sólo tienen acceso las clases derivadas) **y las package** (si está almacenada en el mismo paquete). Los elementos privados de la clase base también se heredan, pero el acceso a ellos está restringido al propio funcionamiento de la superclase y sólo se podrá acceder a ellos si la superclase ha dejado algún medio indirecto para hacerlo (getter, setter...).

Todos los miembros no estáticos de la superclase (atributos y métodos), **salvo los constructores**, los hereda la subclase. Algunos de

estos podrán ser redefinidos o sobrescritos (override) y también podrán añadirse nuevos. De alguna manera podría decirse que estás "ampliando" la clase base con características adicionales o modificando algunas de ellas (proceso de especialización). Por ese motivo una clase derivada extiende (**extends**) la funcionalidad de la clase base sin volver a escribir el código.

EJEMPLO 8: Imagina que también necesitas una clase `Profesor`, que tendrá atributos como `nombre`, `apellidos`, fecha de `nacimiento`, `salario` y `especialidad`. ¿Cómo crearías esa nueva clase y qué atributos le añadirías?

Está claro que un Profesor es otra especialización de Persona, al igual que lo era Alumno, así que podrías crear otra clase derivada de Persona y así aprovechar los atributos genéricos (nombre, apellidos, fecha de nacimiento) que posee todo objeto de tipo Persona. Tan solo faltaría añadirle sus atributos específicos (salario y especialidad):

```
public class Profesor extends Persona {
   String especialidad;
   double salario;
   …
}
```

5.3.2 ACCEDER Y UTILIZAR MIEMBROS HEREDADOS.

ACCESO A MIEMBROS HEREDADOS

Sabemos que desde la subclase no es posible acceder a miembros privados de la superclase. Para acceder podrías pensar en hacerlos públicos, pero entonces estarías dando esa opción a cualquier objeto externo y es probable que tampoco sea aconsejable (aumentas acoplamiento, pierdes ocultación, ...).

Para conseguirlo se inventó el modificador **protected** (protegido) que permite el acceso desde clases heredadas, pero no desde fuera de las clases de la familia, que serían como miembros privados.

Ya comentamos los posibles modificadores de acceso que podía tener un miembro: sin modificador (acceso de paquete), público, privado o protegido. Aquí tienes de nuevo el resumen:

Cuadro de niveles accesibilidad a los atributos de una clase				
	Misma clase	**Subclase**	**Mismo paquete**	**Otro paquete**
Sin modificador (paquete)	X		X	X
public	X	X	X	X
private	X			
protected	X	X	X	

Figura 6: resumen de modificadores de control de acceso.

Si en el ejemplo anterior de la clase Persona se hubieran definido sus atributos como private:

```
public class Persona {
   private String nombre;
   private String apellidos;
   …
}
```

Al definir la clase Alumno, como heredera de Persona, no habrías tenido acceso a esos atributos, pudiendo ocasionar un grave problema de operatividad al intentar manipular esa información. Por tanto, en estos casos lo más recomendable habría sido declarar los atributos como **protected** o bien sin modificador (para que también tengan acceso a ellos otras clases del mismo paquete, si se considera oportuno). Otra opción es dejar private e implementar getters y setters:

```
public class Persona {
   protected String nombre;
   protected String apellidos;
   …
}
```

Si una clase va a tener descendientes, sólo en aquellos casos en los que se quiere/necesita que un miembro de una clase no pueda ser accesible desde una clase derivada debería utilizarse el modificador **private**. En el resto de casos es recomendable utilizar **protected**.

EJEMPLO 9: Reescribe las clases `Alumno` y `Profesor` utilizando el modificador `protected` para sus atributos del mismo modo que se ha hecho para su superclase `Persona`.

1. Clase Alumno. Se trata simplemente de añadir el modificador de acceso protected a los nuevos atributos que añade la clase.

```
public class Alumno extends Persona {
   protected String grupo;
   protected double notaMedia;
   …
}
```

2. Clase Profesor. Exactamente igual que en la clase Alumno.

```
public class Profesor extends Persona {
   protected String especialidad;
   protected double salario;
   …
}
```

USAR ATRIBUTOS HEREDADOS

Los atributos heredados por una clase son, a efectos prácticos, iguales que aquellos que se han definido explícitamente en la nueva clase derivada.

En el ejemplo anterior, la clase Persona disponía de tres atributos y la

clase Alumno, que heredaba de ella, añadía dos atributos más. Desde un punto de vista funcional podrías considerar que la clase Alumno tiene cinco atributos: tres por ser Persona (nombre, apellidos, fecha de nacimiento) y otros dos más por ser Alumno (grupo y nota media).

EJEMPLO 10: Dadas las clases `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos `get()` y `set()` en las clases `Alumno` y `Profesor` para trabajar con sus 5 atributos (3 heredados más 2 específicos).

POSIBLE SOLUCIÓN

1. Clase Alumno. Se trata de heredar de la clase Persona y por tanto utilizar con normalidad sus atributos heredados como si pertenecieran a la propia clase (de hecho se puede considerar que le pertenecen, dado que los ha heredado).

```
public class Alumno extends Persona {
   protected String grupo;
   protected double notaMedia;
   // Método getNombre
   public String getNombre(){ return nombre;  }
   // Método getApellidos
   public String getApellidos(){  return apellidos; }
   // Método getFechaNacim
   public GregorianCalendar getFechaNacim(){
      return this.fechaNacim;
   }
   // Método getGrupo
   public String getGrupo(){ return grupo; }
   // Método getNotaMedia
   public double getNotaMedia(){ return notaMedia; }
   // Método setNombre
   public void setNombre(String nombre){
      this.nombre= nombre;
   }
   // Método setApellidos
   public void setApellidos (String apellidos){
      this.apellidos= apellidos;
```

```
    }
    // Método setFechaNacim
    public void setFechaNacim (GregorianCalendar fechaNacim){
       this.fechaNacim= fechaNacim;
    }
    // Método setGrupo
    public void setGrupo (String grupo){
       this.grupo= grupo;
    }
    // Método setNotaMedia
    public void setNotaMedia(double notaMedia){
       this.notaMedia= notaMedia;
    }
}
```

Si te fijas, puedes utilizar sin problema la referencia **this** al propio objeto con los atributos heredados, pues pertenecen a la clase: **this**.nombre, **this**.apellidos, etc.

2. Clase Profesor. Seguimos exactamente el mismo procedimiento que con la clase Alumno.

```
public class Profesor extends Profesor {
String especialidad;
double salario;
// Método getNombre
public String getNombre(){ return nombre; }
// Método getApellidos
public String getApellidos(){ return apellidos; }
// Método getFechaNacim
public GregorianCalendar getFechaNacim (){
   return this.fechaNacim;
}
// Método getEspecialidad
public String getEspecialidad(){
   return especialidad;
}
// Método getSalario
public double getSalario(){ return salario; }
// Método setNombre
public void setNombre(String nombre){
   this.nombre= nombre;
```

```
    }
    // Método setApellidos
    public void setApellidos(String apellidos){
       this.apellidos= apellidos;
    }
    // Método setFechaNacim
    public void setFechaNacim (GregorianCalendar fechaNacim){
       this.fechaNacim= fechaNacim;
    }
    // Método setSalario
    public void setSalario (double salario){
    this.salario= salario;
    }
    // Método setESpecialidad
    public void setESpecialidad (String especialidad){
       this.especialidad= especialidad;
    }
}
```

Una conclusión que puedes extraer de este código es que has tenido que escribir los métodos get y set para los tres atributos heredados, pero ¿no habría sido posible definir esos seis métodos en la clase base y así estas dos clases derivadas hubieran también heredado esos métodos? La respuesta es afirmativa y de hecho es como lo vas a hacer a partir de ahora.

De esa manera te habrías evitado tener que escribir seis métodos en la clase Alumno y otros seis en la clase Profesor. Así que recuerda: se heredan tanto los atributos como los métodos.

Aquí tienes un ejemplo de cómo podrías haber definido la clase Persona para que luego se hubieran podido heredar de ella sus métodos (y no sólo sus atributos).

EJEMPLO 11: En la clase base se implementan las cosas comunes a todas las subclases. Eso ahorra esfuerzo y reduce el código.

```
public class Persona {
```

```
    protected String nombre;
    protected String apellidos;
    protected LocalDate fechaNacim;
    // Método getNombre
    public String getNombre(){ return nombre; }
    // Método getApellidos
    public String getApellidos(){ return apellidos; }
    // Método getFechaNacim
    public LocalDate getFechaNacim(){
       return this.fechaNacim;
    }
    // Método setNombre
    public void setNombre (String nombre){ this.nombre= nombre; }
    // Método setApellidos
    public void setApellidos(String apellidos){
       this.apellidos= apellidos;
    }
    // Método setFechaNacim
    public void setFechaNacim(LocalDate fechaNacim){
       this.fechaNacim= fechaNacim;
    }
}
```

USAR MÉTODOS HEREDADOS

Del mismo modo que se heredan los atributos, también se heredan los métodos, convirtiéndose a partir de ese momento en otros métodos más de la clase derivada junto a los que se le añadan.

En el ejemplo de la clase Persona, al disponer de métodos get y set para cada uno de sus tres atributos (nombre, apellidos, fechaNacim), tendrías seis métodos que podrían ser heredados por sus clases derivadas. Podrías decir entonces que la clase Alumno, derivada de Persona, tiene diez métodos:

- Seis por ser Persona (getNombre, getApellidos, getFechaNacim, setNombre, setApellidos, setFechaNacim).
- Otros cuatro más por ser Alumno (getGrupo, setGrupo, getNotaMedia, setNotaMedia).

Sin embargo, sólo tendrías que definir esos cuatro últimos (los específicos) pues los genéricos ya los has heredado de la superclase.

EJEMPLO 12: Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, implementa métodos get y set en la clase `Persona` para trabajar con sus tres atributos y en las clases `Alumno` y `Profesor` para manipular sus cinco atributos (tres heredados más dos específicos), teniendo en cuenta que los métodos que ya hayas definido para Persona van a ser heredados en `Alumno` y en `Profesor`.

POSIBLE SOLUCIÓN

1. Clase Persona.

```
public class Persona {
   protected String nombre;
   protected String apellidos;
   protected LocalDate fechaNacim;
   // Método getNombre
   public String getNombre(){ return nombre; }
   // Método getApellidos
   public String getApellidos(){ return apellidos; }
   // Método getFechaNacim
   public Localdate getFechaNacim(){ return this.fechaNacim; }
   // Método setNombre
   public void setNombre (String nombre){ this.nombre= nombre; }
   // Método setApellidos
   public void setApellidos(String apellidos){
      this.apellidos= apellidos;
   }
   // Método setFechaNacim
   public void setFechaNacim(LocalDate fechaNacim){
      this.fechaNacim= fechaNacim;
   }
}
```

2. Clase Alumno. Al heredar de la clase Persona tan solo es necesario escribir métodos para los nuevos atributos (métodos especializados de acceso a los atributos especializados), pues los métodos genéricos (de

acceso a los atributos genéricos) ya forman parte de la clase al haberlos heredado.

```
public class Alumno extends Persona {
   protected String grupo;
   protected double notaMedia;
   // Método getGrupo
   public String getGrupo(){ return grupo; }
   // Método getNotaMedia
   public double getNotaMedia(){ return notaMedia; }
   // Método setGrupo
   public void setGrupo(String grupo){ this.grupo= grupo; }
   // Método setNotaMedia
   public void setNotaMedia(double notaMedia){
      this.notaMedia= notaMedia;
   }
}
```

Aquí tienes una demostración práctica de cómo la herencia permite una reutilización eficiente del código, evitando tener que repetir atributos y métodos. Sólo has tenido que escribir cuatro métodos en lugar de diez.

3. Clase Profesor. Seguimos exactamente el mismo procedimiento que con la clase Alumno.

```
public class Profesor extends Persona {
   String especialidad;
   double salario;
   // Método getEspecialidad
   public String getEspecialidad(){ return especialidad; }
   // Método getSalario
   public double getSalario(){ return salario; }
   // Método setSalario
   public void setSalario(double salario) {
      this.salario= salario;
   }
   // Método setEspecialidad
   public void setESpecialidad(String especialidad){
      this.especialidad= especialidad;
```

```
    }
}
```

5.3.3 REDEFINIR Y AMPLIAR MÉTODOS.

REDEFINIR MÉTODOS HEREDADOS

Una clase puede redefinir algunos de los métodos que ha heredado de su clase base. En tal caso, el nuevo método (especializado) sustituye al heredado. Este procedimiento es un tipo de sobrecarga que también es conocido como **sobrescritura de métodos**.

En cualquier caso, aunque un método sea sobrescrito o redefinido en la clase hija, aún es posible acceder a él a través de la referencia **super**, aunque sólo se podrá acceder a métodos de la clase padre y no a métodos de clases superiores en la jerarquía de herencia.

Los métodos redefinidos pueden ampliar su accesibilidad con respecto a la que ofrezca el método original de la superclase, pero nunca restringirla. Por ejemplo, si un método es declarado como protected o de paquete en la clase base, podría ser redefinido como public en una clase derivada, o mantener su acceso, pero no dejarlo en private.

Los métodos estáticos o de clase no pueden ser sobrescritos. Los originales de la clase base permanecen inalterables a través de toda la jerarquía de herencia.

En el ejemplo de la clase Alumno, podrían redefinirse algunos de los métodos heredados. Por ejemplo, imagina que el método getApellidos devuelva la cadena "Alumno:" junto con los apellidos del alumno. En tal caso habría que reescribir ese método para realizar esa modificación:

```
public String getApellidos() {
    return "Alumno: " + apellidos;
}
```

Cuando sobrescribas un método heredado en Java puedes incluir la anotación **@Override**. En cualquier caso, **no es obligatorio indicar @Override**, pero puede resultar de ayuda a la hora de localizar este tipo de errores (crees que has sobrescrito un método heredado y al confundirte en una letra estás realmente creando un nuevo método diferente). En el caso del ejemplo anterior quedaría:

```
@Override
public String getApellidos() { ...
```

EJEMPLO 13: Dadas las clases `Persona`, `Alumno` y `Profesor` que has utilizado anteriormente, redefine el método `getNombre()` para que devuelva la cadena "Alumno: ", junto con el nombre del alumno, si se trata de un objeto de la clase `Alumno` o bien "Profesor ", junto con el nombre del profesor, si se trata de un objeto de la clase `Profesor`.

1. Clase Alumno. Al heredar de la clase Persona tan solo es necesario escribir métodos para los nuevos atributos (métodos especializados de acceso a los atributos especializados), pues los métodos genéricos (de acceso a los atributos genéricos) ya forman parte de la clase al haberlos heredado. Esos son los métodos que se implementaron en el ejercicio anterior (getGrupo, setGrupo, etc.).

Ahora bien, hay que escribir otro método más, pues tienes que redefinir el método getNombre para que tenga un comportamiento un poco diferente al getNombre que se hereda de la clase base Persona:

```
// Método getNombre
@Override
public String getNombre() { return "Alumno: " + this.nombre; }
```

En este caso podría decirse que se "renuncia" al método heredado para redefinirlo con un comportamiento más especializado y acorde con la

clase derivada.

2. Clase Profesor. Seguimos exactamente el mismo procedimiento que con la clase Alumno (redefinición del método getNombre).

```
// Método getNombre
@Override
public String getNombre(){ return "Profesor: " + this.nombre; }
```

MODIFICAR MÉTODO HEREDADO

Hasta ahora, has visto que para redefinir o sustituir un método de una superclase es suficiente con crear otro método en la subclase que tenga la misma firma que el método que se desea sobrescribir. Pero, en otras ocasiones, puede que no necesites sustituir completamente el comportamiento del método de la superclase, sino simplemente ampliarlo.

Para poder hacer esto, necesitas poder preservar el comportamiento antiguo (el de la superclase) y añadir el nuevo (el de la subclase). Para ello, puedes invocar desde el método "ampliador" de la clase derivada al método "ampliado" de la clase superior (teniendo ambos métodos el mismo nombre). ¿Cómo se puede conseguir eso? Puedes hacerlo mediante el uso de la referencia **super**.

La palabra reservada **super** es una referencia al objeto de la clase padre de la clase en la que te encuentres en ese momento (es algo similar a **this**, que representaba una referencia al objeto de la clase actual). De esta manera, podrías invocar a cualquier método de la superclase (si se tiene acceso a él).

Por ejemplo, imagina que la clase Persona dispone de un método que permite mostrar el contenido de algunos datos personales de los objetos de este tipo (nombre, apellidos, etc.). Por otro lado, la clase

Alumno también necesita un método similar, pero que muestre también su información especializada (grupo, nota media, etc.). ¿Cómo podrías aprovechar el método de la superclase para no tener que volver a escribir sus sentencias en la subclase? Puede hacerse sencillamente:

```
public void mostrar() {
   super.mostrar(); // Llama al método "mostrar" de la superclase
   // A continuación MUESTRA la información "especializada"
   System.out.printf ("Grupo: %s\n", this.grupo);
   System.out.printf ("Nota media: %5.2f\n", this.notaMedia);
}
```

Este tipo de ampliaciones de métodos resultan especialmente útiles por ejemplo en el caso de los constructores, donde se podría ir llamando a los constructores de cada superclase encadenadamente hasta el constructor de la clase en la cúspide de la jerarquía (el constructor de la clase Object).

EJEMPLO 14: Dadas las clases `Persona`, `Alumno` y `Profesor`, define un método `mostrar()` para la clase `Persona`, que muestre el contenido de los atributos (datos personales) de un objeto de la clase `Persona`. A continuación, define sendos métodos `mostrar()` especializados para las clases `Alumno` y `Profesor` que "amplíen" la funcionalidad del método mostrar original de la clase Persona.

1. Método mostrar de la clase Persona.

```
public void mostrar() {
 SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
 String stringfecha= formatoFecha.format(this.fechaNacim.getTime());
 System.out.printf("Nombre: %s\n", this.nombre);
 System.out.printf("Apellidos: %s\n", this.apellidos);
 System.out.printf("Fecha de nacimiento: %s\n", stringfecha);
}
```

2. Método mostrar de la clase Profesor. Llamamos al método mostrar

de su clase padre (Persona) y luego añadimos la funcionalidad específica para la subclase Profesor:

```
public void mostrar() {
   super.mostrar(); // Llama al método "mostrar" de la superclase
   System.out.printf ("Especialidad: %s\n", this.especialidad);
   System.out.printf ("Salario: %7.2f euros\n", this.salario);
}
```

3. Método mostrar de la clase Alumno. Llamamos al método mostrar de su clase padre (Persona) y luego añadimos la funcionalidad específica para la subclase Alumno:

```
public void mostrar() {
   super.mostrar();
   System.out.printf("Grupo: %s\n", this.grupo);
   System.out.printf("Nota media: %5.2f\n", this.notaMedia);
}
```

USAR CONSTRUCTORES HEREDADOS

Recuerda que cuando vimos los constructores se dijo que un constructor de una clase puede llamar a otro constructor de la misma clase, a través de la referencia **this**. En estos casos, la utilización de **this** sólo podía hacerse en la primera línea de código del constructor.

Como ya has visto, un constructor de una clase derivada puede hacer algo parecido para llamar al constructor de su clase base mediante el uso de la palabra **super**. De esta manera, el constructor de una clase derivada puede llamar primero al constructor de su superclase para que inicialice los atributos heredados y posteriormente se inicializarán los atributos específicos de la clase: los no heredados.

Nuevamente, esta llamada también debe ser la primera sentencia de un constructor (con la única excepción de que exista una llamada a otro constructor de la clase mediante **this**).

Si no se incluye una llamada a **super(algo)** dentro del constructor, el compilador incluye automáticamente una llamada al constructor por defecto de clase base (llamada a super()). Esto da lugar a una llamada en cadena de constructores de superclase hasta llegar a la clase más alta de la jerarquía (que en Java es la clase Object).

En el caso del constructor por defecto (el que crea el compilador si el programador no ha escrito ninguno), el compilador añade lo primero de todo, antes de la inicialización de los atributos a sus valores por defecto, una llamada al constructor de la clase base mediante la referencia **super**.

A la hora de destruir un objeto (método **finalize**) es importante llamar a los finalizadores en el orden inverso a como fueron llamados los constructores (primero se liberan los recursos de la clase derivada y después los de la clase base mediante la llamada **super.finalize()**).

Si la clase Persona tuviera un constructor de este tipo:

```
public Persona(String nombre, String apellidos, LocalDate
fechaNacim) {
   this.nombe= nombre;
   this.apellidos= apellidos;
   this.fechaNacim= new LocalDate(fechaNacim);
}
```

Podrías llamarlo desde un constructor de una clase derivada (por ejemplo Alumno) de la siguiente forma:

```
public Alumno(String nombre, String apellidos, LocalDate
fechaNacim, String grupo, double notaMedia) {
   super(nombre, apellidos, fechaNacim);
   this.grupo = grupo;
   this.notaMedia = notaMedia;
}
```

En realidad se trata de otro recurso más para optimizar la

reutilización de código, en este caso el del constructor, que aunque no es heredado, sí puedes invocarlo para no tener que rescribirlo.

Observa que el constructor de la subclase recibe como argumentos todos sus atributos, y pasa a la superclase solo los que necesita el constructor de la superclse (nombre, apellidos y fecha de nacimiento).

EJERCICIO 6: Escribe un constructor para la clase `Profesor` que realice una llamada al constructor de su clase base para inicializar sus atributos heredados. Los atributos específicos (no heredados) sí deberán ser inicializados en el propio constructor de la clase `Profesor`.

Si antes de llamar a `super()` debes calcular en el constructor algún valor para pasarlo como argumento, al tener que aparecer `super()` como la primera sentencia (salvo `this()`) podrías crear un método o una variable global estáticos y privados o un bloque de código que hiciese ese trabajo y llamarlo desde los parámetros.

EJERCICIO 7: Modifica el constructor del ejercicio 6 para que:

a) la fecha de nacimiento se obtenga antes de usar super() desde un bloque de código inicializador y añade un mensaje de texto tanto en el bloque como en el constructor del tipo "estoy en <lugar>" para saber que es lo que se ejecuta en primer lugar.

b) Haz lo mismo que en el apartado a) pero usando un método estático que se llama desde la llamada a super().

5.4. ANIDAMIENTO DE CLASES.

Es posible definir una clase dentro de otra clase (**clases internas**). En ese caso se dice que la clase está anidada. Ejemplo:

```
class claseContenedora {
```

```
        // Cuerpo de la clase contenedora…
        class claseInterna { /* Cuerpo de la clase interna… */ }
        // Más cosas de la clase contenedora…
    }
```

Hay varios beneficios de hacerlo:

- En primer lugar, dejas las clases relacionadas juntas en una unidad lógica, ocultándolas al exterior donde quizás no sean necesarias.
- En segundo lugar la clase contenedora accede a todos los elementos de sus clases anidadas y viceversa.
- En tercer lugar, simplificas el código (a veces).

Por ejemplo se usan mucho con interfaces gráficas como AWT/Swing/.

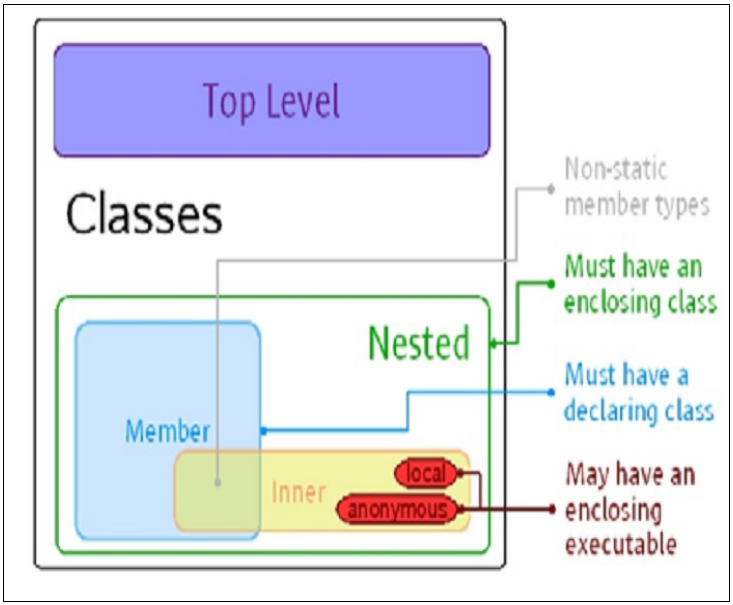


Figura 7: Tipos de clases internas anidadas.

Una clase es un bloque de alto nivel de un programa, representando una

idea compleja: sus datos asociados y su comportamiento. Sin embargo es engorroso tener que escribir clases muy sencillas, es más el trabajo de hacerlas que el beneficio que aportan (un fichero para ellas, etc.).

Esto en Java se puede solucionar con la anidación de unas clases dentro de otra clase. La pequeña clase trivial puede pertenecer a una clase más compleja. Esto se utiliza mucho sobre todo cuando la clase sencilla solamente sirve para facilitar el trabajo de la compleja.

En Java, **una clase anidada es cualquier clase cuya definición está dentro de otra clase** (de hecho, una clase puede estar anidada hasta dentro de un método, que a su vez, está dentro de una clase). Las clases anidadas pueden tener nombre o ser anónimas. Y las que tienen nombre pueden ser estáticas o no estáticas.

Las clases internas se utilizan en algunos casos para:

- Agrupar clases que sólo tiene sentido que existan en el entorno de la clase en la que han sido definidas, de manera que se oculta su existencia al resto del código.
- Incrementar el nivel de encapsulación y ocultamiento.
- Proporcionar un código fuente más legible y fácil de mantener (el código de las clases internas y anidadas está más cerca de donde se utiliza).

Se pueden distinguir varios tipos de clases internas en Java:

- Clases **internas estáticas** (conocidas como **clases anidadas**), declaradas con el modificador **static**. En el caso de que la interna sea una interface, aunque no uses la palabra static, lo es de forma implícita.
- **Clases internas miembro** (conocidas como **clases internas)**, declaradas a nivel de la clase contenedora como no estática. Las

clases internas (no estáticas) tienen una relación especial porque pueden acceder mutuamente a sus miembros y no les afectan los modificadores de acceso. Una limitación de las clases internas es que **no pueden tener miembros estáticos**.

- **Clases internas locales**, se declaran en el interior de un bloque de código (normalmente dentro de un método).
- **Clases anónimas**, similares a las internas locales, pero sin nombre (sólo existirá un objeto de ellas y al no tener nombre, no tendrán constructores). Se suelen usar en la gestión de eventos en los interfaces gráficos.

CLASE ANIDADA (INTERNA Y STATIC)

Las clases anidadas, como miembros de una clase que son (miembros de claseExterna), pueden ser declaradas con los modificadores **public**, **protected**, **private** o de paquete, como el resto de miembros.

La definición es como la de cualquier otra clase, salvo porque se produce dentro de otra clase y tiene el modificador **static** como parte de su declaración.

Es un elemento estático de la clase que la contiene. Se usa para crear objetos y si se utiliza desde fuera de la clase donde se define, su llamada debe prefijarse con la clase donde está definida (clase_contenedora.clase_anidada).

EJEMPLO 16: imagina una clase llamada `ModeloAlambre` que representa un conjunto de líneas en el espacio 3D (se usan para representar figuras 3D). Imagina que esta clase tiene una clase estática anidada llamada `Linea` que representa una línea. Fuera de la clase `ModeloAlambre`, esa clase se referencia como `ModeloAlambre.Linea`.

```
public class ModeloAlambre {
```

```
        // Otros miembros de la clase...
        static public class Linea{ // Linea de (x1,y1,z1) hasta (x2,y2,z2)
           double x1, y1, z1;
           double x2, y2, z2;
        } // fin clase Linea
        // Otros miembros de la clase ModeloAlambre...
    } // fin clase ModeloAlambre
```

Así que dentro de la clase podemos crear objetos Linea con la sentencia **new Linea()** y desde fuera con **new ModeloAlambre.Linea()**.

- **Una clase estática anidada puede acceder a todos los miembros estáticos de la clase contenedora (incluso a los privados)**.
- **De igual forma, la clase contenedora accede a todos los miembros static de la clase anidada (incluso los privados)**.
- **Si la propia clase anidada se declara como private, solo podrá utilizarse dentro de la clase contenedora, no desde el exterior**.

Cuando compilas el fichero fuente anterior, se generan dos ficheros .class. El nombre del fichero de la clase Linea será `ModeloAlambre$Linea.class`.

EJERCICIO 8: Necesitamos utilizar objetos de tipo `Punto2D` que tienen dos variables miembro `x` e `y`. Cada objeto define un punto en el plano bidimensional. Cuando tengamos que utilizar una gran cantidad de estos objetos queremos ahorrar memoria y ganar velocidad aunque se sacrifique precisión (usaremos coordenadas de tipo `float`). En otras ocasiones sin embargo necesitaremos obtener cálculos precisos aunque nos cueste más tiempo y memoria así que usaremos coordenadas de tipo `double`. ¿Cómo tener una sola clase que use uno u otro tipo de variables según necesitemos, pero no las dos al mismo tiempo? Es decir un objeto o tendrá coordenadas float o double pero no ambas.

Una solución sería utilizar sentencias como esta para crear puntos con poco gasto de memoria:

```
Punto2D p = new Punto2D.Float(1.0f, 2.0f, 3.0f);
```

Y esta para cálculos con más precisión:

```
Punto2D p = new Punto2D.Double(1.0, 2.0, 3.0);
```

Debes hacer que la clase `Punto2D` sea abstracta (no se pueda instanciar) y que defina por ejemplo la operación abstracta `public Punto2D suma(Punto2D a)`. Crea `Float` y `Double` como dos clases estáticas e internas suyas que además la extiendan. Podrás conseguir definir en una sola clase ambos comportamientos. Implementa constructor por defecto, constructor completo, getters, setters y sobreescribe `toString()`. Sobreescribe la operación `suma()`. Cuando lo hagas este código debería funcionarte:

```
public static void main(String[] args) {
        System.out.println("Queremos poco gasto (RAM y CPU): ");
        Punto2D p1 = new Punto2D.Float(1.0f, 2.f);
        Punto2D p2 = new Punto2D.Float(1/3.0f,  2.0f);
        System.out.println("  p1= " + p1 + " y p2= " + p2);
        System.out.println("  p1.suma(p2) = " + p1.suma(p2) );
        System.out.println("Queremos Precisión: ");
        Punto2D p3 = new Punto2D.Double(1.0, 2.0);
        Punto2D p4 = new Punto2D.Double(1/3.0,  2.0);
        System.out.println("  p3= " + p3 + " y p4= " + p4);
        System.out.println("  p3.suma(p4)= " + p3.suma(p4) );
        System.out.println("Podemos mezclar: ");
        System.out.println("  p1.suma(p3)= " + p1.suma(p3) );
        System.out.println("  p3.suma(p1)= " + p3.suma(p1) );
    }
```

CLASES INTERNAS (Inner Classes, NO STATIC)

Las clases anidadas no estáticas se denominan **clases internas**. Se

diferencian de las estáticas en que la definición está **asociada a los objetos, no a la clase**.

Cada objeto tendrá su propia definición de clase. La clase interna tendrá acceso a todos los métodos y variables de instancia del objeto, aunque sean privados. Al contrario es cierto también, pero debes instanciar un objeto de la interna antes de acceder (al contrario no es necesario porque el objeto externo ya existe cuando se ejecuta el interno).

La regla para decidir cuando hacer una clase anidada estática o no es sencilla: **si necesita acceder a las variables de instancia/métodos de los objetos, se hace interna, en otro caso anidada estática**.

En la mayoría de los casos, una clase interna se usa solamente dentro de la clase donde se define. Desde dentro de la clase contenedora puedes crear variables y declarar objetos usando el nombre de la clase. Pero si también necesitas usarla desde fuera, por ejemplo para definir una referencia o crear uno de sus objetos, debes prefijar con el objeto: objeto.clase_anidada (cuando trabajas dentro de la clase se usa **this** de forma implícita).

EJEMPLO 17: Definir una clase interna (no estática).

```
public class Poker {           // Juego de poker
    class JugadorPoker {       // Un jugador de Póker
        //...
    } // fin clase Jugador
    private Mazo m;      // El mazo de cartas
    private int bote;    // cantidaad de dinero en juego
    //...
} // fin clase Poker
```

Si juego es una variable de tipo Poker (`Poker juego = new Poker();`), juego tiene su propia copia de la definición de la clase `JugadorPoker`.

Dentro de un método de instancia (del objeto) se puede crear un nuevo jugador haciendo la llamada `new JugadorPoker()` y desde fuera del objeto se haría `juego.new JugadorPoker()`.

Las clases internas tienen acceso a otras variables miembro de los objetos de la clase dentro de la que está definida aunque sean privados (se trata en cierto modo de un miembro más de la clase), mientras que las anidadas (estáticas) solo a las variables estáticas de la clase.

EJEMPLO 18: Para implementar el centro de un círculo, asociamos cada `Circulo` con su punto central definiendo la clase `Punto` como interna.

```
// Circulo.java
public class Circulo {

    class Punto {
        private int xPos, yPos;
        public Punto(int x, int y) { xPos = x;  yPos = y; }
        public String toString() { return "(" + xPos + "," + yPos + ")"; }
    }

    private Punto centro;
    private int radio;
    public Circulo(int x, int y, int r) {
         centro = this.new Punto(x, y);
         radio = r;
    }

    public String toString() {
        return "centro= " + centro + " y radio= " + radio;
    }

    public static void main(String []s) {
        System.out.println( new Circulo(10,10,20) );
    }
    // otros métodos...
}
```

Presta atención a como se instancia el objeto centro, que está declarada como private. Una limitación de las clases internas es que no

pueden tener miembros estáticos:

```
class Externa {
    class Interna { static int i = 10; }
}
Outer.java:3: inner classes cannot have static declarations
 static int i = 10;
```

EJERCICIO 9: Indica cuales de las siguientes líneas hay que comentar para que el siguiente código funcione:

```
1  public class Externa {
2      private int xExterna = 1;
3
4      public void test() {
5          Interna i = new Interna();
6          i.muestra();
7      }
8
9      public class Interna{
10         private int yInterna = 2;
11         public void muestra(){
12             System.out.println("xExterna: " + xExterna);
13         }
14     }
15
16     public void muestra() {
17         System.out.println("yInterna: " + yInterna);
18     }
19
20     public void muestra1() {
21         Interna i = new Interna();
22         System.out.println("yInterna: " + i.yInterna);
23     }
24
25     public static void main(String[] args) {
26         Externa e = new Externa();
27         e.muestra();
28         e.muestra1();
29         e.test();
30     }
31 }
```

CLASES INNER LOCALES

Son clases que **se definen dentro de un bloque de código**. No son

miembros de una clase externa, sino locales al código donde se definen. Por tanto no son accesibles desde el exterior de ese código.

EJEMPLO 19: Definir una clase inner local con nombre.

```
class AlgunaClase {
    void algunMetodo() {
        class Local { }  // inner local al definirse dentro del método
    }
}
```

Igual que las variables locales, no pueden declararse como estáticas. Además, como una interface no puede definir código, no puedes tener clases internas locales dentro de ellas. Y lo contrario también es cierto: no puedes definir interfaces locales dentro de un método.

EJEMPLO 20: tenemos esta clase abstracta:

```
abstract class Figura {

    public static class Color {
        int r, g, b;  // rojo, verde, azul

        public Color() { this(0, 0, 0); }
        public Color(int rp, int gp, int bp) { r = rp; g = gp;  b = bp; }
        public String toString(){ return "RGB=" + r + "," + g + "," + b; }
        // otros miembros...
    }
    // otros miembros...
}
```

Imagina que en una clase `Estado` que utiliza la clase `Figura`, necesitas que el método `toString()` de `Figura` muestre una representación descriptiva de esta forma: `"Color actual RGB=0,0,0"`. Debes definir un método llamado `getEstado()` en la clase `Estado` y creamos una clase derivada de `Figura.Color`:

```
class Estado {
    static Figura.Color getEstado( final Figura.Color c ) {
        // La clase interna local definida dentro del método
        class ColorDescriptivo extends Figura.Color {
              public String toString() { return "Color actual " + c; }
```

```
            }
            return new ColorDescriptivo();
        }

        public static void main(String[] args) {
            Figura.Color cD = Estado.getEstado(new Figura.Color(0, 0, 0));
            System.out.println(cD);
        }
    }
```

El método `getEstado()` acepta de parámetro una `Figura` y devuelve un objeto `Figura.Color`. Dentro del método se define la clase `ColorDescriptivo` que se deriva de la clase `Figura.Color` y define un único método que sobreescribe al método `toString()` de `Figura`. Después de definir la clase, crea un objeto de la clase y la devuelve.

Aunque no declares el parámetro como `final`, si intentas modificarlo en el código, se produce un error de compilación cuando la clase interna la utilice.

EJERCICIO 10: Completa el código para que genere la siguiente salida utilizando una clase interna local en el cuerpo del siguiente bucle:

```
2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
```

```
public class Externa {
    private int xExterna = 2;

    public void test() {
        for(int i= 0; i < 10; i++) {
            class Interna {
                // Completa la clase interna...
            }
            Interna interna = new Interna();
            interna.muestra(i);
        }
    }

    public static void main(String[] args) {
        Externa e = new Externa();
        e.test();
    }
}
```

CLASES INNER ANÓNIMAS

Las clases anidadas anónimas se usan con mucha frecuencia. Algunas veces resulta útil crear clases que no tengan ni nombre, en una sola línea de código:

```
new superclase_o_interface(parámetros){ métodos_y_variables };
```

Este constructor define una nueva clase sin darle nombre y crea un objeto suyo al mismo tiempo. La intención es crear un objeto personalizado de la superclase en el punto donde se ejecute. **Si se usa una interfaz en vez de una clase, los parámetros deben estar vacíos, si se usa una clase, los parámetros sirven para pasarlos al constructor.**

> **IMPORTANTE:** Para diferenciar cuando creas una clase anónima y la instancias de cuando simplemente creas una instancia de una clase, observa que en el primer caso además de llamar al constructor proporcionas un cuerpo donde la modificas: `new Clase(..){..};` y `new Interface(..){..};` mientras que al crear solo un objeto, no indicas el cuerpo (las llaves): `new Clase(...);`

EJEMPLO 21: Considera que la interfaz `Drawable` obliga a implementar el método `draw()`. Imagina que queremos un objetoque implemente `Drawable` y que dibuje un cuadrado relleno de rojo de 100 píxels. En vez de definir una nueva clase separada en su fichero y luego usar esa clase para construir el objeto, podemos hacerlo con una clase anónima:

```
Drawable cuadradoRojo = new Drawable() {
                    void draw(Graphics g) { // Sobreescribo draw()
                        g.setColor(Color.RED);
                        g.fillRect(10,10,100,100);
                    }
                        }; // final de la declaración
```

Las clases anónimas se usan frecuentemente para pasar parámetros actuales. Por ejemplo, el método siguiente dibuja un Drawable en dos contextos gráficos diferentes:

```
void dibuja2Veces( Graphics g1, Graphics g2, Drawable figura ) {
   figura.draw(g1);
   figura.draw(g2);
}
```

Cuando se haga una llamada al método, el tercer parámetro se puede crear en ese mismo momento usando una clase interna anónima. La posibilidad tradicional es crearla independiente con el esfuerzo y engorro que eso supone.

EJEMPLO 22: crear objetos para pasarlos como parámetro en el momento de realizar la llamada a un método ahorra esfuerzo.

```
dibuja2Veces( primerG, segundoG,
              new Drawable() {
                    void draw(Graphics g) {
                        g.drawOval(10,10,100,100);
                    } }
            );
```

Cuando se compilan los fuentes, cada clase anónima genera un fichero class separado. Si el nombre de la clase principal es `UnNombre` por ejemplo, los ficheros .class de las clases anónimas serán `UnNombre$1.class`, `UnNombre$2.class`, `UnNombre$3.class`, etc. Igualmente si pides el nombre de una clase anónima (`getClass().getName()`) obtendrás `UnNombre$1`, etc.

5.5 CLASES ABSTRACTAS.

En ciertas ocasiones es posible que se necesite definir una clase que represente un concepto tan abstracto que nunca vayan a existir

instancias (objetos) de esa clase. ¿Tiene sentido? ¿Y utilidad?

Imagina una aplicación para un centro educativo que utilice las clases `Alumno` y `Profesor`, ambas subclases de `Persona`. Es más que probable que esa aplicación nunca llegue a necesitar objetos de la clase `Persona`, pues son demasiado genéricas como para poder utilizarse (no contienen suficiente información propia). Podrías llegar entonces a la conclusión de que la clase `Persona` ha resultado de utilidad como **clase base** para construir otras clases que heredan de ella, pero no como clase instanciable de la cual vayan a existir objetos. A este tipo de clases se les llama clases abstractas.

Las clases abstractas son clases que nunca serán instanciadas, pero que proporcionan un marco a seguir por sus clases derivadas dentro de una jerarquía de herencia.

La posibilidad de declarar clases abstractas es una de las características más útiles de los lenguajes orientados a objetos, pues permiten dar unas líneas generales de cómo debe ser una clase antes incluso de implementar sus métodos o implementando solamente algunos de ellos. Esto resulta especialmente útil cuando las distintas clases derivadas deban proporcionar los mismos métodos indicados en la clase base abstracta, pero su implementación sea específica y diferente para cada subclase.

Imagina que estás trabajando en un entorno de manipulación de objetos gráficos y necesitas trabajar con líneas, círculos, rectángulos, etc. Estos objetos tendrán en común algunos atributos que representen su estado (ubicación, color del contorno, color de relleno, etc.) y algunos métodos que modelen su comportamiento (dibujar, rellenar con un color, escalar, desplazar, rotar, etc.). Algunos de ellos

serán comunes para todas ellas (por ejemplo la ubicación o el desplazamiento) y sin embargo otros (como por ejemplo dibujar) necesitarán una implementación específica dependiendo del tipo de figura.

Pero, en cualquier caso, todas necesitan esos métodos (tanto un círculo como un rectángulo necesitan el método `dibujar()`, aunque se lleven a cabo de manera diferente). En este caso resultaría muy útil disponer de una clase abstracta `ObjetoGrafico` donde se definan las líneas generales (algunos atributos comunes, algunos métodos comunes implementados y algunos métodos genéricos comunes sin implementar) de un objeto gráfico y más adelante, según se vayan definiendo clases especializadas (líneas, círculos, rectángulos, elipses...), se irán concretando en cada subclase aquellos métodos que se dejaron sin implementar en la clase abstracta.

DECLARACIÓN DE CLASE ABSTRACTA

Una clase abstracta es una clase que no se puede instanciar, es decir, que no se pueden crear objetos a partir de ella. La idea es permitir que otras clases deriven de ella, proporcionando un modelo genérico y algunos métodos de utilidad general. Las clases abstractas se declaran mediante el modificador **abstract**:

```
[modificador] abstract class nombreClase [herencia]
[interfaces] {  …  }
```

Una clase abstracta puede contener en su interior métodos declarados como **abstract** (métodos para los cuales sólo se indica la cabecera, pero no se proporciona su implementación). En tal caso, la clase tendrá que ser necesariamente también **abstract**. **Esos métodos tendrán que ser obligatoriamente implementados en sus clases derivadas.**

Por otro lado, una clase abstracta también puede contener métodos totalmente implementados (no abstractos), los cuales serán heredados por sus clases derivadas y podrán ser utilizados sin necesidad de definirlos (pues ya están implementados).

Cuando trabajes con clases abstractas debes tener en cuenta:

- Una clase abstracta sólo puede usarse para crear nuevas clases derivadas. No se puede hacer un **new** de una clase abstracta. Genera un error de compilación.
- Una clase abstracta puede contener métodos totalmente definidos (no abstractos) y métodos sin definir (métodos abstractos).

EJEMPLO 23: Basándote en la jerarquía de clases de ejemplo (`Persona`, `Alumno`, `Profesor`) las modificamos para que `Persona` sea, a partir de ahora, una clase abstracta (no instanciable) y las otras dos clases sigan siendo clases derivadas de ella, pero sí instanciables.

En este caso lo único que habría que hacer es añadir el modificador abstract a la clase Persona. El resto de la clase permanecería igual y las clases `Alumno` y `Profesor` no tendrían porqué sufrir ninguna modificación.

```
public abstract class Persona {
   protected String nombre;
   protected String apellidos;
   protected LocalDate fechaNacim;
   ...
}
```

A partir de ahora no podrán existir objetos de la clase Persona. El compilador generaría un error.

Existen una gran cantidad de clases abstractas en la API de Java. Aquí

tienes un par de ejemplos:

- La clase java.awt.Component:

```
public abstract class Component extends Object implements
ImageObserver, MenuContainer, Serializable...
```

- La clase javax.swing. AbstractButton:

```
public abstract class AbstractButton extends JComponent
implements ItemSelectable, SwingConstants...
```

MÉTODOS ABSTRACTOS

Un método abstracto es un método declarado en una clase para el cual esa clase no proporciona la implementación. Si una clase dispone de al menos un método abstracto forzosamente debe ser una clase abstracta. Toda clase que herede (sea subclase) de una clase abstracta debe implementar todos los métodos abstractos de su superclase o bien volverlos a declarar como abstractos (y por tanto también sería abstracta).

Para declarar un método abstracto en Java se utiliza el modificador **abstract**) es un método cuya implementación no se define, sino que se declara únicamente su interfaz (cabecera) para que su cuerpo sea implementado más adelante en una clase derivada.

Un método se declara como abstracto mediante el uso del modificador abstract (como en las clases abstractas):

[modificadores] **abstract** <tipo> <nombre> ([parámetros]) [excepciones];

Estos métodos tendrán que ser obligatoriamente redefinidos (en realidad "definidos", pues aún no tienen contenido) en las clases derivadas. Si en una clase derivada se deja algún método abstracto sin implementar, esa clase derivada será también una clase abstracta.

Cuando una clase contiene un método abstracto tiene que declararse como abstracta obligatoriamente.

Imagina que tienes una clase `Empleado` genérica para diversos tipos de empleado y 3 clases derivadas: `EmpleadoFijo` (tiene un salario fijo más ciertos complementos), `EmpleadoTemporal` (salario fijo más otros complementos diferentes) y `EmpleadoComercial` (una parte de salario fijo y unas comisiones por cada operación). Incluso en el futuro podrían aparecer más tipos de trabajadores. La clase `Empleado` debe definir un método `calcularNomina()`, y lo mejor es que sea abstracto pues sabes que el método será necesario para cualquier tipo de empleado (todo empleado cobra una nómina) pero no se calcula igual para cada uno e incluso no sabes como se calculará en el futuro.

El cálculo en sí de la nómina será diferente si se trata de un empleado fijo, un empleado temporal o un empleado comercial, y será dentro de las clases especializadas de `Empleado` (`EmpleadoFijo`, `EmpleadoTemporal`, `EmpleadoComercial`) donde se implementen de manera específica el cálculo de las mismas.

Debes tener en cuenta al trabajar con métodos abstractos:

- Un método abstracto implica que la clase a la que pertenece tiene que ser abstracta, pero eso no significa que todos los métodos de esa clase tengan que ser abstractos.
- Un método abstracto no puede ser privado (no se podría implementar, dado que las clases derivadas no tendrían acceso a él).
- Los métodos abstractos no pueden ser estáticos, pues los métodos estáticos no pueden ser redefinidos (y los métodos abstractos necesitan ser redefinidos).

Puedes echar un vistazo a este vídeo en el se explica el funcionamiento de las clases abstractas y se muestra un ejemplo de creación y utilización: https://www.youtube.com/watch?v=ztpYmmecfQs

EJEMPLO 24: Basándote en la jerarquía de clases `Persona`, `Alumno`, `Profesor`, crea un método abstracto llamado `mostrar()` para la clase `Persona`. Dependiendo del tipo de persona (alumno o profesor) el método tendrá que mostrar unos u otros datos personales (habrá que hacer implementaciones específicas en cada clase derivada). Una vez hecho esto, implementa completamente las tres clases (con todos sus atributos y métodos) y utilízalas en un pequeño programa de ejemplo que cree un objeto de tipo `Alumno` y otro de tipo `Profesor`, los rellene con información y muestre esa información en la pantalla a través del método `mostrar()`. Dado que el método `mostrar()` no va a ser implementado en la clase `Persona`, será declarado como abstracto y no se incluirá su implementación:

```
protected abstract void mostrar();
```

Recuerda que el simple hecho de que la clase `Persona` contenga un método abstracto hace que sea clase sea abstracta (y deberá indicarse como tal en su declaración): `public abstract class Persona`. En el caso de la clase `Alumno` habrá que hacer una implementación específica del método mostrar y lo mismo para el caso de la clase Profesor.

1. Método mostrar para la clase Alumno.

```
// Redefinición del método abstracto mostrar en la clase Alumno
public void mostrar() {
 SimpleDateFormat formatoFecha = new  SimpleDateFormat("dd/MM/yyyy");
 String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());
 System.out.printf("Nombre: %s\n", this.nombre);
 System.out.printf("Apellidos: %s\n", this.apellidos);
```

```
 System.out.printf("Fecha de nacimiento: %s\n", Stringfecha);
 System.out.printf("Grupo: %s\n", this.grupo);
 System.out.printf("Grupo: %5.2f\n", this.notaMedia);
}
```

2. Método mostrar para la clase Profesor.

```
// Redefinición del método abstracto mostrar en la clase Profesor
public void mostrar () {
 SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
 String Stringfecha= formatoFecha.format(this.fechaNacim.getTime());
 System.out.printf("Nombre: %s\n", this.nombre);
 System.out.printf("Apellidos: %s\n", this.apellidos);
 System.out.printf("Fecha de nacimiento: %s\n", Stringfecha);
 System.out.printf("Especialidad: %s\n", this.especialidad);
 System.out.printf("Salario: %7.2f euros\n", this.salario);
}
```

3. Programa de ejemplo de uso. Un pequeño programa de ejemplo de uso del método mostrar en estas dos clases podría ser:

```
// Declaración de objetos
Alumno alumno;
Profesor profe;
// Creación de objetos (llamada a constructores)
alumno= new Alumno ("Juan", "Torres", new GregorianCalendar (1990,
10, 6), "1DAM-B", 7.5);
profe= new Profesor ("Antonio", "Campos", new GregorianCalendar
(1970, 8, 15), "Mates", 2000);
// Utilización del método mostrar
alumno.mostrar();
profesor.mostrar();
```

CLASES Y MÉTODOS FINALES

En unidades anteriores has visto el modificador **final**, aunque sólo lo has utilizado por ahora para variables que toman un valor ya no pueden ser modificados). Pero este modificador también puede ser utilizado con clases y con métodos (con un comportamiento que no es exactamente igual, aunque puede encontrarse cierta analogía: no se

permite heredar o no se permite redefinir).

Una clase declarada como **final** no puede ser heredada, es decir, no puede tener clases derivadas. La jerarquía de clases a la que pertenece acaba en ella (no tendrá clases hijas):

[modificador_acceso] **final class** nombreClase [herencia] [interfaces]

Un método también puede ser declarado como final, en tal caso, ese método no podrá ser redefinido en una clase derivada:

[modificador_acceso] **final** <tipo> <nombreMetodo> ([parámetros]) [excepciones]

Si intentas redefinir un método final en una subclase se producirá un error de compilación.

Además de en la declaración de atributos, clases y métodos, el modificador **final** también podría aparecer acompañando a un método de un parámetro. En tal caso no se podrá modificar el valor del parámetro dentro del código del método. Por ejemplo:

```
public final metodoEscribir(int p1, final int p2);
```

Dada la gran cantidad de contextos diferentes en los que se puede encontrar el modificador **final**, vale la pena hacer un repaso de todos los lugares donde puede aparecer y cuál sería su función en cada uno:

EJEMPLO 25: Veamos un ejemplo de cada posibilidad:

1. Modificador de una clase.

```
public final class ClaseSinDescendencia { /* "no heredable" */ }
```

2. Modificador de un atributo.

```
public class ClaseEjemplo {
```

```
        // Valor constante conocido en tiempo de compilación
        final double PI= 3.14159265;
        // Valor constante conocido en tiempo de ejecución
        final int SEMILLA= (int) Math.random()*10+1;
        …
    }
```

3. Modificador de un método.

```
    public final metodo(int p1) { /* Método "no redefinible" */ }
```

4. Modificador en una variable referencia.

```
    // Referencia constante: siempre se apuntará al mismo objeto
    // Alumno recién creado, aunque este objeto pueda modificarse
    final Alumno PRIMER_ALUMNO= new Alumno("Pepe", "Torres",9.55);
    // Si la variable no es una referencia (tipo primitivo), sería una
    // constante más (como un atributo constante).
    final int NUMERO_DIEZ = 10; // Valor constante
```

5. Modificador en un parámetro de un método.

```
    void metodoConParamFijos(final int p1, final int p2) {
        // p1 y p2 no podrán sufrir modificaciones aquí dentro
        …
    }
```

Distintos contextos en los que puede aparecer el modificador final	
Lugar	**Función**
Como modificador de clase.	La clase no puede tener subclases.
Como modificador de atributo.	El atributo no podrá ser modificado una vez que tome un valor. Sirve para definir constantes.
Como modificador al declarar un método	El método no podrá ser redefinido en una clase derivada.
Como modificador al declarar una variable referencia.	Una vez que la variable tome un valor referencia (un objeto), no se podrá cambiar. La variable siempre apuntará al mismo objeto, lo cual no quiere decir que ese objeto no pueda ser modificado internamente a través de sus métodos. Pero la variable no podrá apuntar a otro objeto diferente.
Como modificador en un parámetro de un método.	El valor del parámetro (ya sea un tipo primitivo o una referencia) no podrá modificarse dentro del código del método.

Figura 8: Resumen del uso de final.

EJERCICIO 11: Utilicemos las clases abstractas:

a) Crea una clase pública y abstracta llamada `Vehiculo`. Como campos tendrá quien le proporciona la potencia para moverse (`potencia`), el número de ruedas (`ruedas`) y el `precio`.

b) Crea dos subclases suyas públicas, ambas usan el `precio`: `Velero` que solo usa la `potencia` y `Bicicleta` que solamente usa el número de ruedas. Además cada una de las clases incluye límites de precio para cada tipo de vehículo.

c) El constructor de Vehiculo acepta dos parámetros y llama a tres setters: El primero fija la `potencia`, el segundo el nº de ruedas y el tercero pregunta al usuario por el precio del vehículo. Define los getters y setters de los campos salvo el setter del precio. El setter del precio es un método abstracto porque cada subclase preguntará por el precio y tendrá diferentes límites de precio para cada tipo de vehículo. Define el método como abstracto.

d) La clase `Velero` importa todas las clases swing: `import javax.swing.*` (por tanto debe indicar en el módulo del proyecto que necesita la el módulo `java.desktop`) y define un campo que contiene la longitud de la embarcación, un entero. El constructor debe llamar al constructor de su clase padre pasando valores de potencia y número de ruedas y llamar al setter de la longitud que la pregunta usando el diálogo de swing: `JoptionPane.showInputDialog(ventanaPadre, mensaje)`, la longitud también tendrá getter. Por último debes sobreescribir el método abstracto, preguntarás el precio con el diálogo de swing y como límite tendrás una cantidad constante definida en el método donde se usa de la propia clase `Velero` de 100_000.00 euros. Por último sobreescribes `toString()`.

e) Ahora completamos la clase `Bicicleta`. Implementa el

constructor por defecto que usa el de su clase padre pasándole `"una persona"` para potencia y 2 para el número de ruedas. Piensa en cómo hacer otro constructor teniendo en cuenta que una bicicleta siempre tendrá dos ruedas y puede ser impulsada normalmente por una persona o por algún tipo de motor eléctrico. El precio el límite será `3_999.99 euros`. Crea getters y setters necesarios y sobreescribe `toString()`.
f) Crea la clase ejecutable `DemoVehiculo` que utilice las clases anteriores. Crea un nuevo velero y una nueva bicleta usando sus constructores por defecto (sin parámetros). Utiliza la llamada a `JoptionPane.showMessageDialog(ventana,"\nDescripciones:\n" +...)` para sacar los datos del velero y de la bicicleta que has creado.

5.6. INTERFACES.

La herencia permite definir especializaciones (o extensiones) de una clase base que ya existe sin tener que volver a repetir de todo el código de ésta. Este mecanismo da la oportunidad de que la nueva clase especializada (o extendida) disponga de toda la interfaz que tiene su clase base.

También se ha comentado cómo los métodos abstractos permiten establecer un patrón que marca las líneas generales de un comportamiento común de superclase que deberían compartir de todas las subclases.

Si llevamos al límite esta idea, podrías llegar a tener una clase abstracta donde todos sus métodos fueran abstractos. De este modo estarías definiendo solamente el comportamiento, sin ningún método implementado, de las posibles subclases que heredarán de esa clase

abstracta.

La idea de una interfaz (o interface) es precisamente ésa: disponer de un mecanismo que permita especificar cuál debe ser el comportamiento que deben tener todos los objetos que formen parte de una determinada clasificación (no necesariamente jerárquica con el sentido de ES UN/UNA).

Una interfaz consiste principalmente en una lista de declaraciones de métodos sin implementar, que definen un comportamiento. Si se desea que una clase tenga ese comportamiento, tendrá que implementar esos métodos establecidos en la interfaz.

En este caso **no se trata de una relación de herencia** (la clase A es una especialización de la clase B, o la subclase A "**es un**" B), sino más bien una relación "**de implementación de comportamientos**" (la clase A implementa los métodos establecidos en la interfaz B, o los comportamientos indicados por B son llevados a cabo por A; pero no significa que A sea un B, más bien significa "**A sabe hacer B**").

Imagina que estás diseñando una aplicación que trabaja con clases que representan distintos tipos de animales. Algunas de las acciones que quieres que lleven a cabo están relacionadas con el hecho de que algunos animales sean depredadores (por ejemplo: observar una presa, perseguirla, comérsela, etc.) y otras sean presas (observar, huir, esconderse, etc.). Si creas la clase León, esta clase podría implementar una interfaz `Depredador`, mientras que otras clases como `Gacela` implementarían las acciones de la interfaz `Presa`. Por otro lado, podrías tener también el caso de la clase `Rana`, que implementaría las acciones de la interfaz `Depredador` (pues es cazador de pequeños insectos), pero también la de `Presa` (pues puede ser cazado y necesita

las acciones necesarias para protegerse).

CONCEPTO DE INTERFACE

Una interfaz en Java consiste en una lista de declaraciones de métodos sin implementar, junto con un conjunto de constantes.

Estos métodos sin implementar indican un comportamiento, un tipo de conducta, aunque no especifican cómo será ese comportamiento (implementación), pues eso dependerá de las características específicas de cada clase que decida implementar esa interfaz.

Podría decirse que una interfaz se encarga de establecer qué comportamientos hay que tener (qué métodos), pero no dice nada de cómo deben llevarse a cabo esos comportamientos (implementación).

En cierto modo podrías imaginar el concepto de interfaz como un guión que dice: "éste es el protocolo de actuación que deben presentar todas las clases que implementen esta interfaz". Se proporciona una lista de métodos públicos y si quieres dotar a tu clase de esa interfaz, tendrás que definir todos y cada uno de esos métodos públicos.

En conclusión: una interfaz se encarga de establecer unas líneas generales sobre los comportamientos (métodos) que deberían tener los objetos de toda clase que implemente esa interfaz, es decir, que no indican lo que el objeto es (de eso se encarga la clase y sus superclases), sino acciones (capacidades) que el objeto debería ser capaz de realizar.

Es por esto que el nombre de muchas interfaces en Java termina con sufijos del tipo "-able", "-or", "-ente" y cosas del estilo, que significan algo así como capacidad o habilidad para hacer o ser receptores de algo (configurable, serializable, modificable, clonable, ejecutable,

administrador, servidor, buscador, depredador, etc.), dando así la idea de que se tiene la capacidad de llevar a cabo el conjunto de acciones especificadas en la interfaz.

Imagina por ejemplo la clase `Coche`, subclase de `Vehículo`. Los coches son vehículos a motor, lo cual implica una serie de acciones como, por ejemplo: arrancar el motor o detener el motor. Esa acción no la puedes heredar de `Vehículo`, pues no todos los vehículos tienen porqué ser a motor (piensa por ejemplo en una clase `Bicicleta`), y no puedes heredar de otra clase pues ya heredas de `Vehículo`.

Una solución podría ser crear una interfaz `Arrancable`, que proporcione los métodos típicos de un objeto a motor (no necesariamente vehículos). De este modo la clase `Coche` sigue siendo subclase de `Vehículo`, pero también implementaría los comportamientos de la interfaz `Arrancable`, los cuales podrían ser también implementados por otras clases, hereden o no de `Vehículo` (por ejemplo una clase `Motocicleta` o bien una clase `Motosierra`). La clase `Coche` implementará su método arrancar de una manera, la clase `Motocicleta` lo hará de otra (aunque bastante parecida) y la clase `Motosierra` de otra forma probablemente muy diferente, pero todos tendrán su propia versión del método `arrancar()` como parte de la interfaz `Arrancable`.

Según esta concepción, podrías hacerte la siguiente pregunta: ¿podrá una clase implementar varias interfaces? La respuesta en este caso es afirmativa. Una clase puede adoptar distintos modelos de comportamiento establecidos en diferentes interfaces. Es decir una clase puede implementar varias interfaces.

DEFINIR INTERFACES

La declaración de una interfaz en Java es similar a la declaración de

una clase, aunque con algunas variaciones:

- Se utiliza la palabra reservada **interface** en lugar de **class**.
- Puede utilizarse el modificador **public**. Si incluye este modificador la interfaz debe tener el mismo nombre que el archivo .java en el que se encuentra (exactamente igual que sucedía con las clases). Si no se indica el modificador **public**, el acceso será "de paquete" (como con las clases).
- Todos los miembros de la interfaz (atributos y métodos) son public de manera implícita. No es necesario indicar el modificador **public**, aunque puede hacerse.
- Todos los atributos son de tipo **final** y **public** (tampoco es necesario especificarlo), es decir, constantes y públicos. Hay que darles un valor inicial.
- Todos los métodos son abstractos también de manera implícita (tampoco hay que indicarlo). No tienen cuerpo, tan solo la cabecera.

Como puedes observar, una interfaz consiste esencialmente en una lista de atributos finales (constantes) y métodos abstractos (sin implementar). Su sintaxis quedaría entonces:

```
[public] interface NombreInterfaz {
    [public] [final] tipo atributo1= <valor1>;
    [public] [final] tipo atributo2= <valor2>;
    ...
    [public] [abstract] tipo nombreMetodo1([parámetros]);
    [public] [abstract] tipo nombreMetodo2([parámetros]);
    ...
}
```

Observa que la declaración de los métodos termina en punto y coma, pues no tienen cuerpo, al igual que sucede con los métodos abstractos

de las clases abstractas. El ejemplo de la interfaz `Depredador` que hemos comentado antes podría quedar entonces así:

```
public interface Depredador {
   void localizar(Animal presa);
   void cazar(Animal presa);
   ...
}
```

Serán las clases que implementen esta interfaz (`León`, `Leopardo`, `Cocodrilo`, `Rana`, `Lagarto`, `Hombre`, etc.) las que definan la implementación de cada uno, pues la manera de cazar de un `León` no será misma que la de un `Lagarto`, por ejemplo.

EJEMPLO 26: Crea una interfaz en Java cuyo nombre sea `Imprimible` y que contenga algunos métodos útiles para mostrar el contenido de una clase:

1. Método `contenidoString`, que crea un String con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve. El formato será una lista de pares "nombre=valor" de cada atributo separado por comas y la lista completa encerrada entre llaves:

   `"{ atributo1 = valor1, ..., atributon = valorn}".`

2. Método `contenidoArrayList`, que crea un ArrayList de String con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve.
3. Método `contenidoHashtable`, similar al anterior, pero en lugar devolver en un `ArrayList` los valores de los atributos, se devuelve en una Hashtable en forma de pares (nombre, valor).

Se trata simplemente de declarar la interfaz e incluir en su interior esos tres métodos:

```
public interface Imprimible {
   String    contenidoString();
   ArrayList contenidoArrayList();
   HashTable contenidoHashtable();
}
```

El cómo se implementarán cada uno de esos métodos dependerá exclusivamente de lo que cada clase necesite.

IMPLEMENTAR INTERFACES

Como ya has visto, todas las clases que implementan una interfaz están obligadas a proporcionar una implementación de los métodos de la interfaz, adoptando el modelo de comportamiento propuesto por ésta. Dada una interfaz, cualquier clase puede implementarla utilizando la palabra reservada implements:

```
class NombreClase implements NombreInterfaz { ...
```

De esta manera, la clase está diciendo algo así como "la interfaz indica los métodos que debo implementar, pero voy a ser yo (la clase) quien los implemente".

Es posible que una clase decida implementar varias intrfaces, en ese caso indica sus nombres separándolos por comas:

```
class NombreClase implements Interfaz1, Interfaz2,... {
```

Cuando una clase implementa una interfaz, tiene que redefinir sus métodos nuevamente con acceso público. Con otro tipo de acceso se producirá un error de compilación. Es decir, que del mismo modo que no se podían restringir permisos de acceso en la herencia de clases, tampoco se puede hacer en la implementación de interfaces.

Una vez implementada una interfaz en una clase, los métodos de esa interfaz tienen exactamente el mismo tratamiento que cualquier otro

método, sin ninguna diferencia: se pueden invocar, heredar, redefinir, etc.

En el ejemplo de los depredadores, al definir la clase León, habría que indicar que implementa la interfaz `Depredador`:

```
class Leon implements Depredador {...
```

Y en su interior habría que implementar aquellos métodos que contenga la interfaz:

```
void localizar(Animal presa) {
   // Implementación del método localizar para un león
   ...
}
```

En el caso de clases que pudieran ser a la vez `Depredador` y `Presa`, tendrían que implementar ambas interfaces, como podría suceder con la clase `Rana`:

```
class Rana implements Depredador, Presa { ...
```

Y en su interior habría que implementar aquellos métodos que contengan ambas interfaces, tanto los de `Depredador` (localizar, cazar, etc.) como los de `Presa` (observar, huir, etc.).

EJEMPLO 27: Haz que las clases `Alumno` y `Profesor` implementen la interfaz `Imprimible` que se ha escrito en el ejemplo anterior.

La primera opción que se te puede ocurrir es pensar que en ambas clases habrá que indicar que implementan la interfaz `Imprimible` y por tanto definir los métodos que ésta incluye: `contenidoString`, `contenidoHashtable` y `contenidoArrayList`.

Si las clases `Alumno` y `Profesor` no heredasen de la misma clase habría que hacerlo obligatoriamente así, pues no comparten superclase y

precisamente para eso sirven las interfaces: para definir y obligar a implementar comportamientos que no pertenecen a la estructura jerárquica de herencia en la que se encuentra una clase (de esta manera, clases que no tienen ninguna relación de herencia podrían compartir interfaz, comportamiento).

Pero en este caso podríamos aprovechar que ambas clases sí son subclases de una misma superclase (heredan de la misma) y hacer que la interfaz `Imprimible` sea implementada directamente por la superclase (`Persona`) y de este modo ahorrarnos bastante código. Así no haría falta indicar explícitamente que `Alumno` y `Profesor` implementan la interfaz `Imprimible`, pues lo estarán haciendo de forma implícita al heredar de una clase que ya la implementa (la clase `Persona`, padre de ambas).

Una vez que los métodos de la interfaz estén implementados en la clase `Persona`, tan solo habrá que redefinir o ampliar los métodos de la interfaz para que se adapten a cada clase hija específica (`Alumno` o `Profesor`), ahorrándonos tener que escribir varias veces la parte de código que obtiene los atributos genéricos de la clase `Persona`.

PASO 1. Clase `Persona`. Indicamos que implementa `Imprimible`:

```
public abstract class Persona implements Imprimible { ...
```

Definimos el método `contenidoHashtable` a la manera de como debe ser implementado para la clase `Persona`. Podría quedar, por ejemplo, así:

```
public Hashtable contenidoHashtable () {
   Hashtable conte = new Hashtable(); // Crea Hashtable
   // Añadimos los atributos de la clase
   SimpleDateFormat f = new SimpleDateFormat("dd/MM/yyyy");
   String sFecha= f.format( this.fechaNacim.getTime() );
   conte.put("nombre", this.nombre);
   conte.put("apellidos", this.apellidos);
   conte.put("fechaNacim", sFecha);
```

```
        return conte; // Devolvemos la Hashtable
    }
```

Del mismo modo, definimos también `contenidoArrayList()`:

```
    public ArrayList contenidoArrayList () { ... }
```

Y por último el método `contenidoString`:

```
    public String contenidoString () { ... }
```

PASO 2. Clase Alumno. Esta clase hereda de la clase `Persona`, de manera que hereda los 3 métodos anteriores. Tan solo habrá que redefinirlos para que, aprovechando el código ya escrito en la superclase, se añada la funcionalidad específica que aporta esta subclase.

```
    public class Alumno extends Persona { ...
```

Como puedes observar no ha sido necesario incluir el **`implements`** `Imprimible`, pues el `extends Persona` lo lleva implícito dado que `Persona` ya implementa la interfaz. Lo que haremos será llamar al método que estamos redefiniendo utilizando la referencia a la superclase super.

El método `contenidoHashtable()` podría quedar así:

```
    public Hashtable contenidoHashtable() {
        Hashtable co = super.contenidoHashtable(); //Llama a superclase
        co.put("grupo", this.grupo); // Añade atributos propios
        co.put("notaMedia", this.notaMedia);
        return co;  // Devolvemos la Hashtable rellena
    }
```

PASO 3. Clase `Profesor`. En este caso habría que proceder exactamente de la misma manera que con la clase `Alumno`: redefiniendo los métodos de la interfaz `Imprimible` para añadir la funcionalidad específica que aporta.

EJERCICIO 12: Sobreescribe los métodos que la clase `Profesor` hereda de `Persona` y que `Persona` implementa para cumplir con el comportamiento de la interface `Imprimible`, sabiendo que los datos que diferencian a un `Profesor` de una `Persona` son `double salario` y `String especialidad`.

HERENCIA DE INTERFACES

Las interfaces, al igual que las clases, también utilizan la herencia. Para indicar que una interfaz hereda de otra se indica nuevamente con la palabra reservada **extends**. Pero en este caso sí se permite la herencia múltiple entre interfaces. Si se hereda de más de una interfaz se indica con la lista de interfaces separadas por comas. Por ejemplo, dadas las interfaces `InterfazUno` e `InterfazDos`:

```
public interface InterfazUno {
   // Métodos y constantes de la interfaz Uno
}
public interface InterfazDos {
   // Métodos y constantes de la interfaz Dos
}
```

Podría definirse una nueva interfaz que herede de ambas:

```
public interface InterfazCompleja extends InterfazUno, InterfazDos{
  // Métodos y constantes específicos de la interfaz compleja...
}
```

EJEMPLO 28: Localiza en la API de Java algún ejemplo de interfaz que herede de una o varias interfaces (puedes consultar la documentación de referencia de la API de Java).

Existen una gran cantidad de interfaces en la API de Java que heredan de otras interfaces. Aquí tienes un par de ejemplos:

- La interfaz `java.awt.event.ActionListener`, hereda de `java.util.EventListener`.

- La interfaz `org.omg.CORBA.Policy`, hereda de `org.omg.CORBA.PolicyOperations`, `org.omg.CORBA.Object` y `org.omg.CORBOA.portable.IDLEntity`.

5.6.1. ¿QUÉ USAR? CLASE ABSTRACTA O INTERFACE.

Observando el concepto de interface, podría caerse en la tentación de pensar que es prácticamente lo mismo que una clase abstracta en la que todos sus métodos sean abstractos.

Es cierto que existe un gran parecido formal entre una clase abstracta y una interface, de forma que en ocasiones se pueden utilizar indistintamente una u otra para obtener un mismo fin. Pero, a pesar de ese gran parecido, existen algunas diferencias, no sólo formales, sino también conceptuales, muy importantes:

- Una clase no puede heredar de varias clases, aunque sean abstractas (herencia múltiple). Sin embargo sí puede implementar una o varias interfaces y además seguir heredando de una clase.
- Una interfaz no puede definir métodos (no implementa su contenido) tan solo los declara.
- Una interfaz puede hacer que dos clases tengan un mismo comportamiento independientemente de sus ubicaciones en una jerarquía de clases (no tienen que heredar las dos de una misma superclase común, pues no siempre es posible según la naturaleza y propiedades de cada clase).
- Una interfaz permite establecer un comportamiento de clase sin apenas dar detalles, pues esos detalles aún no son conocidos (dependerán del modo en que cada clase decida implementar la interfaz).

- Las interfaces tienen su propia jerarquía, diferente e independiente de la jerarquía de clases.

Todo esto puede resumirse en que: una clase abstracta proporciona una interfaz disponible sólo a través de la herencia. Sólo quien herede de esa clase abstracta dispondrá de esa interfaz. Si una clase no pertenece a esa misma jerarquía (no hereda de ella) no podrá tener esa interfaz. Eso significa que para poder disponer de la interfaz podrías:

1. Volver a escribirla para esa jerarquía de clases. Lo cual no parece una buena solución.
2. Hacer que la clase herede de la superclase que proporciona la interfaz que te interesa, sacándola de su jerarquía original y convirtiéndola en clase derivada de algo de lo que conceptualmente no debería ser una subclase. Es decir, estarías forzando una relación "es un" cuando en realidad lo más probable es que esa relación no exista. Tampoco parece la mejor forma de resolver el problema.

Sin embargo, una interfaz puede ser implementada por cualquier clase, permitiendo que clases que no tengan ninguna relación entre sí (pertenecen a distintas jerarquías) puedan compartir un determinado comportamiento (una interfaz) sin tener que forzar una relación de herencia que no existe entre ellas.

A partir de ahora podemos hablar de otra posible relación entre clases: **compartir un comportamiento (interfaz)**. Dos clases podrían tener en común un determinado conjunto de comportamientos sin que necesariamente exista una relación jerárquica entre ellas. Tan solo cuando haya realmente una relación de tipo "es un" se producirá herencia.

Si sólo vas a proporcionar una lista de métodos abstractos (interfaz), sin definiciones de métodos ni atributos de objeto, suele ser recomendable definir una interfaz antes que clase abstracta. Es más, cuando vayas a definir una supuesta clase base, puedes comenzar declarándola como interfaz y sólo cuando veas que necesitas definir métodos o variables miembro, puedes entonces convertirla en clase abstracta (no instanciable) o incluso en una clase instanciable.

Conceptualmente SER o PODER es la diferencia:

- **Clases Abstractas**: indica que una clase está incompleta y que sólo se va a utilizar como clase base. No se podrá crear un nuevo objeto directamente de ella. Todas las clases que hereden de esta clase abstracta la toman como base de su comportamiento, formando parte de su esqueleto, lo podríamos definir como **subclase «ES UN(A)» superclase abstracta**.
- **Interfaces**: una serie de métodos, propiedades, índices y eventos, que podemos definir, y que la clase que se comprometa a implementarlos debe proporcionar. Esto significa que clases que implementen esta interfaz pueden ser diferentes casi por completo, pero van a contener todo lo que la interfaz define. Por esto mismo, podemos decir que es un **clase «PUEDE HACER» interfaz**.

EJEMPLO 29: Utiliza interfaces y clases abstractas en el diseño de las clases del siguiente escenario. Queremos definir una serie de animales, un pato, un tiburón, un gorrión y un mono. Si lo pensamos, todos SON animales, por lo tanto esta será nuestra clase abstracta.

```
public abstract class Animal {
    private String nombre;
    public Animal(String n) { nombre = n;  }
    public getNombre(){ return nombre; }
```

```
        public abstract void come(String comida, int cantidad);
    }
```

El pato y el gorrión, a su vez, PUEDEN volar y el tiburón y el pato PUEDEN nadar. Éstas serán nuestras interfaces.

```
public interface Volador {  void vuela();  }
public interface Nadador {  void nada();   }
```

Este sería el resultado:

```
public class Pato extends Animal implements Volador, Nadador {
    public Pato() { super("pato"); }
    public void come(String comida, int cantidad) { /*...*/ }
    public void vuela() { /*...*/ }
    public void nada() { /*...*/ }
}

public class Tiburon extends Animal implements Nadador {
    public Tiburon() { super("tiburon"); }
    public void come(String comida, int cantidad){ /*...*/ };
    public void nada() { /*...*/ };
}

public class Gorrion extends Animal implements Volador {
    public Gorrion() { super("gorrion"); }
    public void come(String comida, int cantidad){ /*...*/  };
    public void vuela() { /*...*/ }
}

public class Mono extends Animal {
    public Mono() { super("mono"); }
    public void come(String comida, int cantidad){ /*...*/  };
 }
```

5.6.2. ALGUNAS INTERFACES DE JAVA.

El trabajo con interfaces es algo habitual en el desarrollo de aplicaciones en Java. Es por tanto muy importante comprender correctamente su funcionamiento y la interacción con las distintas bibliotecas (paquetes de clases e interfaces) que proporcionan las

APIs. Estas clases e interfaces son fundamentales para la creación de aplicaciones y tendrás que utilizarlas en multitud de ocasiones (además de las que tengas que desarrollar por ti mismo).

Vamos a ver varios ejemplos de una interfaz proporcionada por la API de Java que puede ser implementada por alguna clase creada por ti dentro de una pequeña aplicación. Hemos escogido la interfaz **ActionListener**.

ACTIONLISTENER

Las clases que quieran realizar una determinada acción cada vez que se produzca cierto evento en el sistema deben implementar esta interfaz. Este tipo de interfaces se encuentran dentro de los **Event Listeners** u "oyentes de eventos" y son útiles para detectar que se ha producido un determinado evento asíncrono durante la ejecución de tu aplicación (pulsación de una tecla, clic de ratón, etc.). Son intensivamente utilizadas en el desarrollo de aplicaciones con GUI.

Es interesante que estés al día sobre los conceptos del modelo de gestión de eventos de las GUI: evento, oyente, gestión de eventos, etc. Si deseas profundizar algo más en el tema de los Event Listeners puedes echar un vistazo a los tutoriales de iniciación de Java en el sitio web de Oracle (en inglés) en el que se explican detalladamente ejemplos sencillos así como otros más complejos: enlace.

Vamos a ver un ejemplo: se trata de desarrollar una pequeña aplicación de escritorio (utilizando la API de Swing) con una ventana que contenga un par de botones y que al pulsar alguno de esos botones (evento), la aplicación sea capaz de detectar que se ha producido ese evento y por tanto realizar una determinada acción (por ejemplo generar un sonido o mostrar un determinado texto).

Para ello es necesario que la clase que vaya a gestionar el evento sea un "oyente" de ese evento, es decir, que sea capaz de enterarse de que se ha producido ese evento. Esa capacidad o habilidad la proporciona la API de Java a través de las interfaces de tipo "oyente". En concreto vamos a implementar la interfaz `ActionListener`. La interfaz `ActionListener` tiene un único método: `actionPerformed`. En nuestro ejemplo podríamos hacer lo siguiente:

- Definir una ventana principal, por ejemplo una clase basada en el tipo frame (subclase de `JFrame`) que implemente la interfaz `ActionListener`.
- Definir uno o varios botones (objetos de la clase `JButton`) dentro de la ventana.
- Asociar como oyente de los objetos de tipo botón la propia ventana principal (frame), pues va a ser capaz de gestionar eventos (implementa el método `actionPerformed`).
- Implementar el método actionPerformed como un método de la ventana principal (frame) para reaccionar de algún modo cuando se produzca algún evento desencadenado por un botón (pues se ha establecido a la ventana principal como oyente y gestora de los eventos del botón).

Ejemplo de implementación de la interfaz ActionListener

Creamos un nuevo proyecto Java de tipo "Aplicación de escritorio Java", creamos una clase principal que herede de JFrame e incluimos las	`import java.awt.*`
	`import javax.swing.JFrame;`
	
	`public class Ejemplo extends JFrame {`
	`  public static void main(String[] args) {}`
	`}`

bibliotecas de la intefaz gráfica	

Incrustamos algunos componentes gráficos en el frame que acabamos de crear	`import java.awt.*`
	`import javax.swing.JFrame;`
	
	`public class Ejemplo extends JFrame {`
	`  JPanel p;`
	`  JButton b1, b2;`
	`  public Ejemplo() {`
	`  b1 = new JButton("Un pitido");`
	`  b2 = new JButton("Dos pitidos");`
	`  p = new JPanel();`
	`  p.add(b1);`
	`  p.add(b2);`
	`  this.getContentPane().add(p);`
	`  }`
	`}`

Añadimos el oyente (el objeto actual) a los componentes gráficos que son botones, así los botones envían los eventos `ActionEvent` al oyente.	`import java.awt.*`
	`import javax.swing.JFrame;`
	
	`public class Ejemplo extends JFrame implements ActionListener {`
	`  JPanel p;`
	`  JButton b1, b2;`
	`  public Ejemplo(String[] args) {`
	`  b1 = new JButton("Un pitido");`
	`  b2 = new JButton("Dos pitidos");`
	`  p = new JPanel();`
	`  p.add(b1);`
	`  p.add(b2);`
	`  this.getContentPane().add(p);`
	`  b1.addActionListener(this);`
	`  b2.addActionListener(this);`
	`  }`
	`}`

Implementa el método `actionPerformed()` al comprometerse con `ActionListener`	`// ActionPerformed es necesario para la`
	`// interfaz ActionListener`
	`public void ActionPerformed(ActionEvent e) {`
	`  // Averigura botón pulsado`
	`  JButton pulsado = (JButton)e.getSource();`

	`if( pulsado == b1 ) {`
	`  Toolkit.getdefaultToolkit().beep();`
	`} else if (pulsado == b2) {`
	`  Toolkit.getdefaultToolkit().beep();`
	`  Toolkit.getdefaultToolkit().beep();`
	`}`
	`}`

Implementamos el método main	`public static void main(String[] args) {`
	`  Ejemplo ventana = new Ejemplo();`
	`  ventana.setTitle("Action Listener");`
	`  ventana.setSize(300,80);`
	`  ventana.setVisible(true);`
	`}`

Algunas otras interfaces comunes de la API de Java

Utilizarás mucho las interfaces cuando desarrolles aplicaciones de Java. La API de Java contiene numerosas interfaces y muchos de los métodos de la API de Java reciben argumentos de interfaces y devuelven valores interfaces. **Algunas que debes ir conociendo:**

- **Comparable y Comparator** Java contiene varios operadores de comparación (`<`, `<=`, `>`, `>=`, `==`, `!=`) que permiten comparar valores primitivos. Pero estos operadores no se pueden utilizar para comparar objetos. Las interfaces **Comparable** y **Comparator** se utilizan para permitir que los objetos de una clase se comparen entre sí (establecer un orden: `objeto1 > objeto2`) lo que permite ordenar por ejemplo.
- **Serializable y Externalizable** Una interfaz que se utiliza para indicar clases cuyos objetos pueden escribirse en (serializarse), o leerse desde (deserializarse) algún tipo de almacenamiento (archivo en disco, campo de base de datos, string) o transmitirse a través de una red en forma de array de bytes en

el caso de `Externalizable`.

- **Runnable y Callable**, Las implementan cualquier clase que represente una tarea a realizar en paralelo mediante threads. La interfaz contiene un método, **run()**, que describe el comportamiento de un objeto al ejecutarse.
- **Interfaces de escucha de eventos de las GUI**: en una GUI, por ejemplo, en su navegador Web, podrías escribir en un campo de texto la dirección de un sitio Web para visitarlo, o podrías hacer clic en un botón para regresar al sitio anterior. El navegador Web responde a su interacción y realiza una tarea. La interacción se conoce como **un evento** y el código que utiliza el navegador para responder a un evento se conoce como **manejador de eventos**. Éstos se declaran en clases que implementan una interfaz de escucha de eventos apropiada. Cada interfaz de escucha de eventos especifica uno o más métodos que deben implementarse para responder a las interacciones de los usuarios.
- **AutoCloseable** Es implementada por clases que pueden usarse con la instrucción **try con recursos** para ayudar a evitar fugas de recursos.

EJERCICIO 13: Crea la interface `Asegurable` para usarla en clases que creen objetos que puedan ser cubiertos por un seguro.

a) Crea la interface pública `Asegurable` con los métodos `setCobertura()` y `getCobertura()` que escriben y leen el importe por el que aseguras un objeto (son como un setter y un getter).

b) Crea la clase pública `CocheAsegurado` que extienda a `Vehiculo` e implemente `Asegurable`. Añade una variable miembro con el importe cubierto por el seguro (`double` llamada `cobertura`). Añade un constructor que llame al de la superclase pasando potencia y

ruedas. Implementa el método abstracto con un valor máximo de `60_000.00` euros. Implementa los métodos de la interface: El método `setCobertura()` fija el valor asegurado al 90% del precio del coche y el método `getCobertura()` es como un getter de la clase. Sobreescribe `toString()`.
c) Crea el programa `DemoCocheAsegurado` que cree uno y lo imprima en un diálogo de ventana.

5.6.3. HERENCIA MÚLTIPLE CON INTERFACES.

Una interfaz no tiene espacio de almacenamiento asociado (no se van a instanciar objetos del tipo de la interfaz), es decir, no tiene implementación.

En algunas ocasiones es posible que interese representar la situación de que "una clase X es de tipo A, de tipo B y de tipo C", siendo A, B, C clases disjuntas (no heredan unas de otras). Hemos visto que sería un caso de herencia múltiple que Java no permite.

Para poder simular algo así, podrías definir tres interfaces A, B, C que indiquen los comportamientos (métodos) que se deberían tener según se pertenezca a una supuesta clase A, B, o C, pero sin implementar ningún método concreto ni atributos de objeto (sólo interfaz).

De esta manera la clase X podría a la vez:

1. Implementar las interfaces A, B, C, que le dan los comportamientos que quería heredar de las clases A, B, C.
2. Heredar de otra clase Y, que le proporcionaría determinadas características dentro de su jerarquía de objeto (atributos, métodos implementados y métodos abstractos).

En el ejemplo que hemos visto de las interfaces `Depredador` y `Presa`, tendrías un ejemplo de esto: la clase `Rana`, que es subclase de `Anfibio`, implementa una serie de comportamientos propios de un `Depredador` y a la vez, otros más propios de una `Presa`. Esos comportamientos (métodos) no forman parte de la superclase `Anfibio`, sino de las interfaces. Si se decide que la clase `Rana` debe de llevar a cabo algunos otros comportamientos adicionales, podrían añadirse a una nueva interfaz y la clase `Rana` implementaría una tercera interfaz.

De este modo, con el mecanismo "una herencia pero varias interfaces", podrían conseguirse resultados similares a los obtenidos con la herencia múltiple.

Ahora bien, del mismo modo que sucedía con la herencia múltiple, **puede darse el problema de la colisión de nombres al implementar dos interfaces que tengan un método con el mismo identificador**. En tal caso puede suceder lo siguiente:

- Si los dos métodos tienen diferentes parámetros no habrá problema aunque tengan el mismo nombre pues se realiza una sobrecarga de métodos.
- Si tienen igual firma pero un valor de retorno de diferente tipo, se produce un error de compilación (igual que sucede en la sobrecarga cuando la única diferencia es el tipo devuelto).
- Si los dos métodos son exactamente iguales en identificador, parámetros y tipo devuelto, entonces solamente se podrá implementar uno de los dos métodos. En realidad se trata de un solo método pues ambos tienen la misma interfaz (mismo identificador, mismos parámetros y mismo tipo devuelto).

La utilización de nombres idénticos en diferentes interfaces que pueden ser implementadas a la vez por una misma clase puede causar,

además del problema de la colisión de nombres, dificultades de legibilidad en el código, pudiendo dar lugar a confusiones. Si es posible intenta evitar que se produzcan este tipo de situaciones.

EJEMPLO 30: Localiza en la API de Java algún ejemplo de clase que implemente varias interfaces diferentes (puedes consultar la documentación de referencia de la API de Java).

Existen una gran cantidad de clases en la API de Java que implementan múltiples interfaces. Aquí tienes un par de ejemplos:

- La clase **javax.swing.JFrame**, implementa las interfaces WindowConstants, Accessible y RootPaneContainer.
- La clase **java.awt.Component**, implementa las interfaces ImageObserver, MenuContainer y Serializable.

5.6.4. MEJORAS A PARTIR DE JAVA 8.

MÉTODOS DEFAULT EN LAS INTERFACES

Antes de Java SE 8, los métodos de las interfaces solamente podían ser **public abstract**. Esto significa que especificaba qué operaciones debía realizar una clase implementadora, pero no cómo realizarlas.

En Java SE 8 las interfaces también pueden contener métodos **default public** que además de declarar el método proporcionan una implementación predeterminadas concreta donde se indican cómo deben realizarse las operaciones cuando una clase implementadora no sobrescriba los métodos. Si una clase implementa la interfaz, la clase también recibe las implementaciones **default** de esa interfaz (si las hay). Para declarar un método **default**, escribes la palabra clave **default** antes del tipo de valor de retorno del método y proporcionas una implementación concreta del mismo. Los métodos **default** son

automáticamente **public** (no hay que indicarlo).

EJEMPLO 31: Interface que declara un método del que define una implementación default y una clase que implementa la interface.

```
public interface Leible {     // representa un origen de datos
   public char leeChar();     // lee un char
   default public String leeLinea() {     // lee hasta salto
      StringBuilder linea = new StringBuilder();
      char ch = leeChar();
      while (ch != '\n') {
         linea.append(ch);
         ch = leeChar();
      }
      return linea.toString();
   }
}
```

Una clase concreta que implemente esta interfaz debe aportar una implementación para el método **leeChar()**. Hereda la definición de **leeLinea()**, aunque puede implementar una nueva definición si es necesario. La referencia a **leeChar()** en la definición es polimórfica. Una clase con main() que implementa la interfaz:

```
public class Estrella implements Leible {
   public char leeChar() {
      if (Math.random() > 0.02)
         return '*';
      else
         return '\n';
   }
   public static void main(String[] args) {
      Estrellas es = new Estrella();
      for (int i = 0 ; i < 10; i++ ) {
         String linea = es.leeLinea();
         System.out.println( linea );
      }
   }
}
```

Agregar métodos a las interfaces existentes

Antes de Java SE 8, si a una interface que ya ha sido implementada varias veces le añades más métodos, creas problemas en las clases implementadoras porque esos nuevos métodos no los tienen implementados. Recuerda que si la clase no implementa cada uno de los métodos de la interfaz, se tiene que declarar **abstract**.

Cualquier clase que implemente la interfaz original no se descompone cuando se añade un nuevo método **default**. La clase simplemente recibirá el nuevo método **default**. Cuando una clase implementa una interfaz de Java SE 8, la clase "firma un contrato" con el compilador que dice: "Declararé todos los métodos abstract especificados por la interfaz o declararé mi clase como abstract". La clase implementadora no tiene la obligación de sobrescribir los nuevos métodos **default** de la interfaz, aunque puede hacerlo si es necesario.

> **Nota**: *Los métodos **default** de Java SE 8 permiten evolucionar a las interfaces existentes al agregar nuevos métodos a esas interfaces sin descomponer el código que las usa.*

> **Nota**: *Antes de Java SE 8, se utilizaba por lo general una interfaz (en vez de una clase abstract) cuando no había detalles de implementación a heredar, es decir, no había campos ni implementaciones de métodos. Con los métodos default, puede declarar implementaciones de métodos comunes en las interfaces, lo que da más flexibilidad al diseñar clases.*

MÉTODOS STATIC DE UNA INTERFACE

Antes de Java SE 8, era común asociar con una interfaz a una clase que tuviera **métodos ayudantes static** para poder disponer de objetos que implementen la interfaz. Por ejemplo la clase **Collections**, que

contiene muchos **métodos ayudantes static** para trabajar con objetos que implementan las interfaces **Collection**, **List**, **Set**, ... Por ejemplo, el método **sort()** de Collections puede ordenar objetos de cualquier clase que implemente a la interfaz **List**. Con los métodos static de una interfaz, dichos métodos ayudantes pueden ahora declararse directamente en las interfaces, en vez de hacerlo en cada una de las clases ayudantes.

INTERFACES FUNCIONALES

A partir de Java SE 8, cualquier interfaz que contenga sólo un método **abstract** se conoce como interfaz funcional. Hay muchas de esas interfaces en las API de Java 7 y muchas nuevas en Java 8.

Algunas de las interfaces funcionales que usarás a menudo son:

- **ActionListener**: podrás implementar esta interfaz para definir un método que se invoca cuando el usuario hace clic en un botón.
- **Comparator**: para definir un método que puede comparar dos objetos de un tipo dado para determinar si el primer objeto es menor, igual o mayor que el segundo.
- **Runnable**: para definir una tarea que pueda ejecutarse en paralelo con otras partes de su programa.

Las interfaces funcionales se usan mucho con las nuevas capacidades funcionales de Java SE 8 (expresiones lambda). También se implementa frecuentemente una interfaz mediante la creación de lo que se conoce como **una clase interna anónima**, la cual implementa el o los métodos de la interfaz. En Java SE 8, las lambdas dan una notación abreviada para crear métodos anónimos que el compilador traduce de manera automática en clases internas anónimas.

EJEMPLO 32: Definir una interface funcional y usar una clase anónima

y para usarla.

```
public interface Saludable { public void saluda(); }

public class C1 {
    public static void main(String[] args) {
        Saludable s1 = new Saludable() {
              @Override
              public void saluda(){ System.out.println("Hola"); }
              }  // fin de la clase anónima
        s1.saluda();
    }
}
```

A la hora de contabilizar los métodos de una interfaz, no debes tener en cuenta ni los métodos que tengan una implementación por defecto ni los que coinciden con una firma de alguno de los que tenga Object, porque una clase ya tendría una implementación disponible para ellos.

5.7. POLIMORFISMO.

Es otro de los grandes pilares de la Programación Orientada a Objetos (junto con la ocultación y la herencia). Se trata nuevamente de otra forma más de establecer diferencias entre interfaz e implementación, es decir, entre el qué y el cómo.

La ocultación te ha permitido agrupar características (atributos) y comportamientos (métodos) dentro de una misma unidad (la clase), pudiendo darles una mayor o menor visibilidad, y permitiendo separar al máximo posible la interfaz de la implementación.

Por otro lado la herencia te ha proporcionado la posibilidad de tratar a los objetos como pertenecientes a una jerarquía de clases. Esta capacidad va a ser fundamental a la hora de poder manipular muchos posibles objetos de clases diferentes como si fueran de la misma clase (**polimorfismo**).

El polimorfismo te permite mejorar la organización y la legibilidad del código así como la posibilidad de desarrollar aplicaciones que sean más fáciles de ampliar a la hora de incorporar nuevas funcionalidades. Si la implementación y la utilización de las clases es lo suficientemente genérica y extensible será más sencillo poder volver a este código para incluir nuevos requerimientos.

CONCEPTO DE POLIMORFISMO

El polimorfismo consiste en la capacidad de poder utilizar una referencia a un objeto de una determinada clase como si fuera de otra clase (en concreto una subclase). Es una manera de decir que una clase podría tener varias (poli) formas (morfismo).

Un **método "polimórfico"** ofrece la posibilidad de ser distinguido (saber a qué clase pertenece) en tiempo de ejecución en lugar de en tiempo de compilación. Para poder hacer algo así es necesario utilizar métodos que pertenecen a una superclase aunque en cada subclase se implementan de una forma particular. En tiempo de compilación se invoca al método sin saber exactamente si será el de una subclase u otra (pues se está invocando al de la superclase). Sólo en tiempo de ejecución (una vez instanciada una u otra subclase) se conocerá realmente qué método (de qué subclase) es el que finalmente va a ser invocado.

Esta forma de trabajar te ofrece hasta cierto punto "desentenderte" del tipo de objeto específico (subclase) para centrarte en el tipo de objeto genérico (superclase). De este modo podrás manipular objetos "desconocidos" en tiempo de compilación y que sólo durante la ejecución del programa se sabrá exactamente de qué tipo de objeto (subclase) son.

El polimorfismo puede llevarse a cabo tanto con superclases (abstractas o no) como con interfaces.

Dada una superclase X, con un método m() y dos subclases A y B, que redefinen ese método m(), podrías declarar un objeto O de tipo X que durante la ejecución podrá ser de tipo A o de tipo B (algo desconocido en tiempo de compilación). Esto significa que al invocarse el método m() de X (superclase), se estará en realidad invocando al método m() de A o al método m() de B (alguna de sus subclases). Por ejemplo:

```
// Definición de las clases
class ClaseX { ... }
class ClaseA extends ClaseX { ... }
class ClaseB extends ClaseX { ... }
// Declaración de una referencia a un objeto de tipo ClaseX
ClaseX x; // Objeto de tipo ClaseX (superclase)
...
// Zona del programa donde se instancia un objeto de tipo ClaseA
// (subclase) y se le asigna a la referencia x.
// La variable x adquiere la forma de la subclase A.
x = new ClaseA();
...
// Otra zona del programa.
// Aquí se instancia un objeto de tipo ClaseB (subclase) y se le
// asigna a la referencia el objeto. La variable x adquiere la
// forma de la subclase ClaseB.
x = new ClaseB();
...
// Zona donde se utiliza el método m sin saber realmente qué
// subclase se está utilizando. Sólo se sabrá durante la ejecución
// del programa, pero no durante la compilación.
x.m() // Llamada al método m (sin saber si será el método m
// de ClaseA o de ClaseB o de ClaseX).
...
```

Imagina que estás trabajando con las clases `Persona`, `Alumno` y `Profesor` donde `Alumno` y `Profesor` son subclases de `Persona` y que en determinada zona del código podrías tener objetos, tanto de un tipo como de otro, pero eso sólo se sabrá según vaya discurriendo la ejecución del

programa. En algunos casos, es posible que un determinado objeto pudiera ser de la clase `Alumno` y en otros de la clase `Profesor`, pero en cualquier caso serán objetos de la clase `Persona`. Eso significa que la llamada a un método de la clase `Persona` (por ejemplo `mostrar()`) en realidad será en unos casos a un método (con el mismo nombre) de la clase `Alumno` y en otros, a un método (con el mismo nombre también) de la clase `Profesor`. Esto será posible hacerlo gracias a la ligadura dinámica.

5.5.1 LIGADURA ESTÁTICA Y DINÁMICA.

La conexión que tiene lugar durante una llamada a un método se denomina ligadura (conexión o vinculación que tiene lugar durante una llamada a un método para saber qué código debe ser ejecutado. Puede ser estática o dinámica), vinculación o enlace (en inglés binding).

Si esta vinculación se lleva a cabo durante el proceso de compilación, se le suele llamar **ligadura estática** (la vinculación que se produce en la llamada a un método con la clase a la que pertenece ese método se realiza en tiempo de compilación. Es decir, que antes de generar el código ejecutable se conoce exactamente el método (a qué clase y qué método pertenece). También conocido como **vinculación temprana**). En los lenguajes tradicionales, no orientados a objetos, ésta es la única forma de poder resolver la ligadura (en tiempo de compilación).

Sin embargo, en los lenguajes orientados a objetos existe otra posibilidad: la **ligadura dinámica** (la vinculación que se produce en la llamada a un método con la clase a la que pertenece ese método se realiza en tiempo de ejecución). Es decir, que al generar el código ejecutable no se conoce exactamente el método (a qué clase pertenece) que será llamado. Sólo se sabrá cuando el programa esté en ejecución. También conocida como **vinculación tardía**, enlace tardío o

late binding).

La ligadura dinámica hace posible que sea el tipo de objeto instanciado (obtenido mediante el constructor finalmente utilizado para crear el objeto) y no el tipo de la referencia (el tipo indicado en la declaración de la variable que apuntará al objeto) lo que determine qué versión del método va a ser invocada. El tipo de objeto al que apunta la variable de tipo referencia sólo podrá ser conocido durante la ejecución del programa y por eso **el polimorfismo necesita la ligadura dinámica**.

En el ejemplo anterior de la clase X y sus subclases A y B, la llamada al método m() sólo puede resolverse mediante ligadura dinámica, pues es imposible saber en tiempo de compilación si el método m() que debe ser invocado será el definido en la subclase A o el definido en la subclase B:

```
// Llamada al método m (sin saber si será el método de A o de B).
x.m() // Resuelta en tiempo de ejecución (ligadura dinámica)
```

EJEMPLO 33: Imagina una clase que representa un instrumento musical genérico (`Instrumento`) y dos subclases que representen tipos de instrumentos específicos (por ejemplo `Flauta` y `Piano`). Todas las clases tendrán un método `tocarNota()`, que será específico para cada subclase. Haz un pequeño programa de ejemplo en Java que utilice el polimorfismo (referencias a la superclase que se convierten en instancias específicas de subclases) y la ligadura dinámica (llamadas a un método que aún no están resueltas en tiempo de compilación) con estas clases que representan instrumentos musicales. Puedes implementar el método `tocarNota()` mediante la escritura de un mensaje en pantalla.

La clase `Instrumento` podría tener un único método (tocarNota):

```
public class Instrumento {
    public void tocarNota (String nota) {
       System.out.printf("Instrumento: toco nota %s.\n", nota);
    }
}
```

En el caso de las clases `Piano` y `Flauta` puede ser similar, heredando de `Instrumento` y redefiniendo el método `tocarNota()`:

```
public class Flauta extends Instrumento {
   @Override
   public void tocarNota(String nota) {
      System.out.printf("Flauta: toco nota %s.\n", nota);
   }
}
public class Piano extends Instrumento {
   @Override
   public void tocarNota (String nota) {
      System.out.printf("Piano: toco nota %s.\n", nota);
   }
}
```

A la hora de declarar una referencia a un objeto de tipo `Instrumento`, utilizamos la superclase (`Instrumento`):

```
Instrumento i1; // Ejemplo de objeto polimórfico
```

Sin embargo, a la hora de instanciar el objeto, utilizamos el constructor de alguna de sus subclases (Piano, Flauta, etc.):

```
if (Math.random() < 0.33) {
   // En este caso va adquirir forma de Piano
   i1 = new Piano();
}
else if (Math.random() < 0.33) {
   // En este caso va adquirir forma de Flauta
   i1 = new Flauta();
} else {
   i1 = new Instrumento();
}
```

Finalmente, a la hora de invocar el método `tocarNota()`, no se sabe a qué

versión (de qué subclase) de `tocarNota()` se está llamando, pues depende del tipo de objeto (subclase) que se haya instanciado. Se estará utilizando por tanto la ligadura dinámica:

```
// Interpretamos una nota con el objeto i1
// No sabemos si se ejecutará el método de Piano o de Flauta
// (dependerá de la ejecución)
i1.tocarNota("do"); // Ejemplo de ligadura dinámica
```

LIMITACIONES DEL POLIFORMISMO

Como has podido comprobar, el polimorfismo se basa en la utilización de referencias de un tipo más "amplio" (superclases) que los objetos a los que luego realmente van a ejecutarse (subclases). Ahora bien, existe una importante restricción en el uso de esta capacidad, pues el tipo de dato usado en la referencia limita cuáles son los métodos que se pueden utilizar y los atributos a los que se pueden acceder.

No se puede acceder a miembros específicos de una subclase a través de una referencia de una superclase. Sólo se pueden utilizar los miembros declarados en la superclase, aunque la definición que finalmente se utilice en su ejecución sea la de la subclase.

En el ejemplo de los instrumentos no teníamos problema porque las 3 clases definían el mismo método, `tocarNota()` estaba en la superclase `Instrumento`. Pero si por ejemplo `Flauta` tiene el método:

```
public void cambiaBoquilla() {
     System.out.println("Flauta: cambio  boquilla");
}
```

Que no tiene la clase `Instrumento`, utilizar una referencia de tipo `Instrumento` para usar algo exclusivo de la subclase sería un error. Solamente podrías usar elementos que tenga la clase `Instrumento` en común con las subclases;

```
Instrumento x = new Flauta();
x.tocarNota("re");    // perfecto, Instrumento tiene tocarNota()
                      // se ejecuta el de Flauta (lig. Dinámica)
x.cambiaBoquilla();   // Error, Instrumento no tiene ese elemento
```

En el ejemplo de las clases `Persona`, `Profesor` y `Alumno`, el polimorfismo nos permitiría declarar variables de tipo `Persona` y más tarde hacer con ellas referencia a objetos de tipo `Profesor` o `Alumno`, pero no deberíamos intentar acceder con esa variable a métodos o datos que sean específicos de la clase `Profesor` o de la clase `Alumno`, tan solo a métodos que sabemos que van a existir seguro en ambos tipos de objetos (métodos de la superclase Persona).

EJEMPLO 34: Haz un pequeño programa en Java en el que se declare una variable de tipo `Persona`, se pidan algunos datos sobre esa persona (nombre, apellidos y si es alumno o si es profesor), y se muestren nuevamente esos datos en pantalla, teniendo en cuenta que esa variable no puede ser instanciada como un objeto de tipo `Persona` (es una clase abstracta) y que tendrás que instanciarla como `Alumno` o como `Profesor`. Recuerda que para poder recuperar sus datos necesitarás hacer uso de la ligadura dinámica y que tan solo deberías acceder a métodos que sean de la superclase.

Si tuviéramos diferentes variables referencia a objetos de las clases `Alumno` y `Profesor` tendrías algo así:

```
Alumno obj1;
Profesor obj2;
...
// Si se dan ciertas condiciones el objeto será de tipo Alumno y
// lo tendrás en obj1
System.out.printf ("Nombre: %s\n", obj1.getNombre());
// Si se dan otras condiciones el objeto será de tipo Profesor y
// lo tendrás en obj2
System.out.printf ("Nombre: %s\n", obj2.getNombre());
```

Pero si puedes tratar de una manera más genérica la situación:

```
Persona obj;
// Si se dan ciertas condiciones el objeto será de tipo Alumno y
// por tanto lo instanciarás como tal
obj = new Alumno(<parámetros>);
// Si se dan otras condiciones el objeto será de tipo Profesor y
// por tanto lo instanciarás como tal
obj = new Profesor(<parámetros>);
```

De esta manera la variable obj podría contener una referencia a un objeto de la superclase `Persona` de subclase `Alumno` o bien de subclase `Profesor` (polimorfismo). Esto significa que independientemente del tipo de subclase que sea (Alumno o `Profesor`), podrás invocar a métodos de la superclase `Persona` y durante la ejecución se resolverán como métodos de alguna de sus subclases:

```
// En tiempo de compilación no se sabrá de qué subclase de Persona
// será obj. Habrá que esperar la ejecución para que el entorno lo
// sepa e invoque al método adecuado.
System.out.printf("Contenido del objeto usuario: %s\n",
                  stringContenidoUsuario() );
```

Por último recuerda que debes de proporcionar constructores a las subclases `Alumno` y `Profesor` que sean "compatibles" con algunos de los constructores de la superclase `Persona`, pues al llamar a un constructor de una subclase, su formato debe coincidir con el de algún constructor de la superclase (como debe suceder en general con cualquier método que sea invocado utilizando la ligadura dinámica).

EJERCICIO 14: Crea un array de 5 elementos de la clase Vehiculo.

a) Con un bucle, rellena el array con datos que pides al usuario dándole a elegir si quiere añadir un Velero o una Bicileta.

b) Muestra el array creando un StringBuffer que contenga un contador del vehículo `"#n"` el nombre de su clase (`Object` tiene

`getClass()`) y sus datos en una línea. Y cuenta usando el operador `instanceOf` que diga cuantos veleros y bicicletas hay.

INTERFACES Y POLIMORFISMO

Es posible también aprovechar el polimorfismo mediante el uso de interfaces además de mediante una clase padre común. Un objeto puede tener una referencia cuyo tipo sea una interfaz, pero para que el compilador te lo permita, la clase cuyo constructor se utilice para crear el objeto debe implementar esa interfaz (bien por si misma o bien porque la implemente alguna superclase suya).

Un objeto cuya referencia sea de tipo interfaz sólo puede utilizar aquellos métodos definidos en la interfaz, es decir, que no podrán utilizarse los atributos y métodos específicos de su clase, tan solo los de la interfaz.

Las referencias de tipo interfaz permiten manipular con el mismo código objetos que pertenecen a clases diferentes (pero que todas ellas implementan la misma interfaz). De este modo podrías hacer referencia a diferentes objetos que no tienen ninguna relación jerárquica entre sí utilizando la misma variable (referencia a la interfaz). Lo único que los distintos objetos tendrían en común es que implementan la misma interfaz. En este caso sólo podrás llamar a los métodos de la interfaz y no a los específicos de las clases.

Por ejemplo, si tenías una variable de tipo referencia a la interfaz `Arrancable`, podrías instanciar objetos de tipo `Coche` o `Motosierra` y asignarlos a esa referencia (teniendo en cuenta que ambas clases no tienen una relación de herencia). Sin embargo, tan solo podrás usar en ambos casos los métodos y los atributos de la interfaz `Arrancable` (por ejemplo arrancar) y no los de `Coche` o los de `Motosierra` (sólo los

genéricos, nunca los específicos).

En el caso de las clases `Persona`, `Alumno` y `Profesor`, podrías declarar, por ejemplo, variables del tipo `Imprimible`:

```
Imprimible obj; // Imprimible es una interfaz y no una clase
```

Con este tipo de referencia podrías luego apuntar a objetos tanto de tipo `Profesor` como de tipo `Alumno`, o de tipo `BoletinNotas`, o de cualquier otro tipo siempre que implementan la interfaz `Imprimible`:

```
// En algunas circunstancias podría suceder esto:
obj= new Alumno(nombre, apellidos, fecha, grupo, nota);
// Polimorfismo con interfaces
...
// En otras circunstancias podría suceder esto:
obj= new Profesor(nombre, apellidos, fecha, especialidad,
salario);
// Polimorfismo con interfaces ... Y más adelante hacer uso de la
ligadura dinámica:
// Llamadas sólo a métodos de la interfaz
String contenido;
contenido= obj.contenidoString(); // Ligadura dinámica
```

EJERCICIO 15: Haz estos apartados para usar polimorfismo utilizando interfaces en vez de clases:

a) Define la interface Seleccionable que describa el comportamiento de un deportista que pueda ser seleccionado para jugar en la selección española con los métodos: `void concentrarse()`, `void viajar()`, `void entrenar()` y `void jugarPartido()`.

b) Haz la clase abstracta `MiembroSeleccionFutbol` que implemente la interfaz `Seleccionable` además de tener los datos `nombre` y `edad` tendrá un constructor vacío, otro con todos sus datos, getters y setters y una implementación de los métodos de la interfaz que muestre por consola mensajes del tipo `"<método> clase`

Abstracta".

c) Crea la clase `Futbolista` que extienda a la clase abstracta `MiembroSeleccionFutbol` que añada los datos `dorsal` (nº de camiseta) y `demarcación` (en qué puesto juega: defensa, lateral, medio centro, portero, etc.). Añade los constructores, getters y setters necesarios y sobreescribe los métodos `entrenar()` y `jugarPartido()` y añade el método `entrevista()` donde se escriben mensajes del tipo `"<metodo> (clase)"`.

d) Crea la clase `Entrenador` que extienda a la clase abstracta `MiembroSeleccionFutbol` que añada los datos `federacion` y añade los constructores, getters y setters necesarios y sobreescribe los métodos `entrenar()` y `jugarPartido()` y añade el método `planificaEntrenamiento()` donde se escriben mensajes del tipo `"Realiza <metodo> (clase)"`.

e) Crea la clase `TestSeleccion` y creas un método main donde defines un array de 15 Seleccionables, crea un entrenador y dos jugadores de fútbol y los añades al array. En un bucle haz que todos se concentren. En otro bucle haz que todos viajen. En otro bucle haz que todos entrenen. En otro bucle haz que todos juguen. Y en otro bucle, cuando sean jugadores que concedan una entrevista.

CONVERSIÓN DE OBJETOS

En principio no se puede acceder a los miembros específicos de una subclase a través de una referencia a una superclase. Si deseas tener acceso a todos los métodos y atributos específicos del objeto subclase tendrás que realizar una **conversión explícita (casting)** que convierta la referencia más general (superclase) en la del tipo específico del objeto (subclase).

Para que puedas realizar conversiones entre distintas clases es obligatorio que exista una relación de herencia entre ellas (una debe ser clase derivada de la otra) o de implementación (la clase implementa el comportamiento de la interface). **Se realizará una conversión implícita o automática de subclase a superclase siempre que sea necesario**, pues un objeto de tipo subclase siempre contendrá toda la información necesaria para ser considerado un objeto de la superclase.

Ahora bien, la conversión en sentido contrario (de superclase a subclase) debe hacerse de forma explícita y según el caso podría dar lugar a errores por falta de información (atributos) o de métodos. En tales casos se produce una excepción de tipo **ClassCastException**. Por ejemplo, imagina que tienes una clase A y una clase B, subclase de A:

```
class ClaseA { public int atrib1;  }
class ClaseB extends ClaseA { public int atrib2; }
```

A continuación declaras una variable referencia a la clase A (superclase) pero sin embargo le asignas una referencia a un objeto de la clase B (subclase) haciendo uso del polimorfismo:

```
A obj;         // Referencia a objetos de la clase A
obj= new B(); // Referencia a objetos clase A, pero apunta
               // realmente a objeto clase B (polimorfismo)
```

El objeto que acabas de crear como instancia de la clase B (subclase de A) contiene más información que la que la referencia obj te permite en principio acceder sin que el compilador genere un error (pues es de clase A). En concreto los objetos de la clase B disponen de atrib1 y atrib2, mientras que los objetos de la clase A sólo de atrib1. Para acceder a esa información adicional de la clase especializada (atrib2) tendrás que realizar una conversión explícita (casting):

```
// Casting del tipo A al tipo B (funcionará bien porque el objeto
// es realmente del tipo B)
```

```
System.out.printf("obj.atrib2=%d\n", ( (B)obj ).atrib2);
```

Sin embargo si se hubiera tratado de una instancia de la clase A y hubieras intentado acceder al miembro atrib2, se habría producido una excepción de tipo ClassCastException:

```
A obj; // Referencia a objetos de la clase A
obj= new A(); // Referencia a objetos de la clase A, y apunta
realmente a un objeto de la clase A
// Casting del tipo A al tipo B (puede dar problemas porque el
// objeto es realmente del tipo A):
// Funciona (la clase A tiene atrib1)
System.out.printf("obj.atrib2=%d\n", ((B) obj).atrib1);
// ¡Error en ejecución! (la clase A no tiene atrib2). Producirá
// una ClassCastException.
```

Una forma de hacer el casting seguro es comprobar si efectivamente una referencia se puede considerar que es una instancia de algo antes de intentar convertirlo a eso algo.

EJERCICIO 16: En un mismo fichero define la clase ejecutable `Cuento` y la interface `Besable` que declara el método `besar()`. Declara también las clases no públicas `Principe` que es superclase de `PrincipeAzul` y ambos implementan la interface `Besable` anterior. Tenemos otra clase que también implementa la interface `Besable` que es la clase `Sapo`.
En el main de `Cuento` encierra en un `try-catch` el siguiente código y en caso de error muestra la excepción generada: declaras una referencia de tipo `Sapo` y le asignas una instancia de `Sapo`. Luego declara una referencia a `Principe` y le asignas el sapo creado con anterioridad convertido a `PrincipeAzul`. ¿Qué ocurre?

Antes de hacer un casting de clase, puedes asegurarte de que se puede realizar usando el operador **`instanceof`**. Sintaxis:

```
referencia instanceof Tipo
```

Que devuelve true si la referencia es una instancia de la clase Tipo, en cuyo caso un casting puede hacerse sin problemas, o devuelve false si no lo es y un casting a Tipo fallará. Ejemplo:

```
Integer s = 5;
String s1;
if( s instanceof String )
    s1 = (String)s;
else
    System.out.println("No se puede hacer el casting");
```

EJERCICIO 17: Añade al código del ejercicio anterior después del try-catch las sentencias que permitan decidir si es buena idea convertir el sapo en un príncipe o no, usando el operador `instanceof`.

```
Sapo s = new Sapo();
Principe pa = (PrincipeAzul)s; // Peligroso, s es convertible?
```

5.8. EXPRESIONES LAMBDA.

5.8.1. INTRO A LA PROGRAMACIÓN FUNCIONAL.

Vamos a comenzar por el final aunque no sepamos lo que es ¿Qué nos aporta la programación funcional? Bueno, si pensamos porqué han evolucionado los lenguajes de programación a lo largo de la historia, podemos llegar a la conclusión de que la intención ha sido aumentar la productividad, mejorar la facilidad de hacer, mantener y entender los programas. ¿A qué se deben la mayoría de fallos que tienen los programas? Normalmente a efectos colaterales porque usan elementos mutables. Esta es una de las cosas que quiere aportar esta nueva forma de programar.

La programación funcional propone resolver problemas basándose en el cálculo de funciones matemáticas. ¿Qué es una función matemática? Vemos un ejemplo, la función doble:

```
doble(x) : R ------> R      (definición: nombre, que usa y que devuelve)
           x ------> 2 * x  (implementación: La ley o expresión)
```

Una expresión o ley que relaciona un conjunto de entradas (origen) con un conjunto de salidas (imagen). La salida depende solo de su entrada (determinista). Lo que les da potencia como sistema de cálculo es que las funciones se pueden componer:

```
(f o g) (x) = f( g(x) )   // función f compuesta con función g
```

Se basa en la teoría matemática del **Lambda-Cálculo** desarrollada por Alonzo Chruch en la década de 1930 que define un sistema de expresiones que forman un modelo de cálculo universal.
Los paradigmas de programación se suelen dividir en **declarativos** (se dice lo que se quiere obtener) e **imperativos** (se indican las sentencias a realizar para conseguirlo). Nosotros hemos trabajado durante este curso con:

- **La programación estructurada y procedural:** un tipo de programación imperativa en el que hay un flujo de sentencias se van ejecutando leyendo y modificando el estado (los datos) hasta que se alcanza cierto estado final o acaba el flujo de sentencias. La programación modular es simplemente tener varios de estos artefactos en vez de solo uno.
- **La orientación a objetos:** crea unidades de más alto nivel que engloban los datos y las sentencias y extiende algunos conceptos de la programación procedural.

Pero aún hay más:

- **El paradigma declarativo:** no indicas las sentencias a realizar sino qué quieres conseguir. SQL es un ejemplo.
- **El paradigma funcional**: un subconjunto de lenguaje declarativo.

Cualquier lenguaje de programación actual que aspira a ser de propósito general (que se pueda usar para muchas cosas), incorpora más de un paradigma.

PRINCIPIOS Y CONCEPTOS DE PROGRAMACIÓN FUNCIONAL

FUNCIONES DE PRIMERA CLASE Y DE ALTO NIVEL

Un lenguaje de programación se dice que **soporta funciones de primera clase** si trata a sus funciones como ciudadanos de primera clase, es decir, les permite hacer lo que hagan otras cosas del lenguaje. Por ejemplo: asignar a una variable una función, pasar como argumento a una función otra como prámetro, que el resultado de ejecutar una función pueda ser otra función, etc.

Si un lenguaje tiene esto, permite definir **funciones de alto nivel** que son capaces de recibir funciones como parámetros y devolver una función como resultado o el currying.

En Java era posible pasar funciones como parámetros usando interfaces funcionales o clases internas anónimas. Por ejemplo queremos ordenar un array o una colección de objetos complejos usando sort() y le pasamos dos parámetros: numeros es la colección y el segundo parámetro es como queremos comparar esos elementos (una función, un código) que indicamos con una interface funcional que implementa una clase anónima.

```
Collections.sort(numeros,
                 new Comparator<Integer>() {
                     @Override
                     public int compare(Integer n1, Integer n2) {
                         return n1.compareTo(n2);
                     }
                 }
                );
```

Usar programación funcional con esta tediosidad es insoportable para un ser humano normal, por eso en Java aparecen mejoras a partir del JDK 8: expresiones lambda, referencias a métodos e interfaces funcionales predefinidas. La misma sentencia de antes con una expresión lambda:

```
Collections.sort( numeros, (n1, n2) -> n1.compareTo(n2) );
```

Pero el cambio es sintáctico, Java envuelve las expresiones lambda en el objeto de una clase anónima que implementa una interface funcional. Porque en Java, el único ciudadano de primera clase es un objeto :-O

FUNCIONES PURAS

Devuelven un resultado a patir de sus parámetros y no tiene otro efecto colateral (no afecta a nada del mundo exterior). Por tanto, un setter de un objeto no sería una función pura. Ejemplo: imagina que queremos calcular la suma de todos los números que hemos ordenado antes

```
Integer sum(List<Integer> n){
   return n.stream().collect(Collectors.summingInt(Integer::intValue));
}
```

IMMUTABILIDAD

Es otra propiedad apreciada en programación funcional. Ya sabes que consiste en que una entidad no debe modificar su estado después de crearse. Ya sabes que en Java es posible conseguir la immutabiliad de un objeto, pero no de manera automática, hay que currárselo.

TRANSPARENCIA REFERENCIAL

Una expresión es referencialmente transparente si reemplezarla con su valor no tiene consecuencias para el programa. Esto permite técnicas potentes como las funciones de alto nivel y la evaluación

perezosa o tardía (lazy). Un ejemplo: Comprobemos si esta clase de Java tiene transparecia referencial.

```
public class SimpleDato {
    private String dato;
    public String getDato() {
        System.out.println("Llamada a SimpleDato.getDato()");
        return dato;
    }
    public SimpleDato setDato(String d) {
        System.out.println("llamada a SimpleDato.setDato()");
        this.dato = d;
        return this;
    }
}
```

Si lo probamos en estas sentencias:

```
String d = new SimpleDato().setDato("Jose").getDato();
System.out.println( new SimpleDato().setDato("Jose").getDato() );
System.out.println(d);
System.out.println("Jose");
```

Las tres impresiones son semánticamente equivalentes pero no referencialmente transparentes. La primera llamada produce un efecto colateral (consultar y modificar el dato también genera salida). Luego no es equivalente a la tercera llamada.
La segunda llamada tampoco porque al ser mutable, el setter no hace fácil que pueda sustituirse por un valor sin riesgo a equivocarnos. Podría cambiar y no ser "Jose" la cadena que imprime.

Así que básicamente, para tener transparencia referencial debemos tener funciones puras e immutables. Esto nos permite generar código libre-de-contexto, es decir, puede ejecutarse en cualquier orden y contexto que funcionará. Y esto abre un gran abanico de posibilidades para optimizaciones y ejecuciones en paralelo.

TÉCNICAS USADAS EN PROGRAMAIÓN FUNCIONAL

COMPOSICIÓN DE FUNCIONES

Permite generar funciones complejas combinando funciones simples. En Java podemos hacerlo partiendo de interfaces funcionales y de manera más cómoda usando las que nos proporciona en el paquete `java.util.function`. Muchas de estas dan soporte a la composición en forma de métodos default o static definidos en las propias interfaces funcionales.

Por ejemplo: **Function** es una interface funcional que acepta un parámetro y produce un resultado.

```
Function<tipo_entrada, tipo_salida> nombre = (entrada) -> salida;
```

Además cuenta con dos métodos con implementación por defecto que se llaman **compose()** y **andThen()**, que nos ayudan a componer funciones (recuerda que devuelven otra función):

```
Function<Double, Double> log = (v) -> Math.log(v);
Function<Double, Double> raiz = (v) -> Math.sqrt(v);
Function<Double, Double> logYRaiz = raiz.compose(log);
System.out.println( String.valueOf( logYraiz.apply(3.14) ) );
Function<Double, Double> raizYLog = raiz.andThen(log);
System.out.println( String.valueOf(raizYLog.apply(3.14)) );
```

Es decir, explicándolo de manera funcional y no funcional:

```
public double raiz(double entrada){ return Math.sqrt(entrada); }
public double log(double entrada){ return Math.sqrt(entrada); }
public double raizComposeLog(double entrada){
    return Math.sqrt( Math.log(entrada) );
}
public double raizAndThenLog(double entrada){
    return Math.log( Math.sqrt(entrada) );
}
```

Hay otros métodos que componen también como: **and**, **or**, **negate** en la

interface **Predicate**. También hay versiones con dos argumentos: **BiFunction** y **BiPredicate**.

MÓNADAS

Un monada es una abstracción que permite construir programas genéricos. Te permite envolver un valor, aplicarle una serie de transformaciones y deolver el valor con todas las transformaciones aplicadas. Hay 3 leyes que un monada necesita seguir: identidad izquierda, identidad derecha y asociatividad.
En Java, hay unos cuantos usados frecuentemente como **Optional, CompletableFuture** y **Stream**: y también podemos definir los nuestros propios. Ejemplo:

```
Optional.of(2).flatMap(f->Optional.of(3).flatMap(s->Optional.of(f + s)))
```

¿Porqué decimos que Optional es un monada? Nos permite envolver un valor usando el método of().

CURRYING

Es un truco matemático que convierte una función con varios parámetros en una secuencia de funciones con un único parámetro. ¿Qué tiene esto de utilidad? Nos da un mecanismo de composición muy potente porque en vez de llamar a una función con todos sus parámetros (lo que obliga a tenerlos todos para que se pueda realizar la llamada) se pueden llamar por separado los que ya tengamos.

```
f(a, b, c) ----> f().a().b().c()
```

En lenguajes como Haskel todas las funciones son curryed, en Java no es tan directo, es más manual. Ejemplo:

```
Function<Double, Function<Double, Double>> peso =
                                    masa -> gravedad -> masa * gravedad;
Function<Double, Double> pesoTierra = peso.apply(9.81);
System.out.println("Mi peso en la Tierra: " + pesoTierra.apply(60.0) );
```

```
Function<Double, Double> pesoMarte = peso.apply(3.75);
System.out.println("Mi peso en Marte: " + pesoMarte.apply(60.0) );
```

Aunque nuestra masa es la misma, la gravedad cambia en cada planeta, así que si dividimos manualmente el trabajo en dos etapas que podemos realizar de manera independiente. Podemos aplicar parcialmente la función pasando la gravedad. Para tener currying el languaje debe ofrecer: **expresiones lambda** y **cerraduras**. Las expreciones lambda en Java van bien, las cerraduras tienen limitaciones, pero no ahondamos.

RECURSIÓN

Nos permite resolver un problema acercándonos en pequeños pasos hasta llegar a una solución base. Además tiene la característica de que elimina efectos colaterales. Ejemplo:

```
Integer factorial(Integer n){ return (n<=1)? 1:n*factorial(n-1); }
```

Una desventaja de estas soluciones es que en cada estapa hay que guardar el estado hasta llegar al caso base y luego retroceder completando las soluciones. Podemos solucionarlo usando un acumulador:

```
Integer factorial(Integer n, Integer a) {
    return (n <= 1) ? a: factorial(n-1, a * n);
}
```

5.8.2 EXPRESIONES LAMBDA.

La sintaxis de las clases anónimas es incómoda. Muchas veces implementan una clase que cumple con una interfaz con solo un método. Java 8 introduce una nueva sintaxis que puede utilizarse en lugar de las clases anónimas en esa circunstancia: **una expresión lambda**. Por ejemplo, una interface como `Drawable`:

```
import java.awt.Graphics;
public interface Drawable {  public void draw(Graphics g);  }
```

Esta interface es funcional porque solamente obliga a implementar un método abstracto a quien se comprometa a implementarla. El método `dibuja(Graphics g, Drawable d){ ... }` lo que hará será dibujar un objeto que implemente draw en el contexto gráfico g:

```
public void dibuja(Graphics g, Drawable d) {  d.draw(g);  }
```

Podemos hacer una llamada para dibujar un óvalo haciéndonos una clase que implemente la interface en su fichero, pero es demasiado fauena para simplemente dibujar un óvalo ¿no crees? Mejor utilizamos una clase anónima que implemente la interface:

```
dibuja( unG,
        new Drawable(){
            @Override
            public void draw(Graphics g) {
                g.drawOval(10,10,100,100);
            }
        }
    );
```

Esta es la manera más sencilla hasta la aparición de las expresiones lambda. Pero usando estas, la sentencia equivalente a la llamada anterior:

```
dibuja( unG, g -> g.drawOval(10,10,100,100) )
```

La expresión lambda es **`g -> g.drawOval(10,10,100,100)`**. Su significado es: "el método a ejecutar tiene un parámetro g y ejecuta el código **`g.drawOval(10,10,100,100)`**". El sistema deduce que g es de tipo Graphics porque espera un parámetro de este tipo en la interfaz `Drawable` como parámetro actual, y como el único método en la interfaz `Drawable` tiene un parámetro de tipo **`Graphics`** pues blanco y en botella.

Las expresiones Lambda solamente pueden utilizarse en situaciones donde este tipo de inferencias se pueden realizar. La sintaxis general

de una expresión lambda es:

parámetros_formales -> cuerpo_del_método

donde **cuerpo_del_método** puede ser una expresión sencilla, una llamada a subrutina o un bloque de sentencias: `{sentencias}`. Hay que tener en cuenta que:

- Cuando el cuerpo es una expresión sencilla o una llamada a función, el valor de la expresión se usa como valor de retorno del método que se ha definido.
- La lista de parámetros no debe especificar los tipos de datos, aunque puede hacerlo.
- Los paréntesis alrededor de la lista de parámetros son opcionales si solo hay un parámetro y no se indica tipo. De lo contrario, o si no hay parámetros, los paréntesis son obligatorios.
- Para un método sin parámetros, la lista de parámetros es solo un par de paréntesis vacíos.

Puedes imaginar que una expresión lambda es una función anónima:

- Es anónima porque no tiene un nombre explícito como lo tiene un método o una clase o una variable.
- Es una función porque tiene una lista de parámetros, un cuerpo, un valor de retorno y una posible lista de excepciones que puede lanzar.
- Sin embargo, al contrario que una función normal, no tiene que pertenecer a una clase.
- Una expresión lambda puede pasarse como argumento a un método y puede ser devuelta como resultado (podemos enviar y devolver código!!).

La expresión lambda puede acceder a los datos del código donde se

define, pero ni ella puede modificarlos, ni el código tampoco (se declaran o se usan como datos final). Lo mismo que les ocurría a las funciones internas locales y anónimas.

Otra cosa a tener en cuenta es que puedes usar **this** dentro de una función lambda, pero representa al objeto donde se define la expresión lambda.

Sintaxis de las Expresiones Lambda:
Las expresiones lambda tienen 3 partes:

- Una lista de parámetros entre paréntesis (opcionales si solo tiene uno) y obligatorios si no hay parámetros. Ejemplo: **(File f)**
- Una flecha de dos caracteres: **->**
- Un cuerpo que es una sola sentencia o un bloque. Ej: `f.getName().endsWith(".xml")`. Por tanto podemos verlas:

```
(parámetros) -> expresión;
(parámetros) -> { varias_sentencias; }
```

Se puede omitir el tipo de los parámetros, puede inferirlo el compilador. Pero si indicas uno, debes indicarlos todos.

Las expresiones lambda son código, pero pueden almacenarse en variables, por ejemplo en una variable de una **interfaz funcional** (una interfaz de la que solo haya que implementar un método) y la expresión lambda equivale a ese método. Por ejemplo las interfaces siguientes:

```
@FunctionalInterface
public interface Runnable { void run(); }

@FunctionalInterface
public interface FileFilter { boolean accept(File pathname); }
```

El truco para deducir sus expresiones lambda equivalentes debemos

fijarnos en el método que debe implementar. Los parámetros que tenga son los que debemos pasar en la expresión lambda, y el código que queramos ejecutar es lo que debemos ejecutar en el cuerpo:

```
Runnable r = () -> System.out.println("Hola");
    // () porque run() no tiene parámetros
    // System.out.println() sirve porque no devuelve nada (void)
FileFilter esXml = (File f) -> f.getName().endsWith(".xml");
    // (File f) porque accept tiene ese tipo de parámetro
    // Si devuelve boolean debo ejecutar algo que devuelva boolean
```

Ejemplos de expresiones lambda:

```
() -> System.out.println("Hola mundo")
g -> { g.setColor(Color.RED); g.drawRect(10,10,100,100); }
(a, b) -> a + b;
(int n) -> {
                while (n > 0) {
                   System.out.println(n); n = n/2;
                }
            }; // lambda
```

Una expresión lambda puede interpretarse como una manera concisa (corta) de expresar una función anónima. El código que desarrollas será más rápido de escribir, corto y sencillo de entender. Ejemplo:

```
// Sin expresiones lambda, con una clase anónima
Comparator<Manzana> porPeso = new Comparator<Manzana>() {
    public int compare(Manzana m1, Manzana m2){
        return m1.getPeso().compareTo( m2.getPeso() );
    }
};

// Con expresiones lambda
Comparator<Manzana> porPeso =
       (Manzana m1, Manzana m2) -> m1.porPeso().compareTo( m2.porPeso() );
```

Una expresión lambda está formada por parámetros (en rojo), una flecha (en verde) y el cuerpo (en amarillo). La sintaxis básica es:

```
(parámetros) -> expresión
```

```
(parámetros) -> { sentencias... }
```

EJEMPLO 35: Más ejemplos de expresiones lambda:

```
(String s) -> s.length()          // devuelve la longitud del string s
(Manzana m) -> m.getPeso() > 150  // devuelve un booleano
(int x, int y) -> { System.out.printll("Resultado:";)
                    System.out.println("  " + (x + y) );
                  }  // 2 parámetros enteros y no devuelve nada
() -> 42                                  // Sin parámetros devuelve entero
(List<String> lista) -> lista.isEmpty() // devuelve booleano
() -> new Manzana(10)                     // Crea objeto y lo devuelve
(Manzana m) -> { System.out.println( m.getPeso() ); }
```

EJERCICIO 18: ¿Cuales de estas expresiones lambda no son correctas y porqué?

```
1. () -> {}
2. () -> "Raúl"
3. () -> { return "Mario"; }
4. (Integer i) -> return "Alfonso " + i;
5. (String s) -> { "Iron Man"; }
```

¿DÓNDE Y CÓMO USARLAS?

INTERFACES FUNCIONALES

Una interface funcional es aquella que solo tiene un método abstracto (no cuentan ni los que tienen una implementación por defecto, ni los que sobreescriben a métodos de Object). Por ejemplo:

```
public interface Comparator<T>{ int compare(T o1, T o2); }  // java.util

public interface Runnable{  void run(); }                    // java.lang

public interface ActionListener extends EventListener{   // java.awt.event
    void actionperformed(ActionEvent e);
}

public interface Callable<V>{  V call(); }        // java.util.concurrent

public interface privilegedAction<V>{ T run(); }  // java.security
```

EJERCICIO 19: ¿Cuales de estas tres interfaces son funcionales?

```
public interface Adder{ int add(int a, int b); }
public interface SmartAdder extends Adder{ int add(double a, double b); }
public interface Nada{ }
```

¿Qué pueden hacer las funciones lambda con las interfaces funcionales? Bueno, pueden implementar su método abstracto. También lo puedes hacer con clases internas anónimas, pero eso sí, escribiendo más código.

EJEMPLO 36: Implementar interfaces funcionales con lambdas.

```
Runnable r1= () -> System.out.println("Hola mundo");   // usando lambda

Runnable r2= new Runnable() {              // usando clase interna anónima
    public void run() { System.out.println("Hola mundo"); }
};

public static void process(Runable r) { r.run(); }

process(r1);  // Imprime Hola mundo 1
process(r2);  // Umorime Hola mundo 2
process( () -> System.out.println("Hola mundo") ); // lambda directa
```

DESCRIPTORES DE FUNCIONES

Igual que el método abstracto de una interfaz funcional tiene su firma, la función lambda que lo implementa debe tener una estructura compatible, es lo que se llama un descriptor de función.

EJEMPLO 37: Firma del método y descriptor de función lambda.
Por ejemplo, el método run() de la interface funcional Runnable tiene la firma: se llama run y no acepta parámetros. Además devuelve void. Por tanto, una función lambda que la implemente debe tener el descriptor: "() -> void" como por ejemplo "() -> {}".

Interfaz funcional	Descriptor de función	Especialización de primitivas
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate

Interfaz funcional	Descriptor de función	Especialización de primitivas
`Consumer<T>`	`T -> void`	`IntConsumer, LongConsumer, DoubleConsumer`
`Function<T, R>`	`T -> R`	`IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>`
`Supplier<T>`	`() -> T`	`BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier`
`UnaryOperator<T>`	`T -> T`	`IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator`
`BinaryOperator<T>`	`(T, T) -> T`	`IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator`
`BiPredicate<L, R>`	`(L, R) -> boolean`	
`BiConsumer<T, U>`	`(T, U) -> void`	`ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>`
`BiFunction<T, U, R>`	`(T, U) -> R`	`ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>`

Tabla 1: interfaces abstractas comunes.

EJERCICIO 20: ¿Cuales son expresiones lambda correctamente usadas? (observa la firma y el descriptor)

a)
```
ejecuta( () -> {} );
   public void ejecuta( Runnable r ){ r.run(); }
```

b)
```
public Callable<String> fetch() { return () -> "ejemplo ;-)"; }
```

c) `Predicate<Pera> p = (Pera a) -> a.getPeso();`

INTERFACES FUNCIONALES DE FÁBRICA.

Dentro del paquete **java.util.function** se han creado algunas interfaces funcionales de las que describimos algunas pensadas para usar programación funcional.

PREDICATE

La interface **Predicate<T>** define el método **test(T)** que devuelve un booleano. La usas cuando quieras comprobar si un objeto T cumple una condición.

```
@FunctionalInterface
public interface Predicate<T>{ boolean test(T t); }
```

EJEMPLO 38: uso de Predicate.

```
public static <T> List<T> filtra(List<T> lista, Predicate<T> p) {
    List<T> resultados = new ArrayList<>();
    for(T s: lista) {
        if( p.test(s) ){ resultados.add(s); }
    }
    return resultados;
 }

    :
    Predicate<String> pStringNoVacio = (String s) -> !s.isEmpty();
    List<String> noVacios = filtra(listaCadenas, pStringNoVacio);
    :
```

CONSUMER

La interface **Consumer<T>** define **accept(T)** que no devuelve nada. Puedes usarla cuando necesites hacer alguna operación sobre un objeto T sin devolver nada.

```
@FunctionalInterface
public interface Consumer<T> { void accept(T t); }
```

EJEMPLO 39: Uso de Consumer.

```
public static <T> void forEach( List<T> lista, Consumer<T> c ) {
    for(T i: lista) { c.accept(i); }
}

:
forEach( Arrays.asList(1,2,3,4,5),
         (integer i) -> System.out.println(i)
       );  // usar con lambda
:
```

FUNCTION

La interface `Function<T, R>` define `apply(T)` que devuelve un objeto R. La usas cuando quieres mapear un objeto de tipo T en otro de tipo R.

```
@FunctionalInterface
public interface Function<T, R> { R apply(T t); }
```

EJEMPLO 40: transformar una lista de strings en una lista de enteros con la longitud de cada string.

```
public static <T, R> List<R> map(List<T> lista, Function<T,R> f) {
    List<R> resultado = new ArrayList<>();
    for(R s:  lista){ resultado.add( f.apply(s) ); }
    return resultado;
}

:
List<Integer> li = map( Arrays.asList(),
                        (String s) -> s.length()
       );  // usar con lambda
:
```

ESPECIALIZACIONES DE PRIMITIVAS.

Las interfaces que hemos visto se aplican a referencias a objetos (Integer, Double, ...), pero no a tipos primitivos (int, double, ...) esto es debido al mecanismo que usa la parametrización con genéricos, aunque Java tiene **boxing** para convertir automáticamente objetos en tipos primitivos y **unboxing** para la conversión auntomática contraria, lo que permite que este código sea legal:

```
List<Integer> lista = new ArrayList<>();
for(int i = 300; i < 400; i++){ lista.add(i); }
```

Pero en eficiencia tiene un coste: los valores a los que se aplican boxing, al ser objetos, necesitan memoria del heap para existir, por tanto consumen más memoria. Para evitarlo, a partir de Java 8 se crean **interfaces funcionales prefabricadas para cada tipo primitivo** y mejorar la eficiencia. Así por ejemplo tenemos **IntPredicate** que equivale a `Predicate<Integer>` pero sin hacer boxing.

EJEMPLO 41: Uso de primitivas especializadas.

```
public interface IntPredicate {  boolean test(int t);   }
IntPredicate pares = (int i) -> i % 2 == 0;
pares.test(1000);
```

5.8.3 REFERENCIA A MÉTODOS&COMPOSICIÓN LAMBDAS

Una referencia a método te permite crear una función lambda desde la implementación del método. Esto permite indicar directamente una referencia a un método que ya exista, en vez de definir una expresión lambda que utilice ese método. Antes:

```
inventario.sort( (Pera p1, Pera p2) -> p1.getPeso().compareTo( p2.getPeso()) );
```

Después usando una referencia a método y java.util.Comparator.

```
inventario.sort( Comparing( Pera::getPeso) );
```

Cuando quieras usar una referencia a método, debes escribir primero el objeto o la clase que tenga el método, luego se usan "::" y a continuación el nombre del método. **No lleva paréntesis porque no estás llamando al método**, sino indicando o pasando el método que quieres que se use.

EJEMPLO 42: ejemplos de uso de referencias a métodos:

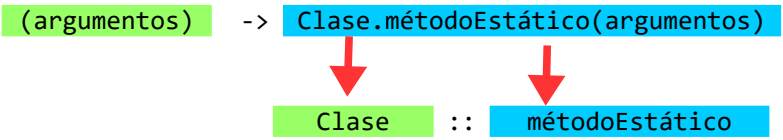
Método Lambda	Referencia Equivalente

```
(Pera p) -> p.getPeso()
() -> Thread.currentThread().dumpStack()
(str, i) -> str.substring(i)
(String s) -> System.out.println(s)
```

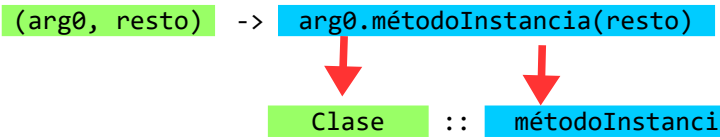
```
Pera::getPeso
Thread.currentThread()::dumpStack
String::substring
System.out::println
```

Hay 3 tipos de referencia a métodos:

- **A un método estático**. Por ejemplo a `Integer.parseInt()` donde escribimos `Integer::parseInt`.

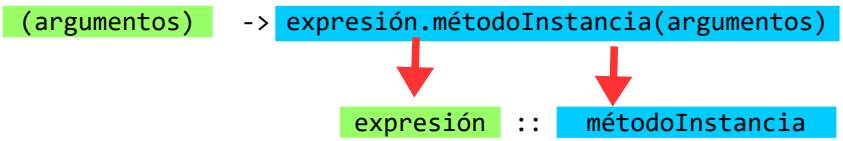


- **A un método de instancia de un tipo arbitrario**. Por ejemplo `String.length()` donde escribimos `String::length`



`arg0 es de tipo Clase`

- **A un método de instancia de un objeto existente**. Por ejemplo si la variable local `unaCadena` tiene una referencia a `String`, puedes escribir `unaCadena::length`.



EJEMPLO 43: imagina que quieres ordenar una lista de strings, ignorando mayúsculas y minúsculas. El método `sort()` de `List` espera un parámetro `Comparator`. `Comparator` describe un descriptor de función con la firma `(T, T) -> int`. Puedes definir una expresión lambda que use el

método `compareToIgnoreCase()` de la clase `String`:

```
List<String> str = Arrays.asList("a", "b", "A", "B");
str.sort( (s1, s2) -> s1.compareToIgnoreCase(s2) );
```

La expresión lambda tiene una firma compatible con el descriptor de la función de `Comparator`. Por tanto podemos escribir lo anterior usando una referencia a método:

```
List<String> str = Arrays.asList( "a", "b" ,"A" ,"B");
str.sort( String::compareToIgnoreCase );
```

EJERCICIO 21: ¿Cuales son las referencias a métodos equivalentes de las siguientes expresiones lambda?

1. `Function<String, Integer> sTi= (String s) -> Integer.parseInt(s);`
2. `BiPredicate<List<String>, String> c = (lista, elemento) -> lista.contains(elemento);`

También puedes usar referencias a métodos con los constructores de objetos usando: `Clase::new` que es como llamar a un método estático.

EJEMPLO 44: uso de referencias a constructores y equivalencias con expresiones lambda.

```
Supplier<Pera> c1 = Pera::new;
Pera p1 = c1.get();
```

Que equivale con expresiones lambda a esto:

```
Supplier<Pera> p1 = () -> new Pera();
Pera p1 = c1.get();
```

Si tienes un constructor con la firma similar a esta: `Pera(int peso)`, puedes usar `Function`.

EJEMPLO 45: constructores con parámetros.

```
Function<Integer, Manzana> c2 = Manzana::new;
Manzana a2.apply(110);
```

Que equivale con expresiones lambda a esto:

```
Function<Integer, Manzana> c2 = (peso) -> new Manzana(peso);
Manzana a2 = c2.apply(110);
```

Si tienes una lista de objetos a crear, puedes usar map.

EJEMPLO 46: crear más de un objeto.

```
List<Integer> listaPesos = Arrays.asList(7, 3, 4);
List<Pera> la = map(listaPesos, Pera::new);

public static List<Pera> map(List<Integer> lista,
                             Function<Integer, Pera> f) {
    List<Pera> resultado = new ArrayList<>();
    for(Integer e : lista) { resultado.add( f.apply() ); }
    return resultado;
}
```

Si tienes un constructor con 2 argumentos `Pera(String color, int peso)` la firma es como la función de `BiFunction`.

EJEMPLO 47: constructor con dos argumentos.

```
BiFunction<String, Integer, Pera> c3 = Pera::new;
Pera a3 = c3.apply("verde", 10);
```

Que equivale con expresiones lambda a esto:

```
BiFunction<String,Integer,Pera> c3= (color,peso) -> new Pera(color, peso);
Pera a3 = c3.apply("verde", 10);
```

EJEMPLO 48: para finalizar con el uso de lambdas vamos a ver como resolver el problema de ordenar por diferentes criterios una lista de datos. Primero usaremos el código tradicional, y poco a poco iremos generando código equivalente cada vez más conciso con lambdas, y referencias a métodos. Incluso en las primeras versiones de Java, la parte más complicada ya estaba resuelta con el método `sort()`:

```
void  sort( Comparator<? Super E> c );
```

```
// ------ SOLUCIÓN ORIGINAL: creando una clase que implemente la estrategia
```

```
public class ComparadorPesoPera implements Comparator<Pera> {
    public int compare(Pera a1, Pera a2) {
        return a1.getPeso().compareTo( a2.getpeso() );
    }
}
:
inventario.sort( new ComparadorPespPera );
:

// ------ SOLUCIÓN CON CLASE ANÓNIMA:
inventario.sort( new Comparator<Pera()> {
                        public int compare(Pera a1, Pera a2) {
                           return a1.getPeso().compareTo( a2.getpeso() );
                        }
                } );
:

// ------- SOLUCIÓN CON LAMBDAS
inventario.sort( (a1, a2) -> a1.getPeso().CompareTo( a2.getPeso() ) );
:

// ------- SOLUCIÓN CON LAMBDAS usando comparing
import static.java.util.Comparator.comparing;
Comparator<Pera> c = Comparator.comparing( (Pera a)-> a.getPeso() );
inventario.sort( c );

// ------ IGUAL MÁ CONCISA
import static.java.util.Comparator.comparing;
inventario.sort( comparing( (a) -> a.getPeso() ) );
:

// ------ SOLUCIÓN CON REFERENCIAS A MÉTODOS
import static.java.util.Comparator.comparing;
inventario.sort( comparing(Pera::getPeso) );
```

COMPOSICIÓN DE EXPRESIONES LAMBDA.

La mayoría de interfaces funcionales vienen preparadas para soportar composición de funciones lambda, es decir, combinar varias expresiones lambda sencillas para realizar un trabajo más complejo.

Por ejemplo puedes componer comparadores. Usando `comparing` puedes indicar con qué dato de un objeto quieres generar un `Comparator`, como hemos realizado antes:

```
inventario.sort( comparing( (a) -> a.getPeso() ) );
```

Si quieres ordenarlas al contrario, no es necesario que crees un nuevo `Comparator`, la interfaz tiene un método por defecto llamado `reversed()`:

```
inventario.sort( comparing( (a) -> a.getPeso() ).reversed() );
```

A veces quizás quieras aplicar más de un segundo criterio para ordenar, primero por uno y luego por otro. En este caso tienes el método **`thenComparing()`**:

```
inventario.sort( Pera::getPeso )
          .reversed()
          .thenComparing( Pera:getColor );
```

COMPONER PREDICADOS

La interface **`Predicate`** implementa 3 métodos para poder aprovechar resultados anteriores y combinarlos con otros. Estos 3 métodos son: **and()**, **or()** y **negate()**. Ejemplos:

```
Predicate<Apple> rojas = (a) -> getColor.equals("rojo");
Predicate<Apple> noRojas = rojas.negate();
Predicate<Apple> nuevo = rojas.and( a -> a.getPeso() > 150 )
                              .or( a -> "verde".equals( a.getColor()) );
```

COMPONER FUNCIONES

La interface **`Function`** implementa 2 métodos para poder aprovechar resultados anteriores y combinarlos con otros. Estos 2 métodos son: **`andThen()`** que se aplica después de aplicar la primera y **`compose()`** que se aplica antes de aplicar la primera.

EJEMPLO 49: Componer expresiones

```
Function<Integer,Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.anThen(g);
Function<Integer, Integer> h2 = f.compose(g);
int resultado1 = h.apply(1);   // devuelve 4: g( f(x) )
int resultado2 = h2.apply(1);  // devuelve 3: f( g(x) )
```

5.9. INTRODUCCIÓN A LOS JAVABEANS.

Son componentes software diseñados para ser reutilizables en diferentes entornos. Puede realizar una función sencilla como obtener un valor sumando otros o tareas complejas como hacer un forecasting (una predicción basándose en datos del pasado y del presente) del stock de una empresa. Un bean podría ser visible al usuario como un botón de una GUI o invisible como un decodificador de streams de datos multimedia. Por último, un Bean podría fabricarse para trabajar de forma autónoma o colaborando con otra serie de componentes distribuidos en una red. El software que genera un gráfico de tarta a partir de unos datos podría ser el primer tipo, pero un bean que informe del cambio de divisas en tiempo real necesita interactuar con otros componentes que le suministren los datos. Las ventajas de un JavaBean:

- Sigue el paradigma "Escribir una vez, ejecutar siempre".
- Sus propiedades, eventos y métodos son expuestos para que otras aplicaciones los usen.
- Software auxiliar puede ayudar a configurar un Bean en tiempo de diseño. Estos elementos no son necesarios en tiempo de ejecución.
- El estado de un bean puede almacenarse para restarurarse más tarde.
- Pueden recibir eventos de otros objetos y generar eventos que son enviados a otros objetos.

Es un modelo de componentes de Java cuya definición podría ser: "Un componente de software reusable que puede manipularse visualmente en una herramienta de diseño" y al que es posible cambiar su apariencia o su comportamiento en tiempo de diseño o en tiempo de ejecución

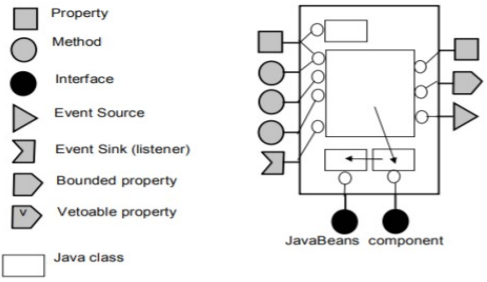
(mediante código). Es una clase puramente Java desarrollada con unos patrones de diseño bien definidos, que permiten que sea usada en posteriores aplicaciones y gestionada de forma automática.

Es un modelo sencillo, soportado directamente por el entorno Java, Multiplataforma (aunque no multilenguaje). Ejemplos de JavaBeans: Librerías gráficas AWT (de Sun) y SWT (de Eclipse).
Representa una implementación del modelo Propiedad-Evento-Método.
Un componente JavaBean se define a través de:

- Las propiedades que expone.
- Los métodos que ofrece.
- Los eventos que atiende o genera.

Para gestionar estas características, todo JavaBean debe ofrecer:

- **Soporte para "Introspection"**: El bean tiene que ofrecer la información necesaria para que la herramienta de diseño pueda analizar sus características de forma opaca.
- **Soporte para "Customization"**: La herramienta de construcción de la aplicación puede adaptar ("customizar") la apariencia o comportamiento del bean a la aplicación.
- **Soporte para Persistencia:** El estado de un bean customizado puede ser almacenado para ser utilizado más tarde.
- **Soporte para Eventos:** Los beans se comunican a través del envío y recepción de eventos.



Reglas de diseño básicas para la construcción de un JavaBean:

- Clases públicas.
- Constructor vacío por defecto (puede tener más).
- Implementación de la interfaz Serializable (para tener persistencia).
- Seguir las convenciones de nombres establecidas.

EJEMPLO 50: un primer Bean sencillo.

```
public class MiPrimerBean implements Serializable {
    String unaPropiedad;
    public MiPrimerBean(){ unaPropiedad = ""; }
    public void unMetodo(){
        System.out.println("Ejemplo sencillo de JavaBean");
    }
    public void setUnaPropiedad(String s){ unaPropiedad = s }
    public String getUnaPropiedad(){ return unaPropiedad; }
}
```

Propiedades

- Son atributos con nombre que definen el estado y el comportamiento del componente.
- **Patrón de diseño:**
  - `public void setX(valor);`
  - `public tipoX getX();`
- Si la propiedad x es booleana, el metodo de acceso es:
  - `public boolean isX();`

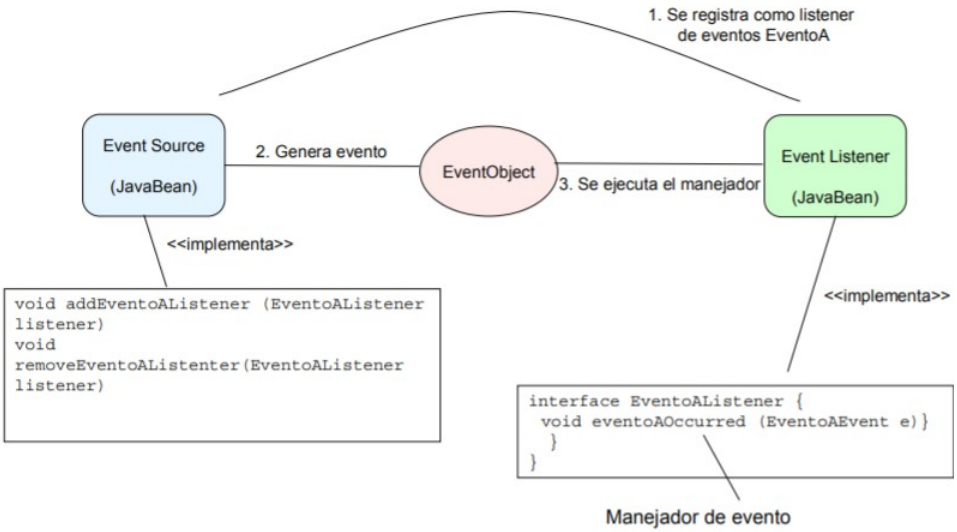
- Estos métodos constituyen el único modo de acceso a las propiedades.
- Tipos de propiedades:
  - Simples : con un valor único
  - Indexed: Representa arrays de valores. En este caso los métodos de acceso son los siguientes:
    - `public void setX([] valor);`
    - `public [] getX();`
    - `public void setX(valor, int indice);`
    - `public getX(int indice);`

MODELO DE EVENTOS JAVA

El mecasnismo de comunicación de eventos en Java es síncrono y participan: los eventos que se intercambian, quienes los generan (fuentes) y quienes los escuchan y tratan (Listeners).

Objetos Event

- Es el único parámetro que reciben los manejadores de eventos.
- Encapsulan toda la información asociada a la ocurrencia de un evento. Entre ella el objeto que lanza el evento, el momento de tiempo y otra información que ayuda a trabajar con él.
- Todos heredan de la clase *java.util.EventObject*.
- Por cada tipo de evento que exista en la aplicación se define una subclase denominada <NombreEvento>Event.



EJEMPLO 51: Evento que notifica un cambio de temperatura

```
public class CambiaTemperaturaEvent extends java.util.EventObject{
    protected double temperatura;  // La nueva temperatura

    public CambiaTemperaturaEvent(Object source, double grados){
        super(source);
        temperatura = grados; // se guarda
    }
    public double getTemperatura() { return temperatura; }
}
```

EventListeners

- Son objetos que necesitan ser avisados cuando ocurran eventos.
- La notificación del evento se produce invocando a métodos manejadores en el objeto "listener".
- Los métodos manejadores se agrupan en interfaces que extienden a la interfaz *java.util.EventListener*.
- Por cada tipo de evento que se quiera manejar, se define una

interfaz denominada *<NombreEvento>Listener*.

- Los métodos para el manejo de eventos siguen un patrón:

```
void <eventoNombreMétodo>(<EventObjectType> evt);
```

EJEMPLO 52: Clase que maneja eventos del tipo `CambiaTemperaturaEvent`

```
interface CambiaTemperaturaListener extends
java.util.EventListener {
    // Este método es llamado cuando cambie la temperatura
    void cambiaTemperatura(CambiaTemperaturaEvent evt);
}

// Ejemplo de objeto "listener" de eventos CambiaTemperaturaEvent
class Termometro implements CambiaTemperaturaListener {
    public void cambiaTemperatura(CambiaTemperaturaEvent evt) {
        ...
    }
}
```

Event Sources

- Son los objetos que generan y lanzan eventos
- Proporcionan métodos para que los objetos "listener" puedan registrarse en ellos y así ser notificados de los eventos que se produzcan.
- Estos métodos siguen el siguiente patrón de diseño:
  - ***public void*** *add<TipoListener>(<TipoListener> listener);*
  - ***public void*** *remove<TipoListener>(<TipoListener> listener);*
- Toda clase que presente el anterior patrón se identifica como fuente de eventos del tipo correspondiente a `TipoListener`
- Cuando se produce un evento es notificado a todos los listeners que se hayan registrado (Multicast delivery)
- Existe también la posibilidad de notificar el evento en modo unicast:
  - ***public void*** *add<TipoListener>(<TipoListener> listener)* ***throws*** *java.util.TooManyListenersException;*

- *public void remove<TipoListener>(<TipoListener> listener);*

- La invocación de los métodos de notificación en los objetos listener se realiza de forma síncrona.

EJEMPLO 53: notificación de un evento.

```
public class Temperatura {
  protected double t;
  // lista de Listeners para cambios de temperatura
  private List<CambioTemperaturaListener> lis= new LinkedList<>();
  public Temperatura() { t = 22.2; }
  public synchronized void
  addCambioTemperaturaListener(CambioTemperaturaListener li) {
      lis.add(li);
  }
  public synchronized void
  removeCambioTemperaturaListener(CambioTemperaturaListener li) {
      lis.remove(li);
  }
  // Notificar de cambios a los objetos oyentes
  protected void notifyCambioTemperatura() {
   List<CambioTemperaturaListener> l;
   // crear objeto event
   CambioTemperaturaEvent e = new CAmbioTemperaturaEvent(this, t);
   // Copiar lista de objetos Listener para que no cambie
   // mientras se disparan los eventos
   synchronized(this) {
       l = (LinkedList)lis.clone(); }
       for(int i = 0; i < l.size(); i++) {
           (l.elementAt(i)).cambiaTemperatura(e);
       }
   }
```

EJEMPLO 54: Event adapters:

```
public class Termometro
   // referencia a 2 objetos temperatura que monitoriza
   protected Temperatura t1;
   protected Temperatura t2;
   // Adaptadores de cambios de temperatura
   protected TemperaturaAdapter1 tA1;
   protected TemperaturaAdapter2 tA2;
```

```
    // La primera clase adaptadora
    class TemperaturaAdapter1 implements CambiaTemperaturaListener{
       TemperaturaAdapter1(Temperatura t) {
           t.addCambiaTemperaturaListener(this);
       }
       public void cambiaTemperatura(CambiaTemperaturaEvent e) {
           cambiaTemperatura( e.getTemperatura() );
       }
    }
    // La segunda clase adapter
    class TemperaturaAdapter2 extends TemperaturaAdapter {
     TemperaturaAdapter2(Temperatura t) {
         t.addCambiaTemperaturaListener(this);
         public void cambiaTemperatura(CambiaTemperaturaEvent e) {
             cambiaTemperatura2( e.getTemperatura() );
         }
     }
     // constructor de la clase Termometro
     Termometro() {
         // guarda referencias a los objetos temperatura
         t1 = new Temperatura();
         t2 = new Temperatura();
         // crea los adaptadores
         tA1 = new TemperaturaAdapter1(t1);
         tA2 = new TemperaturaAdapter2(t2);
     }
     // Manejar cambios de temp. en t1
     protected void cambiaTemperatura(double nuevaTemp){}
     // Manejar cambios de temp en t2.
     protected void cambiaTemperatura2(double nuevaTemp){}
}
```

Propiedades Bound y Constrained

Bound: Cuando el valor de la propiedad cambia, se le notifica a otros beans a través de un evento del tipo `PropertyChangeEvent`.

- Un bean con propiedades de este tipo debe implementar:
- *`public void addPropertyChangeListener(PropertyChangeListener x);`*
- *`public void removePropertyChangeListener(PropertyChangeListener x);`*
- Los listener implementan la interfaz `PropertyChangeListener`
- La notificación se realiza cuando el cambio de la propiedad ya

se ha realizado

Constrained: Cuando el valor de la propiedad cambia, se le notifica a otros beans que pueden rechazar el cambio.

- El método set para este tipo de propiedades tiene un patrón especial:
  - ```
    public void set<PropertyName>(<PropertyType> value) throws java.beans.PropertyVetoException;
    ```
- Un bean con propiedades de este tipo debe implementar:
  ```
  public void addVetoableChangeListener(VetoableChangeListener p);
  public void removeVetoableChangeListener(VetoableChangeListener p);
  ```
- Los listener deben implementar la interfaz `VetoableChangeListener`
- La notificación se realiza antes de que el cambio de la propiedad se haya realizado

INTROSPECTION

- Mecanismo para conocer las características de un componente, necesarias para poder manipularlo desde la herramienta de diseño sin acceder a su código:
  - En el caso de un componente JavaBean permite conocer sus propiedades, métodos y eventos.
  - Se requieren mecanismos simples, que no obliguen al desarrollador a conocer o escribir código complicado.
- El modelo JavaBeans ofrece dos tipos de introspección:
  - Implícita: A través de la interfaz *java.beans.Introspector*
  - **Explícita:** A través de clases que implementan la interfaz *java.beans.BeanInfo*, que se añaden y se distribuyen junto a la implementación del JavaBean.

Primera opción: interface Introspector
  ```

Mecanismo reflector de bajo nivel ofrecido por Java: analiza el código de una clase, extrayendo los métodos que la clase ofrece. A partir de ellos se puede extraer la información acerca de propiedades y eventos

- Se basa en reconocimiento de patrones de diseño:
  - Para propiedades (según sea de lectura o lectura/escritura):
    - *public <PropertyType> get<PropertyName>();*
    - *public void set<PropertyName>(<PropertyType> a);*
  - Para propiedades indexadas (arrays de varios elementos):
    - *public tipo[] get<PropertyName>();*
    - *public void set<PropertyName>(tipo[] a);*
    - *public tipo get<PropertyName>(int indice);*
    - *public void set<PropertyName>(int indice, tipo e);*
  - Para eventos:
    - *public void add<EventListenerType>(<EventListenerType> a)*
    - *public void remove<EventListenerType>(<EventListenerType> a)*
    - …
  - Para métodos:
    - Todos los métodos públicos son expuestos
- En JavaBeans los patrones no son obligatorios, pero son necesarios si se quiere utilizar la interfaz `Introspector`
- Cuando se analiza una clase, se analizan todas las clases superiores en la jerarquía.
- Es el mecanismo que usa el plug-in de Eclipse para desarrollo de interfaces gráficas.

Segunda opción: clases Bean Info

El desarrollador del componente especifica qué métodos, propiedades y eventos quiere que su Bean exponga a las herramientas de diseño.

- Para ello acompaña a la implementación del Bean con una clase que implemente la interfaz BeanInfo. La clase se denomina siempre *<BeanName>BeanInfo;*
- BeanInfo ofrece métodos para describir los métodos, eventos y

propiedades de un JavaBean. Ofrece, entre otros, estos métodos:

- getBeanDescriptor()
- getEventSetDescriptors()
- getPropertyDescriptors()
- getMethodDescriptors()

- Mejor que implementar directamente toda la interfaz, es extender la clase `java.beans.SimpleBeanInfo`
- En este caso no se examinan las clases antecesoras, hay que dar toda la información a través del objeto.

EJEMPLO 55: este ejemplo muestra el uso de introspection usando una clase `BeanInfo`. Hace uso de las clases `Introspector`, `PropertyDescriptor` y `EventSetDescriptor`. La primera clase es el Bean y se llama `Colores`:

```
//Fichero: Colores.java
package p5;

import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colores extends Canvas implements Serializable {
    transient private Color c;    // no persistente
    private boolean rectangular;  // persistente
    private static final long serialVersionUID = 1L;

    public Colores() {
        addMouseListener(
                        new MouseAdapter() {
            public void mousePressed(MouseEvent me) { cambia(); }
            } );
            rectangular = false;
            setSize(200, 100);
            cambia();
    }

    public boolean getRectangular() { return rectangular; }
    public void setRectangular(boolean flag) {
        rectangular = flag;
        repaint();
```

```
    }

    public void cambia() {
        c = randomColor();
        repaint();
    }

    private Color randomColor() {
        int r = (int) (256 * Math.random());
        int g = (int) (256 * Math.random());
        int b = (int) (256 * Math.random());
        return new Color(r, g, b);
    }

    public void paint(Graphics g) {
        Dimension d = getSize();
        int altura = d.height;
        int anchura = d.width;
        g.setColor(c);
        if(rectangular)
                g.fillRect(0,0, anchura-1, altura-1);
        else
                g.fillOval(0,0, anchura-1, altura-1);
    }
}
```

El Bean `Colores` muestra un objeto coloreado en un frame. El color del componente lo decide la variable privada c, y la figura la variable rectangular.
El constructor define una clase anónima que extiende a `MouseAdapter` y sobreescribe su método `mousePressed( )`. El método `cambia()` se llama en respuesta a una pulsación de ratón. Selecciona un color aleatorio y redibuja la figura. Los métodos `getRectangular()` y `setRectangular()` ofrecen el acceso a la propiedad del Bean. EL método `paint()` del Bean usa las variables c y rectangular para decidir como lo visualiza.

La siguiente clase es `ColoresBeanInfo`. Es subclase de `SimpleBeanInfo` que aporta información explícita sobre el Bean `Colores`. Sobreescribe `getPropertyDescriptors()` para decidir que propiedades ve el usuario del Bean. En este caso, solamente es rectangular. El método crea un

objeto `PropertyDescriptor` y lo devuelve. El constructor de `PropertyDescriptor` que se utiliza es este:

```
PropertyDescriptor(String property, Class<?> beanCls) throws
IntrospectionException
```

El primer parámetro es el nombre de la propiedad y el segundo es la clase del Bean.

```
//Fichero ColoresBeanInfo.java
package p5;

import java.beans.*;

public class ColoresBeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[]  getPropertyDescription(){
        try {
            PropertyDescriptor rectangular = new
                        PropertyDescriptor("rectangular", Colores.class);
            PropertyDescriptor[] pd = { rectangular };
            return pd;
        } catch(Exception e) {
            System.out.println("Excepción " + e );
            return null;
        }
    }
}
```

La última clase se llama `IntrospectorDemo`. Usa introspection para mostrar las propiedades y eventos disponibles en el Bean `Colores`.

```
//Fichero IntrospectorDemo.java
package p5;

import java.beans.*;

public class InstrospectorDemo {
    public static void main(String[] args) {
        try {
            Class<?> c = Class.forName("p5.Colores");
            BeanInfo bi = Introspector.getBeanInfo(c);
            System.out.println("Propiedades: ");
            PropertyDescriptor[] pd = bi.getPropertyDescriptors();
            for(int i = 0; i < pd.length; i++)
                System.out.print("\t" + pd[i].getName() );
```

```
            System.out.println("\nEventos: ");
            EventSetDescriptor[] esd = bi.getEventSetDescriptors();
            for(int i = 0; i < pd.length; i++)
                System.out.print("\t" + esd[i].getName() );
            } catch(Exception e) {
                System.out.println("Excepción " + e );
            }
        }
}
```

Customizers

- Mecanismo que permite al usuario de un componente adaptar su apariencia y comportamiento a la aplicación concreta en la que se vaya a utilizar:
- Sin necesidad de acceder o escribir ningún código
- Modificando sus propiedades
- Dos mecanismos:
  - Editor de propiedades incluido en el entorno de desarrollo
  - Vista Properties en el VisualEditor
- Uso de clases "customizer" (incluidas en la clase BeanInfo)
- Son asistentes (*wizards*) que van pidiendo los valores de configuración del componente

Persistencia

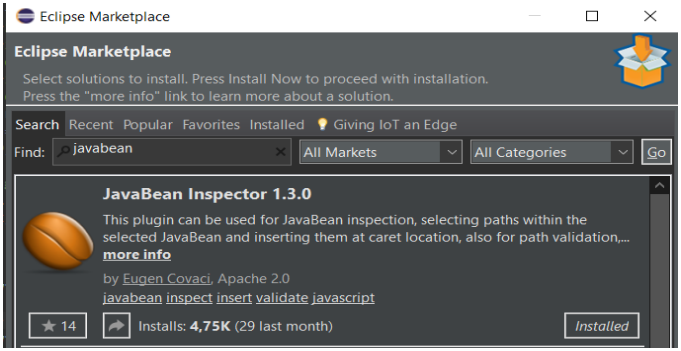
- Capacidad de almacenar el componente en el estado concreto que tenga en un momento determinado, y poder re-instanciarlo posteriormente en el mismo estado.
- Implementando la interfaz Serializable se puede almacenar el estado de los beans a través de las librerías de Streams que ofrece Java.
- Se deberán guardar aquellas características que permitan reincorporar al bean posteriormente en el mismo estado y con la misma apariencia.
  - Propiedades expuestas

  - Propiedades internas
- Cuando se carga el componente en el programa se hace a partir del fichero binario serializado.

Empaquetamiento

- Una vez desarrollado un JavaBean se empaqueta para su posterior distribución y utilización.
- El empaquetamiento y distribución de JavaBeans se realiza a través de paquetes .jar. Se pueden incluir varios beans en un mismo jar
- En el .jar correspondiente a un bean se incluirán todas las clases que lo forman:
  - El propio bean,
  - Objetos BeanInfo,
  - Objetos Customizer,
  - Clases de utilidad o recursos que requiera el bean, etc
- El .jar debe incluir siempre un manifest-file (fichero .mf) que describa su contenido.

Cuando el Bean se use en una herramienta de dessarrollo visual, debe exponer sus propiedades, métodos y eventos y así permitir a la herramienta y al usuario que manipule su apariencia y comportamiento:



¿Un Bean solo sirve para aplicaciones con GUI?
En realidad no. Aunque se asocian a los componentes visuales usados en los editores de interfaces en realidad también se usan en servidores sin nisiquiera interfaz gráfica. La clase **java.beans.Beans** contiene el método `setGuiAvailable()`, que puede utilizarse para saber si hay GUI disponible. Así el Bean puede saber si no debe intentar interactuar con el usuario a través de cajas de diálogo u otro tipo de sistema.

Aunque hay Beans que no tienen sentido sin la presencia de una GUI, la interface **java.beans.Visibility** puede implementarse en un Bean que quiera funcionar en un entorno sin GUI. El método `dontUseGui()` puede llamarse para avisar al Bean que no use la GUI y el método `avoidingGui()` se llama para comprobar si el Bean efectivamente no está usando la GUI.

5.9.1. ENTERPRISE JAVA BEANS

Se trata de una tecnología que permite definir componentes que encapsulan objetos de negocio gestionados por un servidor de aplicaciones. A partir de la especificación 3.0 de EJB, se utilizan anotaciones para definir los beans, sustituyendo a los ficheros de descripción de propiedades XML y la introducción de un framework de persistencia muy similar a Hibernate, el JPA (Java Persistence API).

Comparamos la arquitectura Enterprise JavaBeans (EJB).
Lo mejor es compararla con la arquitectura de las aplicaciones Web que permite el desarrollo de aplicaciones distribuidas cliente-servidor. Los clientes (navegadores Web) realizan peticiones a un servidor. Las peticiones se realizan en el protocolo HTTP que permite definir la petición y los argumentos que se pasan al servidor. El servidor recoge la petición, comprueba ciertos permisos (seguridad) y, si todo es

correcto, la ejecuta mediante la llamada a un **servlet** (código Java) asociado a la petición. El servlet recoge los parámetros de la petición, los procesa, consulta recursos del servidor (bases de datos, objetos, etc.) y devuelve el resultado, en forma de texto HTML. Supongamos que tenemos un servidor que gestiona una empresa de mensajería y queremos saber los pedidos que va a entregar un agente determinado. La forma de hacerlo siguiendo esta arquitectura sería realizar una petición HTTP en la que pondríamos algo parecido a:

```
/listaPedidos?agente=107
```

con la que pedimos la lista de pedidos asignada al agente 107. El resultado sería un código HTML parecido a:

- Pedido id=883247110
- Pedido id=392349429
- Pedido id=232494592
- Pedido id=945821134

El navegador interpretará este texto HTML y mostrará la lista de pedidos por pantalla. Este tipo de comportamiento puede ser apropiado si queremos mostrar el listado de pedidos. Pero si queremos que esta respuesta sea procesada por un programa que realiza algún proceso con los identificadores de los pedidos, el enfoque ya no es tan apropiado.

Podríamos definir un cliente Java que (utilizando las librerías de red) realizara una petición HTTP y recogiera la respuesta (fichero de texto HTML). El proceso debería recoger el HTML, descodificarlo (buscando ciertas etiquetas que deberíamos saber que están ahí) y obtener los identificadores. Este enfoque es factible pero introduce muchos problemas, no sólo de programación, sino de mantenimiento. Si en un momento alguien cambia el texto devuelto por el servlet, hay que cambiar el código del proceso que descodifica el resultado. ¿No sería

interesante poder escribir un código Java en el cliente que pida los datos "directamente" al servidor? Sin utilizar HTTP, sino de una forma mucho más directa. Por ejemplo:

```
(1) PedidosFactory pF = getPedidosFactory(serverConection);
(2) Pedidos p = pF.create();
(3) List listaP = p.getLista("107");
```

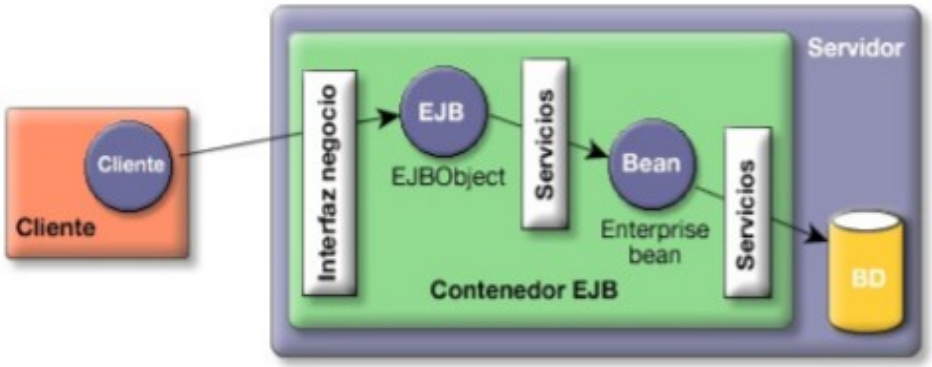
En la primera instrucción obtenemos una factoria de Pedidos. Esta factoría crea a su vez objetos de tipo `Pedidos` (con el método create) con los que podremos realizar directamente peticiones relacionadas con los pedidos. En línea 2 se crea un objeto de esta clase. Después (línea 3) usamos ese objeto para pedir el método `getLista()` que devuelve una lista de pedidos (transfer objects de tipo `Pedido`) asociados al agente "107". Esta vez obtenemos una lista de objetos Java, que podemos procesar con sus métodos get correspondientes.
Esta forma de obtener una lista de pedidos es mucho más sencilla (y mantenible) que descodificar un fichero de texto. Las tres llamadas del ejemplo anterior son remotas, realizadas desde un cliente en una máquina virtual (u ordenador) hacia un ordenador remoto en el que reside un servidor que procesa estas peticiones. El servidor del ordenador remoto recibe las peticiones, las "analiza" y las responde, incorporando sus funcionalidades añadidas.

La arquitectura EJB permite este tipo de funcionamiento. El nombre que recibe el servidor que procesa este tipo de peticiones es el de contenedor EJB. El contenedor mantiene los objetos remotos y procesa las peticiones de los clientes, de la misma forma que un servidor web contiene aplicaciones web y procesa peticiones HTTP. Sin embargo, en esta arquitectura las peticiones son más similares a las llamadas a procedimientos remotos, en las que se intenta que la red sea transparente y que parezca que estamos llamando a un objeto

normal situado en la propia máquina virtual del cliente. Para posibilitar esta arquitectura, el programador y diseñador debe implementar componentes que residen en el contenedor EJB.

Con la programación orientada a objetos puedes reutilizar clases, pero con componentes es posible reutilizar funcionalidades de mayor nivel e incluso es posible modificar estas funcionalidades y adaptarlas a cada entorno de trabajo particular sin tocar el código del componente desarrollado.

La implementación de los componentes EJB remotos se basa en el modelo de programación de objetos remotos de Java, denominado **RMI**. Con RMI es posible enviar peticiones a objetos que están ejecutándose en otra máquina virtual Java. Podemos ver un componente EJB como un objeto remoto RMI que reside en un contenedor EJB que le proporciona un conjunto de servicios adicionales. Para el desarrollador de componentes EJB es útil el conocimiento de RMI a nivel teórico, porque explica qué está sucediendo por debajo de la capa de abstracción proporcionada por la arquitectura EJB.



5.10. EJERCICIOS.

EA1. TEST

T1. Para declarar un objeto de una clase determinada, como atributo de otra clase, es necesario especificar que existe una relación de composición entre ambas clases mediante el modificador object. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T2. Si se declaran dos variables objeto a y b de la clase X, ambas son instanciadas mediante un constructor, y posteriormente se realiza la asignación a=b, el contenido de b será una copia del contenido de a, perdiéndose los valores iniciales de b. ¿Verdadero o Falso?
( )Verdadero ( )Falso

T3. Cuando escribas una clase en Java, puedes hacer que herede de una determinada clase padre (mediante el uso de **extends**) o bien no indicar ninguna herencia. En tal caso tu clase no heredará de ninguna otra clase Java. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T4. En Java los métodos heredados de una superclase deben volver a ser definidos en las subclases. ¿Verdadero o Falso?
( )Verdadero ( )Falso

T5. Dado que el método **finalize()** de la clase **Object** es **protected**, el método **finalize()** de cualquier clase que tú escribas podrá ser **public**, **private** o **protected**. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T6. Puede invocarse al constructor de una superclase mediante el uso de la referencia **this**. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T7. ¿Cuál de las siguientes características dirías que no es una de las que se suelen considerar como uno de los tres grandes pilares de la

Programación Orientada a Objetos?

- Recursividad
- Herencia
- Polimorfismo
- Encapsulación

T8. El polimorfismo ofrece la posibilidad de que toda referencia a un objeto de una clase A pueda tomar la forma de una referencia a un objeto de cualquier otra clase B. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T9. El polimorfismo puede hacerse con referencias de superclases abstractas, superclases no abstractas o con interfaces. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T10. Una clase abstracta no podrá ser nunca instanciada. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T11. Puede llamarse al constructor de una clase abstracta mediante el operador **new**. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T12. Los métodos de una clase abstracta tienen que ser también abstractos. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T13. Los modificadores **final** y **abstract** son excluyentes en la declaración de un método. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

T14. Una interfaz en Java no puede contener la implementación de un método mientras que una clase abstracta sí. ¿Verdadero o Falso?
( )Verdadero ( )Falso

T15. En Java una clase no puede heredar de más de una clase abstracta ni implementar más de una interfaz. ¿Verdadero o Falso?

( )Verdadero ( )Falso

T16. ¿Qué palabra reservada se utiliza en Java para indicar que una clase va a definir los métodos indicados por una interfaz?

- Implements
- uses
- extends

T17. Dada una clase Java que implementa dos interfaces diferentes que contienen un método con el mismo nombre, indicar cuál de las siguientes afirmaciones es correcta.

- Si los dos métodos tienen un valor de retorno de un tipo diferente, se producirá un error de compilación.
- Si los dos métodos tienen un valor de retorno de un tipo diferente, se implementarán dos métodos.
- Si los dos métodos son exactamente iguales en identificador, parámetros y tipo devuelto, se producirá un error de compilación.
- Si los dos métodos tienen diferentes parámetros se producirá un error de compilación.

T18. En Java no está permitida la herencia múltiple ni para clases ni para interfaces. ¿Verdadero o Falso? **( )Verdadero ( )Falso**

EA2. Define jerarquías de clases, o interfaces para los siguientes elementos:

- Las aeronaves pueden aterrizar y despegar. Las hay militares y civiles.
- Los aeropuertos tienen un nombre.
- Hay aeorpuertos terrestres que están en un país y una ciudad.

- También hay barcos que permiten despegar y aterrizar aeronaves.
- Un vuelo lo realiza un avión cuando despega y aterriza en el mismo o diferente lugar y está volando durante cierto tiempo.

EA3. Usa una clase abstracta llamada Figura2D que tenga los datos int x,y como su posición y los métodos abstractos perimetro() y area(). Crea las clases hija Rectángulo, Triángulo y Círculo.

EA4. Estamos desarrollando un juego y necesitamos una clase llamada `Escenario` que sea capaz de dibujar en un área de la ventana de una aplicación. Para ello, la hacemos hija de una clase llamada `Component` que le permite heredar el método `paint(Graphics g)` y dentro de él le permite dibujar. Como la queremos aprovechar para varios juegos, también debería implementar el método `insertaPersonaje(Personaje p)` y quizás algunos otros. No tenemos claro si crear una clase abstracta que los agrupe y hacer que `Escenario` sea una subclase suya o agruparlos en una interface ¿Cuál crees que sería la mejor opción? Haz un esqueleto de como quedarían definidos los elementos que aparecen en la descripción.

EA5. Tenemos las siguientes clases:

```
class Evaluacion {
	private int eval1, eval2, eval3;
	// Otros elementos
}

class Alumno {
	private String nombre;
	private Evaluacion e;
	private double notaGlobal;
```

```
        public Alumno(String nombre, Evaluacion e) {
                this.nombre = nombre;
                this.e = e;
        }

        public Evaluacion getEvaluacion() { return e; }
        public void setEvaluacion(Evaluacion e) { this.e = e; }
        // Otros elementos
}
```

a) Examina el código y piensa si hay alguna relación de composición o de agregación.
b) Puedes dibujar el diagrama de clases de UML de estas clases.
c) ¿Hay algún problema al usar la composición de esta manera?
d) Si lo hay indica qué elementos hay que modificar.
e) Haz las modificaciones necesarias.

EA6. Intentemos usar la palabra super:
La clase **Mascota** tiene las siguientes variables de instancia: int numPatas, String nombre, String voz y double peso. Crea la clase con un constructor donde se indiquen todos los atributos y getters y setters y sobreescribe tostring() para que devuelva "nombre numPatas voz peso". Además tiene el método come(double cantidad) que hace incrementar el peso de la mascota en cantidad y el método boolean juega(double minutos) que hace bajar el peso en minutos/2 siempre dejando el peso por encima de 0, o devuelve false si jugar tanto tiempo le hiciese perder todo su peso. El peso no tiene setter pero sí getter.

b) Crea la clase hija Perro que añade la variable de instancia color y usa la palabra super en el constructor y modifica el método juega para que si no quiere/puede jugar, antes coma minutos-1 cantidad y luego juegue minutos/2. Usa super también en la implementación.

EA7. Usemos this y super en una clase que representa un Intervalo (un trozo) cerrado de la recta de números enteros [menor, mayor]:

```
// Representa el intervalo [menor, mayor]
public class Intervalo {
    private int menor, mayor;

    public Intervalo(int menor, int mayor) {
         if( mayor < menor ) {
           int tmp = mayor;
           mayor = menor;
           menor = tmp;
        }
        this.mayor = mayor;
        this.menor = menor;
    }
}
```

a) Añade la precondición en el constructor de que mayor debe ser distinto de menor.

b) Añade getters y setters manteniendo la precondición del apartado a

c) Añade el método boolean estaDentro(int x) que devuelva true cuando x está dentro del intervalo o false en caso contrario.

d) Queremos añadir a esta clase un constructor por defecto (sin parámetros) que genere a partir de un LocalDateTime menor y mayor:

d1 <- la hora del día + segundos * 10

d2 <- el mes del año * minutos

Pero queremos aprovechar el constructor anterior. ¿Cómo podrías hacerlo?

e) Creamos la interface Localizable que obliga a implementar el método "boolean estaDentro(int x)".

f) Haz que la clase Intervalo la implemente.
g) Define la subclase IntervaloAbierto que añade dos variables de instancia:
boolean abiertoAbajo, abiertoArriba.
h) Añade constructor por defecto para que sea abierto por arriba y por abajo y aproveche el constructor por defecto de la clase padre.
i) Crea el constructor con todos los atributos y aprovecha el contructor de la clase padre.
j) Modifica el método estaDentro(int x) para que en caso de que el método del padre indique que pertenece, comprobemos si es abierto por abajo que x es mayor estricto que menor. Y si es abierto por arriba, que x sea menor estricto que mayor.

EA8. Escoge entre clase, clase abstracta e interfaz y crea el esqueleto de las siguientes cosas que hay en una granja teniendo en cuenta que:

- subclase ES UN/UNA superclase -> Herencia
- supeclase no sabe como sus hijos harán algo, pero quiere que lo hagan -> declara pero no implementa algún comportamiento -> clase abstracta.
- clase DEBE SER CAPAZ DE hacer cosas -> Interface

Lo que nos vamos a encontrar en la granja de mi tía es:

- Seres vivos que no hacen la fotosíntesis ni se reproducen por esporas y tienen dos variables de instancia de esta forma: "LocalDateTime nacimiento" y "String formaDeNacer" con el método "void dimeNacesDe()" del que se delega la implementación.
- Es capaz de volar (tiene el método void vuela() que imprime

como vuela).
- Vaca.
- Oveja.
- Gallina ponedora.
- Perro.
- Paloma
- Gato.
- Tigre.
- MulaMecanica (una especie de pequeño tractor).
- Canario
- Es un animal mascota (acompaña) dentro de la casa: habla()
- Es un animal que vive con el hombre porque le es útil (trabaja, proporciona alimento, etc.)
- Desempeña un trabajo (void trabaja()).

EA9. Indica si las dos interfaces siguientes son funcionales y en caso afirmativo crea expresiones lambda equivalentes y las implementas y las pruebas en una clase ejecutable:

a)
```
interface I1 {
  public void carga(int Kg);
  default public boolean cabeMas() {
       return true;
  }
}
```

b)
```
interface I2 {
 public int maximo(int n1, int n2);
       public String toString();
}
```

EA10. La interface java.io.FileFilter de java permite crear un filtro para descartar nombres de ficheros. Esta interface funcional declara el método:

boolean accept(File f);

Adapta el siguiente ejemplo que llama al método listFiles(ficherosJava) donde ficherosJava es un objeto de una clase que implemente FileFilter y que devuelva true cuando el nombre del fichero (getName()) acabe en '.java' y false en otro caso. Así consigues un array de nombres de ficheros fuente de la carpeta actual. La clase MiFiltro la defines como una clase pública externa en su propio fichero .java:

```
import java.io.*;

public class EA10 {
   MiFiltro ficherosJava = new MiFiltro(".java");
   // Hace un listado de los ficheros que acaban en .java de la
   // carpeta actual
   static public void main(String[] args) throws IOException {
       Files[] fuentes = new File(".").listFiles( ficherosJava );
       for(int i= 0; i < fuentes.length; i++)
            System.out.println( fuentes[i] );
    }
}
```

EA11. Copia el fichero EA10.java en el fichero EA11.java y vuelve a implementa de nuevo el mismo escenario, pero ahora crea una clase no pública (en el mismo fichero que EA11.java).

EA12. Copia el fichero EA11.java en el fichero EA12.java pero ahora la clase MiFiltro que sea una clase clase interna a la clase EA12.

EA13. Copia el fichero EA12.java en el fichero EA13.java y ahora usa una clase clase anónima como filtro.

EA14. Copia el fichero EA13.java en el fichero EA14.java y ahora usa una función lambda para usarla como filtro.