original:
59 páginas
16000 palabras

UD5. POO avanzada

1. Herencia:

Permite que las características pasen de padres a hijos. Cuando una clase **hereda de otra, adquiere sus atributos y métodos visibles**, permitiendo **reutilizar el código** y las funcionalidades, que se pueden ampliar o extender.

- Clase de la que se hereda: clase padre o superclase
- Clase que hereda: clase hija o subclase.

Clase Object

`java.lang`

Clase especial de la que heredan, directa o indirectamente, todas las clases de Java. **Todas** las clases que componen la API descienden de la clase Object.

Objetivo:

- Implementar **métodos universales** en Java. (*comparaciones, clones, .toString()*)
- **Poder referenciar cualquier objeto**/tipo con una variable de tipo Object.

1.1. Subclase y superclase

Subclase: dispone de lo heredado + sus ampliaciones → atributos y métodos.
Esto aumenta su funcionalidad, a la vez que evita la repetición innecesaria de código.

La mayoría de las clases no se definen desde cero, si no que se heredan. En Java todo se hereda de la clase Object.
Otros lenguajes, como C++, permiten herencia múltiple. Java sólo permite herencia simple.
Una clase hija solo tiene una superclase, pero una superclase puede tener varias clases hijas. Una clase puede ser hija y padre a la vez.

1.2. Modificadores de acceso para herencias

La subclase hereda todos los miembros de su superclase salvo los *private*, que no son accesibles directamente en la subclase → se puede acceder a ellos indirectamente (*getters*)

protected: pensado para facilitar la herencia. **Los miembros *protegidos* serán siempre visibles para las clases que hereden**, independientemente de si la superclase y la subclase son vecinas o externas, aunque en este último caso habrá que importar la superclase.

1.3. Redefinición de miembros heredados

En una herencia, si queremos redefinirla, se pueden modificar
→ @**Override** en métodos, **ocultación** para atributos

@Override: Para sobreescribir un método tiene que tener el mismo **nombre, mismos parámetros de entrada y el mismo tipo devuelto**. Si no, **es sobrecarga**, y si se devuelve algo diferente, puede dar **error de compilación**.
Ocultación: estatura como longitud (int) → estatura como talla del uniforme (String).

1.3.1. super y super()

- super: referencia a miembros de la superclase.
- super(): referencia a los constructores de la superclase.

Igual que **this** se utiliza para la propia clase → **super** referencia a la superclase.

El método **super() invoca un constructor de la superclase**.
Restricción → si lo utilizamos, tiene que ser la **primera instrucción** que aparezca en la implementación de un **constructor**.

Las subclases no heredan los **atributos privados**, pero *están ahí y son accesibles indirectamente a través de métodos públicos heredados*. Además de esto, **deben ser inicializados** al crear un objeto de la subclase.

1.3.2. Selección dinámica de métodos

Selección dinámica de métodos: proceso por el cual se ejecuta un método u otro según la clase del método referenciado.

Cuando definimos una clase como subclase de otra, los **objetos de la subclase son también objetos de la superclase**. (Empleado es, al mismo tiempo, una Persona)

Al referenciar un objeto de una subclase, (Empleado) con una variable de la superclase, por ejemplo de tipo Persona, solo será visible lo definido en la superclase.

Cuando hay ocultación de atributos, los atributos accesibles son los definidos en la clase de la variable.

1.4.1. Método toString()

Devuelve una cadena que represente al objeto que lo invoca con toda la información que interese mostrar.

Implementación:

> **nombre cualificado de la clase** + **@** + **referencia** `paquete.Persona@2al 39a55`

Como es inútil, se debe redefinir. Hay que declararlo *public* para que trague. (Si se hace override tiene que tener el mismo nivel de acceso).

1.4.2. Método equals()

Compara dos objetos y decide si son iguales.
Es **útil para comparar tipos primitivos**, pero **no sirve para comparar objetos**, **ya compara referencias, sin fijarse en su contenido**.

El prototipo de equals() tiene un parámetro de entrada de tipo Object para poder comparar objetos de cualquier clase.
Se debe mantener al hacer Overriding, pero **para acceder a los atributos tenemos que castear el parámetro**. El cast **siempre es necesario.**

La mayoría de clases de la API tienen equals() propio, pero **las tablas no.**

> Para comparar dos tablas se compara elemento a elemento o se utiliza equals() de Arrays, pero sólo es *true* si las tablas tienen los **mismos elementos en el mismo orden**.

Double e Integer no se pueden comparar con el operador ==, ya que son objetos, no primitivos. Con ellos también se usa equals(). Los Double también pueden dar problemas con == por falta de precisión numérica.

8.4.3. Método getClass()

getClass () devuelve la clase que,a su vez, es un objeto de la clase Class.
Todas las clases de Java, incluidas object y la propia Class, son objetos de la clase Class.

instanceof

```
objeto instanceof Clase
```

- objeto: Es la instancia del objeto que se desea verificar.
- clase: Es la clase o interfaz con la que se desea comparar el objeto.

Verifica si un objeto es una instancia de una clase, una subclase de esa clase, o una clase que implementa una interfaz. Devuelve true o false.

2. Clases abstractas

Clases abstractas: tienen al menos un método sin implementar
Método abstracto: definido en una superclase, pero implementado sólo en las subclases.

Métodos que no podemos implementar en una clase por falta de datos, pero sí en sus subclases, con los atributos necesarios. Se **implementan vacíos**, solo con el prototipo, en la superclase, y hacer *overriding* en las subclases con las particularidades de la clase, que la superclase no conoce.

```
abstract void mostrarDatos();
```

Si una clase tiene un método abstracto debe ser declarada **abstract**.

Las clases abstractas no son instanciables (no se pueden crear objetos de esa clase), **existen para ser heredadas por otras**, y no para ser instanciadas.

Si una clase hereda de una abstracta, pero no implementa alguno de sus métodos, será también abstracta.

→ Sin embargo, una clase abstracta puede tener algún método implementado o atributos definidos, que heredarán las subclases, pudiendo sustituirlos u ocultarlos.

3. Interfaces

Ejemplo Animales:

Pueden, o no, emitir un sonido.

Si pueden (perro) implementan el método `void voz()`

Es distinto para cada uno, (Gato / Perro) pero *voz* es común,

9.1. Concepto de interfaz

Interfaz: funcionalidad común de clases que no tienen otra relación entre ellas.
No son instanciables por sí mismas, hay que implementarlas en la clase que la utilice.

clase abstracta Animal

→ heredan Gato y Perro, con el método abstracto voz(), pero eso descartaría otras clases que no son animales, pero emiten sonido, como Persona o Piano.

9.1.1. Definición de una interfaz

```
interface Sonido { void voz(); }
```

No se implementa el cuerpo del método → depende de la clase concreta que implemente Sonido.

9.2. Implementación de una interfaz

Definición en cada clase donde se decide qué hace exactamente voz(). (Perro → guau)
Si una clase implementa una interfaz, es obligatorio que la utilice, o el IDE canta.

9.2.1. Método default

Son **public** sin tener que especificarlo. (si quisiéramos añadir el ruido que hacen todos los animales durmiendo → default void vozDurmiendo())

9.2.2. Métodos estáticos

Se pueden implementar. **Pertenecen a la interfaz, no a las clases que los implementan ni a sus objetos**.

2.2.2. Métodos estáticos

Pueden ser static o no, y no son accesibles fuera del código de la interfaz. Generalmente son métodos auxiliares.

Los métodos declarados pero no implementados son abstractos. Se implementan en una interfaz y pueden ser estaticos / no estaticos / privados / publicos.

3.3. Herencia de interfaces

Heredan todos los atributos y métodos y se pueden añadir nuevos. Se tienen que implementar todos los métodos de las interfaces que se llamen.

Diferencia con las clases: en **interfaces sí es posible la herencia múltiple**.

3.4 Variables de tipo interfaz

Se pueden crear variables con tipo interfaz (Sonido son;). Se pueden referenciar objetos de cualquier clase que implemente Sonido. (si se llama para perro y para gato hará guau y miau → selección dinámica de métodos.

Si se usa una variable tipo interfaz, sólo se tiene acceso a métodos de Sonido y de Object, pero no a los métodos de cada clase.

Clases anónimas

Clase anónima: implementación directa de una interfaz en una línea → crea un objeto unico que no tiene una clase cuando una implementación sólo es necesaria en un sitio y no merece la pena definir una clase entera.

Acceso entre miembros de una interfaz

Todos los métodos **default** y **private** tienen **acceso entre ellos** y a los **abstractos**.
Si se omite el tipo de acceso, queda restringido a su paquete.

Comparable y Comparator.

Comparable:

Criterio de **ordenación natural** entre 2 elementos.

si obj1 va antes que obj2: < 0
si obj1 va después que obj2: > 0
si obj1 es igual que obj2: = 0
No tiene que ser -1 o +1, solo positivo y negativo

Se castea como Object igual que equals() porque para que sea útil en todas las clases tiene que ser lo más general posible.
Se tiene que implementar con equals(). Si queremos probarlo, ambos deberían dar el mismo resultado *(comparteTo da 0 e equals da true*)

Comparator:

Para ordenaciones siguiendo criterios que no necesariamente son naturales.
Comparator se implementa en una clase específica.

Anexo. Clases Internas

Clases Anidadas Estáticas:

- Definidas con la palabra clave `static`.
- Solo se crean dentro de una clase al máximo nivel.
- No requieren una instancia de la clase contenedora para crear un objeto.

Clases Internas Miembro (no estáticas):

- Definidas sin `static`.
- Un objeto de la clase interna existe dentro de un objeto de la clase contenedora.
- Acceso directo a las variables miembro de la clase contenedora.
- No pueden tener miembros `static`.

Clases Internas Locales:

- Definidas dentro de un bloque de código, como un método o un inicializador.
- Solo accesibles dentro del bloque donde se definen.
- Pueden acceder a variables locales final del método.
- Su visibilidad es limitada al bloque donde están definidas.

Clases Anónimas:

1. Similares a las clases internas locales, pero sin nombre.
2. Solo se puede crear un único objeto.
3. Comúnmente usadas para manejar eventos en interfaces de usuario.
4. Se definen dentro de una expresión `new`, y no pueden tener constructores.

JVM no sabe nada de clases internas. El compilador las convierte en clases globales, contenidas en ficheros **.class** cuyo nombre es **ClaseContenedora$ClaseInterna.class**.
Inserta variables ocultas, métodos y argumentos en los constructores.
Lo que afecta al programador es el nombre de los ficheros.

Características Comunes:

- **Clases e Interfaces Internas Estáticas**:
  - Definidas dentro de una clase o interfaz con `static`.
  - Acceden a miembros `static` de la clase contenedora.
  - No requieren instanciar la clase contenedora.
- **Acceso a Miembros**:
  - Las clases internas pueden acceder a los miembros `private` de la clase contenedora.
  - Pueden tener modificadores de acceso como `private`, `protected`, y `public`.

Restricciones:

- **Clases Internas Miembro**:
  - No pueden tener miembros `static`.
  - No pueden tener el mismo nombre que la clase contenedora.
- **Clases Internas Locales**:
  - No pueden tener miembros `static`.
  - No pueden ser declaradas con modificadores de acceso (`public`, `private`, etc.).
  - Solo pueden acceder a variables locales `final`.
- **Clases Anónimas**
  - No pueden tener constructores.
  - Usadas para implementar interfaces funcionales o clases con poca funcionalidad.
  - Se compilan como clases globales con nombres como `ClaseContenedora$1.class`.

Uso y Ejemplo Común:

- **Clases Internas Miembro**: Representan objetos internos relacionados con la clase contenedora, como el motor de un coche.
- **Clases Internas Locales y Anónimas**: Son útiles cuando se necesita una clase sólo dentro de un bloque de código, como para manejar eventos en interfaces gráficas (AWT).

Clases e interfaces internas static

Sólo pueden ser creadas dentro de otra clase al máximo nivel, directamente en el bloque de definición de la clase contenedora y no en un bloque más interno.

También es posible definir clases e interfaces internas static dentro de una interfaz contenedora. Este tipo de clases internas **se definen utilizando la palabra static**. Todas las interfaces internas son implícitamente static.
En cierta forma, las clases internas static se comportan como clases normales en el mismo paquete (*package*).

Para utilizar su nombre desde fuera de la clase contenedora hay que precederlo por el **nombre de la clase contenedora y el operador punto (.)**. Este tipo de relación entre clases se puede utilizar para agrupar varias clases dentro de una clase más general. Lo mismo puede decirse de las interfaces internas. Las clases internas static pueden ver y utilizar los miembros static de la clase contenedora.

No se necesitan objetos de la clase contenedora para crear objetos de la clase interna static. Los métodos de la clase interna static no pueden acceder directamente a los objetos de la clase contenedora, en caso de que los haya: deben disponer de una referencia a dichos objetos, como cualquier otra clase.

Otras características importantes son las siguientes:

1. Pueden definirse clases e interfaces internas dentro de interface y clases contenedoras, con las cuatro combinaciones posibles.
2. Puede haber varios niveles, esto es una clase interna static puede ser clase contenedora de otra clase interna static, y así sucesivamente.
3. Las clases e interfaces internas static pertenecen al *package* de la clase contenedora.
4. Pueden utilizarse los calificadores final, public, private y protected. Ésta es una forma más de controlar el acceso a ciertas clases.

Clases internas miembro (no static)

Las clases internas miembro o simplemente clases internas, son clases definidas al máximo nivel de la clase contenedora, **sin la palabra static**. Se suelen llamar clases internas miembro o simplemente clases internas. No existen interfaces internas de este tipo.

Las clases internas no pueden tener variables miembro static. Tienen una nueva sintaxis para las palabras this, new y super, que se verá un poco más adelante.
La característica principal de estas clases internas es que cada objeto de la clase interna existe siempre dentro de un y sólo un objeto de la clase contenedora. Un objeto de la clase contenedora puede estar relacionado con uno o más objetos de la clase interna. Tener esto presente es muy importante para entender las características que se explican a continuación.

Relación entre las clases interna y contenedora respecto al acceso a las variables miembro:

1. Debido a la relación uno a uno, los métodos de la clase interna ven directamente las variables miembro del objeto de la clase contenedora, sin necesidad de cualificarlos.
2. Sin embargo, los métodos de la clase contenedora no ven directamente las variables miembro de los objetos de la clase interna: necesitan cualificarlos con una referencia a los correspondientes objetos. Esto es consecuencia de la relación uno a varios que existe entre los objetos de la clase contenedora y los de la clase interna.
3. Otras clases diferentes de las clases contenedora e interna pueden utilizar directamente los objetos de la clase interna, sin cualificarlos con el objeto o el nombre de la clase contenedora. De hecho, se puede seguir accediendo a los objetos de la clase interna aunque se pierda la referencia al objeto de la clase contenedora con el que están asociados.

Respecto a los permisos de acceso:

1. Las **clases** internas pueden también ser **private** y **protected** (las clases normales sólo pueden ser public y package). Esto permite nuevas posibilidades de encapsulación.
2. Los métodos de las clases internas acceden directamente a todos los miembros, incluso private, de la clase contenedora.

3. También la clase contenedora puede acceder –si dispone de una referencia- a todas las variables miembro (incluso private) de sus clases internas.
4. Una clase interna puede acceder también a los miembros (incluso private) de otras clases internas definidas en la misma clase contenedora.

Otras características de las clases internas son las siguientes:

1. Una clase interna miembro puede contener otra clase interna miembro, hasta el nivel que se desee (aunque no se considera buena técnica de programación utilizar muchos niveles).
2. En la clase interna, la palabra **this** se refiere al objeto de la propia clase interna. Para acceder al objeto de la clase contenedora se utiliza **ClaseContenedora.this**.
3. Para crear un nuevo objeto de la clase interna se puede utilizar **new**, precedido por la referencia al objeto de la clase contenedora que contendrá el nuevo objeto: **unObjCC.new()**. El tipo del objeto es el nombre de la clase contenedora seguido del nombre de la clase interna, como por ejemplo:

   Supóngase como ejemplo adicional que B es una clase interna de A y que C es una clase interna de B. La creación de objetos de las tres clases se puede hacer del siguiente modo:
4. Nunca se puede crear un objeto de la clase interna sin una referencia a un objeto de la clase contenedora. Los constructores de la clase interna tienen como argumento oculto una referencia al objeto de la clase contenedora.
5. El nuevo significado de la palabra **super** es un poco complicado: Si una clase deriva de una clase interna, su constructor no puede llamar a **super()** directamente. Ello hace que el compilador no pueda crear un constructor por defecto. Al constructor hay que pasarle una referencia a la clase contenedora de la clase interna super-clase, y con esa referencia, ref, llamar a ref.super().

Las clases internas pueden derivar (heredar) de otras clases diferentes de la clase contenedora. En este caso, conviene tener en cuenta las siguientes reglas:

1. Las clases internas constituyen como una segunda jerarquía de clases en Java: por una parte están en la clase contenedora y ven sus variables; por otra parte pueden derivar de otra clase que no tenga nada que ver con la clase contenedora. Es muy importante evitar **conflictos con los nombres**. En caso de conflicto entre un nombre heredado y un nombre en la clase contenedora, el nombre heredado debe tener prioridad.
2. En caso de conflicto de nombres, Java obliga a utilizar la referencia **this** con un nuevo significado: para referirse a la variable o método miembro heredado se

utiliza **this.name**, mientras que se utiliza **NombreClaseCont.this.name** para el miembro de la clase contenedora..

3. Si una clase contenedora deriva de una super-clase que tiene una clase interna, la clase interna de la sub-clase puede a su vez derivar de la clase interna de la super-clase y redefinir todos los métodos que necesite. La casuística se puede complicar todo lo que se desee, pero siempre hay que recomendar hacer las cosas lo más sencillas que sea posible.

El uso de las clases internas miembro tiene las siguientes restricciones:

1. Las clases internas no pueden tener el mismo **nombre** que la clase contenedora o package.
2. Tampoco pueden tener **miembros static**: variables, métodos o clases.

A continuación se presenta un ejemplo completo de utilización de clases internas miembro:

Clases internas locales

Las clases internas locales o simplemente clases locales no se declaran dentro de otra clase al máximo nivel, sino **dentro de un bloque de código, normalmente en un método**, aunque también se pueden crear en un **inicializador static** o de objeto.

Las principales características de las clases locales son las siguientes:

1. Como las variables locales, las clases locales **sólo son visibles y utilizables en el bloque de código en el que están definidas**. Los objetos de la clase local deben ser creados en el mismo bloque en que dicha clase ha sido definida. De esta forma se puede acercar la definición al uso de la clase.
2. Las clases internas locales tienen **acceso a todas las variables miembro y métodos de la clase contenedora**. Pueden ver también los miembros heredados, tanto por la clase interna local como por la clase contenedora.
3. Las clases locales pueden utilizar las **variables locales y argumentos** de métodos visibles en ese bloque de código, pero sólo si son **final** (en realidad la clase local trabaja con sus copias de las variables locales y por eso se exige que sean final y no puedan cambiar).
4. Un objeto de una clase interna local sólo puede existir en relación con un objeto de la clase contenedora, que debe existir previamente.
5. La palabra **this** se puede utilizar en la misma forma que en las clases internas miembro, pero no las palabras **new** y **super**.

Restricciones en el uso de las clases internas locales:

1. No pueden tener el mismo **nombre** que ninguna de sus clases contenedoras.
2. No pueden definir variables, métodos y clases **static**.
3. No pueden ser declaradas public, protected, private o package, pues su visibilidad es siempre la de las variables locales, es decir, la del bloque en que han sido definidas.

Las clases internas locales se utilizan para definir **clases Adapter en AWT**. A continuación
se presenta un ejemplo de definición de clases internas locales:

Clases anónimas

Las clases anónimas son muy similares a las clases internas locales, pero sin nombre. En las clases internas locales primero se define la clase y luego se crean uno o más objetos. En las clases anónimas se unen estos dos pasos: Como la clase no tiene nombre **sólo se puede crear un único objeto**, ya que las clases anónimas no pueden definir constructores. Las clases anónimas se utilizan con mucha frecuencia en comparadores, interfaces funcionales y en AWT para definir clases y objetos que gestionen los eventos de los distintos componentes de la interfaz de usuario. No hay interfaces anónimas.

Formas de definir una clase anónima:

1. Las clases anónimas requieren una extensión de la palabra clave **new**. **Se definen en una expresión de Java, incluida en una asignación o en la llamada a un método.** Se incluye la palabra new seguida de la definición de la clase anónima, entre llaves {...}.
2. Otra forma de definirlas es mediante la palabra **new** seguida del nombre de la clase de la que hereda (sin **extends**) y la definición de la clase anónima entre llaves {...}. El nombre de la superclase puede ir seguido de argumentos para su constructor (entre paréntesis, que con mucha frecuencia estarán vacíos pues se utilizará un constructor por defecto).
3. Una tercera forma de definirlas es con la palabra **new** seguida del nombre de la **interfaz** que implementa (sin **implements**) y la definición de la clase anónima entre llaves {...}. **En este caso la clase anónima deriva de Object**. El nombre de la interfaz va seguido por paréntesis vacíos, pues el constructor de Object no tiene argumentos. Para las clases anónimas compiladas el compilador produce ficheros con un nombre del tipo **ClaseContenedora$1.class**, asignando un número correlativo a cada una de las clases anónimas.

A continuación se presenta un ejemplo de definición de clase anónima en relación con AWT, donde **la clase anónima**, que se pasa como parámetro al método addActionListener(), **deriva de Object e implementa la interface ActionListener**.

Las clases anónimas se utilizan en lugar de clases locales para clases con muy poco código, de las que sólo hace falta un objeto. No pueden tener constructores, pero sí inicializadores static o de objeto. Además de las restricciones citadas, tienen restricciones similares a las clases locales.