M5 - Apuntes UF3 (Definitivos)

Índice

Clases y Objetos

Clases

Clase: Una clase describe la estructura de datos que la forman y las funciones asociadas con ella. Dicho de otra manera, es como el plano, esquema o modelo con el que se construyen objetos. Una clase está formada por atributos y métodos. Es el único bloque de construcción, y, por tanto, en Java solo hay clases, no existen datos sueltos ni procedimientos. Otra definición: Abstracción que define un tipo de objeto específico (atributos o datos sobre sí mismo) y operaciones disponibles.

Objeto: Un objeto es un ejemplar concreto de una clase, que se estructura y comporta según como se ha definido en la clase. Su estado es particular e independiente del resto de ejemplares. Otra definición: Entidad que existe en memoria que tiene unas propiedades y unas operaciones disponibles específicas.

Instanciar (Instanciar una clase): Es el proceso de crear un objeto utilizando como modelo la clase.

Principio de encapsulación: Está relacionado con la vista que presentan los atributos de una clase. Una clase puede tener los atributos visibles (públicos), visibles para los hijos (protected) o solo visibles para la clase (private). La vista privada (implantación) encapsula todos los detalles de la estructura de datos y la implementación de los métodos.

Atributos y Estado

Atributo: Son cada uno de los datos que describen una clase. No incluyen datos auxiliares utilizados para una implementación concreta. Normalmente son privados. La consulta de estos se debe hacer mediante métodos públicos (getters y setters). La información que puede guardar puede ser mediante variables primitivas o clases.

Estado: Son los valores que adopta un atributo al ser instanciado. Ejemplo: DNI dni = new DNI (“12345678C”).

Cosas a observar:

- El nombre de un paquete, por convención, empieza en minúscula.
- Para definir una clase, se utiliza la palabra reservada class.
- El nombre de las clases, por convención, empiezan en mayúscula.
- Los atributos llevan una visibilidad que puede ser private, protected o public. En este curso, no pondremos que sean públicos para respetar el principio de encapsulación.

Para crear una instancia de un objeto, se utiliza la palabra reservada **new**. Dos objetos creados con la misma clase, evolucionan de manera distinta, pero parten del mismo estado.

Métodos y Mensajes

Método: Define una operación sobre un objeto. Pueden consultar el estado del objeto o modificarlo. Los métodos disponen de parámetros que permiten delimitar la acción del mismo.

Los tipos de métodos son los siguientes:

- **Accessors:** Consultan o modifican un atributo. Pueden ser **getters** o **setters**.
- **Métodos Genéricos:** Realizan operaciones sobre un conjunto de atributos, calculando valores o modificaciones.
- **Constructores:** Inicializan los atributos.
- **Destructores:** Liberan recursos al final del ciclo.

Mensaje: Relación entre clases.

Métodos Genéricos

Formato:

[public | private] {void | tipo} identificadorDelMètode([tipus param1], [tipus param2],...)

- Primera palabra establece la visibilidad
- Segunda palabra si es tipo void (sin retorno) o algún tipo de retorno (int, double…)
- Tercera palabra, representa el nombre del método.
- Siguientes palabras, parámetros que recibe.

Dentro de los parámetros existen diferentes tipos:

- **Trabajar por valor:** Estos pasan el valor, y se realiza una copia del mismo, que es la que guarda el parámetro. Si el método modifica el parámetro, no afecta a la variable del mensaje entrante. Normalmente, este tipo de "trabajo" es por variables primitivas como (int, double, String…). Ejemplo:

```
int num = 18;
public void suma(int num){
        num = num + 18
// De esta manera, sé cambiaria a 36 dentro del método, pero num = 18 fuera del método, porque no hay ningún retorno, y, por tanto, no realiza nada.

        int resultado = num + 18
        return resultado
// De esta manera, sé cambiaria a 36 fuera del método, porque se está retornando el resultado y, en consecuencia, se le asignaría a la referencia que pasa el valor.
}
```

- **Trabajar por referencia:** Los valores se pasan por referencia, es decir, que todo lo que se modifica en los parámetros dentro del método, se modifican también fuera del método, sin la necesidad del return (aunque se puede agregar). Normalmente, este tipo de "trabajo" es por arrays u objetos como (Integer). Ejemplo:

```
int[] num = {3,4,5}
public void multiplicacion(int[] array) {
        for (int i=0; i< array.length; i++){
                array[i] * 10;
        }
// De esta manera el array cambiaria, tanto dentro del método como fuera.
}
public void multiplicacion(Integer num) {
        num = num + 18;
```

```
        // De esta manera el número cambiaria dentro y fuera del array
        }
```

Si el método es del tipo void, y la variable es primitiva, el valor fuera del método no cambiará. Si es de otro tipo, y tiene un return, el valor si cambiará fuera del método. Por otro lado, con referencia, siempre cambiará (tanto dentro como fuera).

Pregunta de examen: ¿Cómo se puede cambiar el valor de un atributo al ser pasado por parámetro?
Puede ser cambiado al trabajar por referencia, al realizar un return o al pasar un atributo de un objeto directamente al parámetro.

Dentro de un método se pueden definir variables locales creadas en el momento de ejecutarse. Estas solo serán visibles dentro del propio ámbito en las que se define. **Ámbito:** Son los {}.

Getters y Setters

Gramaticalmente, son métodos genéricos. Semánticamente, tienen connotaciones a especificar. Cada atributo tiene dos posibles acciones, leer o modificar. No tiene que ser obligatorias ambas opciones, puede haber solo una de ellas. Tienen una nomenclatura especial:

- Establecer valor a un atributo: set + NomAtribut (setNomAtribut).
- Leer valor de un atributo: get + NomAtribut (getNomAtribut).

Constructores

Son métodos especiales que inicializan los atributos de un objeto instanciado. Establecer constructores no es obligatorio, ya que si no se define uno, existe el constructor por defecto que no posee nada en su interior. Importante mencionar, que no se debe poner, ya que java lo incluye por defecto.

La ejecución de un constructor es implícita a la creación de una instancia. Por otro lado, el constructor no retornada nada, ni es de ningún tipo, es su restricción semántica.

Un constructor puede inicializar otro constructor dentro de sí mismo, es decir, como el siguiente ejemplo:

```
public Punt (){
        this.(0,0);
}

public Punt (int x, int y){
        this.x = x;
        this.y = y;
}
```

Sobrecarga de métodos

Sobrecarga (Overload): Definir dos o más métodos con el mismo nombre, pero con parámetros distintos por cantidad o tipo. La idea es reducir el nombre de identificadores

diferentes para una misma acción con matices diferentes. Se puede realizar en métodos generales como en constructores.
La sobrecarga es un polimorfismo estático (static), porque es el compilador quien resuelve el conflicto del método a referenciar. Si se define un constructor con parámetros, el que no tiene deja de estar disponible.

Objetos

Los objetos ocupan memoria y tienen un estado, que representa el conjunto de valores de sus atributos. Se pueden crear tantos objetos como se desee, todos ellos contaran con una existencia propia y pueden evolucionar por diferentes caminos. Cada objeto creado, cuenta con una referencia única. Para crear un objeto:

```
Punt punto = new Punt(1,2);
Punt puntoVacio = new Punt();
```

Cada vez que se emplea la palabra **new** se instancia un nuevo objeto, por tanto, tienen una referencia distinta. Cuando se comparan objetos, se comparan direcciones de memoria.

Para destruir un objeto, se debe asignar a “null”. De esta manera pierde la referencia en memoria y en consecuencia queda “huérfano”. Cada X tiempo pasará el **garbage collector** que es quien se encarga de retirar lo que no sea necesario para ahorrar memoria (objetos nulos). Principalmente, lo que hacer es revisar si tiene referencia o no. Si fuera necesario, se puede forzar el activar el garbage collector sin necesidad de esperar. Ejemplo: finalize() en el trycatch.

Por otro lado, un array, aunque sea nula, tiene un espacio en memoria asignado, ya que posee una referencia. Únicamente, serán los valores de su interior los que estarán en “null”. Se puede rellenar esta array si se desea.

Mensajes .this

Creado el objeto es posible pasarlos por mensajes o invocar métodos para solicitar una acción. El objeto establece a qué método corresponde el mensaje recibido.

Para enviar un mensaje, se utiliza la notación (“.”). Dependiendo de si se guarda el valor retornado del método, tenemos dos posibilidades dependiendo de si el método retorna algún valor.

- referencia.metodo (param1, param2);
- var = referencia.metodo (param1, param2);

Es recomendable utilizar los getters y setters, y no los propios atributos, desde el main. Eso se utiliza para no romper el principio de encapsulación. Ejemplo:

- punt1.setX (punt1.getX() + 1);
- punt1.x = punt1.x + 1;

This

Dentro de la implementación de un metodo, a veces, es necesario referenciar a la propia instancia a la que pertenece. Para esto, se utiliza la palabra **this**. Ejemplo: this.num = 3;

Es recomendable dotar de **this** todas las referencias de los atributos y métodos de la propia clase, y es obligatorio si se quiere distinguir entre un parámetro y una variable de instancia con el mismo nombre.

Se debe evitar la redundancia de código. Los constructores tienen que hacer uso de los métodos de acceso (getters y setters) para acceder a los atributos, y así, se aplican los filtros y controles pertinentes. Por ejemplo, si se quiere aplicar un filtro para que el atributo sea de tal manera, se hará en el setter, y posteriormente desde el constructor se llamará al setter.

Elementos estáticos (static)

En general, la clase describe el comportamiento de cada uno de sus objetos. También hay cuestiones en la clase y de todos los objetos de su generalidad. Por ejemplo, si se define una constante de una clase, que siempre tenga el mismo valor, será común para todos los objetos, y, por tanto, tiene que estar asociado a las clases y no instancias. **Se pueden definir atributos estáticos y métodos estáticos.**

Atributos estáticos

Para definirlos, se antepone la palabra **static**. Se debe inicializar independientemente del constructor.

Estos atributos están compartidos para todos los objetos de la clase y son accesibles desde cada uno de ellos. Cualquier modificación que se haga, afectará al resto. Como utilizarlo: Classe.atributo

Para acceder al atributo, no se utiliza this, sino que se utiliza el nombre de la clase. Normalmente, las constantes asociadas a la clase son buenas candidatas para hacer estadística. Ejemplos de variable estática:

- private static final int MAX = 1000; → En este caso, no se puede modificar en ninguna parte, ya que se ha agregado el final. (Este es el caso más frecuente).
- private static int MAX = 1000; → se puede modificar el valor, pero este cambiará en todas partes, en las que se haya utilizado.
- private final int MAX = 1000; → No es una variable estática, pero no se puede modificar.

Métodos estáticos

Igual que en los atributos estáticos, se antepone la palabra **static**. En un método estático, no se puede utilizar la palabra this, ya que está asociada exclusivamente a instancias. Se pueden acceder a los atributos estáticos de la misma manera que pueden las instancias, anteponiendo el nombre de la clase. Ejemplo: Classe.atributo

Se permite definir un código estático, para iniciar atributos estáticos. Su ejecución se disparará una sola vez al principio.

Ejemplo de metodo estático: public static int num(){
    return Punt.numero; → Retorna un atributo estático

}

IMPORTANTE: Si se desean modificar atributos de clase, un metodo no puede ser estático, ya que no se podrán acceder a los atributos mediante el this. Por otro lado, no tiene sentido crear un metodo en el que se agregue otro objeto para modificar sus atributos, ya que no estaríamos modificando los atributos del que deseamos. (Esto salió en el examen).

Ámbitos de las variables

Se entiende por **abast, ambit o scope** de una variable, la parte del programa donde la variable es accesible. Los programas de Java están organizados en forma de clases. Cada clase es un paquete. Las reglas de ámbito de Java se pueden cubrir con las siguientes categorías.

Variables miembro (Ámbito de nivel de clase)

Estas variables se declaran dentro de la clase (fuera de cualquier metodo). Se puede acceder directamente a ellas desde cualquier parte de la clase. Se pueden declarar, también, en cualquier lugar de la clase, pero fuera de los métodos. **Por convención, se definen primero los atributos, constructores, métodos propios, getters y setters y por último overrides.**

El acceso especificado de las variables miembro no afecta dentro de la clase. Es decir, aunque una variable este en private, se puede seguir utilizando en la clase, pero en sus hijas no, a no ser que se utilice el getter o setter. Por otro lado, se puede acceder desde fuera de la clase, según la visibilidad que tengan.

Ejemplo: public class MyClass {
    private String mensaje;
}

Modificador	Paquete	Subclase
public	si	si
protected	si	si
default (sin modificador)	si	no
private	no	no

Variables locales (Ámbito a nivel de metodo)

Las variables declaradas dentro de un metodo, tienen abasto a nivel de metodo, pero no se puede acceder fuera del metodo. Las variables locales no existen tras finalizar el metodo. Es muy recomendable utilizar la palabra this para diferenciar entre las variables locales y las de clase, todo y que no sea necesario.

Ejemplo: public int MyMethod {
    String mensaje;
}

Variables de bucle (Ámbito de bloque)

Una variable declarada dentro de un par de llaves {} en un metodo, solo tienen abasto dentro de los claudators.

```
Ejemplo: public static void main(String args[]) {
    int x = 10;
        for (int i = 0; i < 4; i++) {
            System.out.println(x);
        }
    System.out.println(i);  // no funcionará
}
```

Puntos importantes:

- En general, un conjunto de llaves definen un abasto (abast).
- En Java, generalmente podemos acceder a una variable siempre que se haya definido en el mismo conjunto de llaves que el código que estamos escribiendo o en cualquier paréntesis dentro de las llaves donde se definió la variable.
- Cualquier variable definida en una clase fuera de cualquier metodo puede ser utilizada por todos los métodos miembros.
- Cualquier metodo tiene la misma variable local que un miembro, la palabra clave **this** se puede utilizar como referencia de la variable de clase actual.
- Para que una variable siempre se lea después de que termine un ciclo, se debe declarar antes del ciclo.

Herencia

Es uno de los mecanismos fundamentales de la P.O.O. que se utiliza para construir una clase a partir de otra. Una de las funciones más importantes que tiene es promover el polimorfismo.

La herencia relaciona las clases de manera jerárquica. Una clase padre (superclase, clase base) sobre otras clases hijas (subclases o clase derivada). Los descendientes de una clase heredan todos los atributos y métodos que los ascendentes hayan especificado como heredables, además de crear los suyos propios. Cuando decimos que se heredan de la superclase, decimos que se pueden referenciar desde la clase hija. Ejemplo: subClase.metodo01();

En java solo se permite la herencia simple, es decir, que la jerarquía funciona en forma de árbol. Todas las clases heredan de la clase Object, que es la clase padre de todas las clases.

Para especificar que una clase es hija de otra, se utiliza la palabra **extends**. En caso de que no se especifique, hereda de Object.

Para que un atributo o metodo se pueda utilizar directamente con la hija, este debe ser public, protected o friendly. Los miembros que sean private, no se podrán invocar directamente con this, pero su funcionalidad está latente en la herencia.

Para referirnos a los miembros de la clase padre, se realizará mediante **super**. Con “super”, accedemos a los miembros y con “super()” accedemos a los métodos. Los constructores no se heredan, todo y que existan llamadas implícitas a los constructores de la clase padre.

Cuando se crea una instancia de la clase hija, primero se llama al constructor de la clase padre para que se inicialicen los atributos del padre, si no se especifica, se llama constructor sin parámetros. Si se quiere invocar otro constructor, la primera sentencia del constructor de la clase hija, será la llamada del constructor padre. Ejemplo:

```
public PoligonRegular(int numCostats, int longCostats) {
        this.numCostast = numCostats;
        this.longCostats = longCostats;
}

public Quadrat (int costat) throws Exception { (Esta es hija de PoligonRegular)
        super (4, costat);
}
```

Compatibilidad entre variables de clases

Una variable de tipo Object, puede contener cualquier instancia. En cambio, está prohibido que una variable tenga instancias de superclases. Como ejemplo, un PoligonRegular puede tener una instancia de Quadrat, pero no viceversa.

Final

Si se aplica a una clase, no se permite que esta pueda tener subclases. Ejemplo:

```
public final class PoligonRegular{
    (No puede tener hijos)
}
```

Si se aplica a un metodo, no se puede redefinir en las subclases. Ejemplo:

```
public class PoligonRegular{
    public final int calculaArea(){
        (No se puede redefinir en clases hijas)
    }
}
```

La clase Object

Es una clase especial definida por Java. Es una superclase implica de todas las otras clases, lo que quiere decir, que es padre de todas las clases de Java. Eso significa que una variable de referencia de tipo Object puede referirse a otro Objeto de cualquier otra clase. Object, además, define otros métodos, que están disponibles para todas las clases. Estos son: (Los importantes son).

Método	**Sintaxis**	**Propósito**
getClass	class getClass()	Obtiene la clase de un objeto en tiempo de ejecución. Como se muestra: [paquete][clase].[hashCode].
hashCode	int hashCode()	Devuelve el código hash asociado con un objeto invocado.
equals	boolean equals (Object obj)	Determina si un objeto es igual a otro.
clone	Object clone()	Crea un nuevo objeto que es el mismo objeto que se está clonando.
toString	String toString()	Devuelve una cadena que describe el objeto.
notify	void notify ()	Retoma la ejecución de un hilo esperando en el objeto invocado.
wait	• void wait (long timeout) • void wait (long timeout, int nanos) • void wait ()	Espera en otro hilo de ejecución.
finalize	void finalize ()	Determina si un objeto es reciclado (obsoleto en JDK9).

toString()

Proporciona la representación String de un objeto y se utiliza para convertir un objeto a cadena (String). El metodo predeterminado retorna una cadena que consiste en el nombre del paquete + nombre de la clase + @ + hashCode(); **Es muy recomendable reescribir el metodo toString() para obtener nuestra propia representación String del objeto.** Siempre que se intente imprimir cualquier referencia objeto, se llamará internamente al metodo toString().

hashCode()

Java genera un numero único que es el hashCode. Devuelve enteros diferentes para objetos diferentes. No devuelve la dirección del objeto, sino que convierte la dirección interna del objeto en un entero usando un algoritmo.

hashCode en Java es nativo porque no se puede encontrar la dirección del objeto, por eso usa otros lenguajes nativos como C/C++ para encontrar la dirección del objeto.

Devuelve el valor hash que se utiliza para buscar objetos en una colección. JVM utiliza el hashCode() para guardar objetos en estructuras de datos relacionadas con el hashing. Su principal ventaja es que hace más sencilla la búsqueda de objetos.

La anulación del metodo hashCode() se tiene que hacer de tal manera para que cada objeto genere un numero único.

equals(Object obj)

Compara el objeto dado con el objeto **this** (objeto sobre el que se efectúa el metodo). Da una forma genérica de comparar objetos por igualdad. Se recomienda anularlo, para redefinir uno propio.

Diferencias entre equals, == y compareTo:

- **equals** → compara valores, es decir, si un objeto es igual a otro por el valor de sus atributos. Si tienen la misma referencia, si o sí, será true. Si no, puede ser tanto true como false.
- **==** → compara la referencia de ambos objetos, será true si la referencia es la misma, pero si no es la misma, será false.
- **compareTo** → Compara objetos y los ordena. Se podría decir que es como el equals, ya que compara los atributos, pero además proporciona información extra, ya que los ordena y el equals no.

getClass()

Devuelve la clase del objeto en tiempo de ejecución del objeto. Este metodo es estático y final, por tanto, no se puede redefinir.

finalize()

Este metodo se llama antes de que un objeto sea recogido por el garbage collector. Se llama, automáticamente, cuando el recolector determina que el objeto no posee referencias.

Es recomendable redefinirlo para eliminar recursos del sistema, realizar actividades de limpieza y minimizar las perdidas de memoria.

clone()
Devuelve un nuevo objeto que es idéntico al objeto que se desea clonar.

Clases abstractas
Una clase abstracta no se puede instancia, pero sí heredar. Por otro lado, también se pueden definir constructores. Se utilizan para englobar clases de diferentes tipos, pero con aspectos comunes. Se pueden definir métodos sin implementar y obligar a las clases hijas a implementarlos.

Si, añadimos el cualificador **abstract** después de un cualificador public del metodo, lo convertiremos en abstracto, y no hace falta que se implemente, con la firma del metodo basta y además convertirá la clase en abstracta. Importante mencionar, que también se puede añadir a una clase el cualificador **abstract** y se convertirá en abstracta, y no tienen por qué ser todos sus métodos abstractos, algunos pueden serlo y otros no.

IMPORTANTE: Según una clase abstracta, que tenga todos los métodos abstractos, no quiere decir que sea una interface. Una interface tiene todos los métodos abstractos, pero no es una clase abstracta.

La ventaja que aporta utilizar este tipo de clases, es que se pueden trabajar con variables o parámetros, y llamar a métodos comunes sin saber a priori el tipo de objeto concreto. Esto permite en un futuro agregar diferentes objetos, sin cambiar las clases definidas. Esto se llama **polimorfismo**.

Polimorfismo: Es la manera en la que la P.O.O. implementa la polisemia, es decir, un solo nombre para muchos conceptos. Este tipo de polimorfismo, se resuelve en tiempo de ejecución, por ello lo llamamos **polimorfismo dinámico**, a diferencia del **polimorfismo estático** que se resuelve en tiempo de compilación (con el programa parado), que se implementa mediante la sobrecarga de métodos.

Conversión de tipo, Casting
Java permite la conversión de tipo, pero con ciertas limitaciones. Existen varios tipos de conversión, que dependen de la jerarquía de clases.

Conversión hacia arriba
Se trata de poner un objeto de nivel inferior, como uno de nivel superior. Por ejemplo, poner a un profesor sustituto, como profesor. Código:

```
profesor = profesorSustituto;
```

Esto es posible, ya que profesor sustituto tiene más campos y métodos que profesor por si solo, lo que ocurrirá es que perderemos parte de la información, pero nada más.

Conversión hacia abajo

Se trata de poner un objeto de nivel superior, como uno de nivel inferior. Por ejemplo, poner a un profesor como profesor sustituto. Esto no siempre es posible. El objeto de nivel superior admite cualquier forma (es polinómico) de los subtipos. Si el supertipo guarda el subtipo al que queremos realizar la conversión (que pertenecen a la misma herencia), será posible convertirlo utilizando el **Emascaramiento de tipo** *(Emmascarament de tipus)* o **Casting** (significa "modelar"). Código:

profesorSubstituto = (profesorSubstituto) profesor; → es correcto
profesorSubstituto = profesor; → dará un error

Si el supertipo no guarda el subtipo al que queremos convertir, no será posible realizar un Casting, y saltará un error.

El nombre, entre paréntesis, se llama **operador de enmascaramiento de tipo** *(operador d'emmascarament de tipus)* y a la operación se le llama **Casting**. No siempre es sencillo determinar que tipo de objeto apunta a una variable. Hemos de evitar en la medida de lo posible utilizar el casting, ya que si se utiliza mucho, quiere decir que la jerarquía planteada no es del todo correcta. Normalmente, no se utiliza, o se utiliza poco y de manera controlada.

Cuando se utilicen conversiones de tipo Casting, para evitar errores conviene filtrar las operaciones de enmascaramiento asegurándonos de que el objeto padre contiene un hijo del subtipo al que se desea hacer la conversión. Esta verificación utiliza la palabra clave de **instanceof**.

Conversión entre hermanos

No es posible realizar una conversión entre hermanos.

Determinación del tipo de variable con instanceof

instanceof: Sirve para verificar el tipo de una variable. Ejemplos:

if (profesor1 instanceof ProfesorSubstituto) → correcto
if (substituto1 instanceof ProfesorSubstituto) → correcto
if (substituto1 instanceof Profesor) → correcto
if (data1 instanceof Profesor) → incorrecto, ya que no están en la misma herencia.

Solo se puede utilizar en los siguientes casos:

- Comparar instancias que se relacionen dentro de la jerarquía de tipo (en cualquier dirección) pero no objetos que no estén relacionados en una jerarquía. Como en los ejemplos anteriores.
- Solo se puede utilizar **instanceof** cuando está asociado a un condicional. No se puede utilizar directamente en una impresión de pantalla como System.out.print(…).

Sobrescribir métodos en Java, métodos polimórficos

Las variables que representan a un objeto no son el objeto en si mismo, sino referencias al objeto. Cuando hablamos de una variable que apunta a un objeto, se puede distinguir, a causa de la herencia, entre dos tipos:

- Tipo declarado por una variable en el código fuente: es de **tipo estático (static)**.
- Tipo de objeto al cual apunta la variable en un momento dado (tiempo de ejecución): puede ser de tipo declarada o del subtipo del declarado, y es de **tipo dinámico**.

Ejemplo:

```
ProfessorSubstitut profesorsubstitut04 = new ProfesorSubstitut();
Persona p01 = new Persona(); → estático
p01 = profesorsubstitut04; → dinámico
```

En un principio, p01 es de tipo Persona, y, por tanto, es estático. Pero al convertirlo, en la tercera instrucción, pasa a ser un ProfesorSubstituto, y en consecuencia es de tipo dinámico.

El tipo estático es Persona y el tipo dinámico es ProfesorSubstituto, porque estamos creando una variable de tipo Persona, e inmediatamente le estamos asignándole, de manera dinámica, un objeto anónimo de tipo ProfesorSubstituto. Pero igualmente podemos invocar un metodo exclusivo de ProfesorSubstituto sobre p01 para que el compilador se basa en los tipos estáticos para hacer comprobaciones.

El compilador controla los tipos basándose en el tipo estático declarado, ya que no conoce en un momento dado (tiempo de ejecución) si una variable apunta al subtipo. Por esto, no se puede llamar a un metodo del subtipo si la variable corresponde a una clase padre, todo y que la variable apunte a la clase hija. Esto es una limitación relativa, ya que los métodos sobreescritos, si se puede llegar a identificar si corresponde a una clase padre o clase hija.

El control de tipo del compilador se basa en los tipos estáticos, pero en tiempo de ejecución los métodos se ejecutan dando preferencia al tipo dinámico. Esto quiere decir, que Java está constantemente buscando el metodo que corresponda en función del tipo dinámico al cual apunta la variable. Si no lo encuentra, va subiendo por la herencia hasta encontrarlo. Si no lo encuentra, ni en la clase Object, entonces no llegará a compilar. Los métodos en Java son polimórficos porque una misma llamada en un programa puede dar lugar a la ejecución de diferentes métodos según el tipo dinámico de una variable. **Otra forma de expresión de polimorfismo en Java: polimorfismo en variables y métodos.**

Es importante distinguir entre dos momentos temporales en un programa.

- **Compilación:** El compilador realiza una serie de verificaciones y transforma el código fuente en código máquina. Solo conoce un tipo: estático (static) o declarado código fuente.
- **Ejecución:** Una variable puede cambiar de tipo debido al polimorfismo que admite Java. Gracias a la JVM, Java puede identificar tipos dinámicos de las variables que apuntan a los objetos y asignarles un metodo u otro en función del tipo dinámico.

Esto puede tener ciertos riesgos, debido a que si un programa no encuentra un metodo reescrito, irá a llamar al metodo de la clase padre, y, por tanto, puede mostrar o ejecutar el metodo con la misma firma, pero no realizando lo que deseamos. Por otro lado, no veremos problemas de compilación. Es importante reescribir (redefinir) los métodos.

Palabra clave SUPER, refinamiento de métodos

Cuando definimos una herencia, la clase padre cuenta con unos atributos, que las clases hijas heredan, pero, en caso de que estén en private, debido al principio de encapsulación, no se podrán utilizar con el **this**. Para los constructores en las clases hijas, utilizamos el super(), para invocar el constructor de la clase padre. Sí utilizamos la palabra **super** en un metodo, podemos llamar (invocar) a un metodo de la clase padre, en la clase hija. Ejemplo: **super.NombreDelMetodoDeLaClasePadre(parametros si tiene)**. Código:

```
public void mostrarDades(){
        super.mostrarDades();
        System.out.println(this.dadesIniciSubstitucio.getTime().toString());
}
```

Dentro del metodo hijo, se ha incluido una llamada al metodo padre, porque había perdido visibilidad porque lo hemos reescrito. La llamada puede ser efectuada en cualquier parte del código. Es opcional y se debe hacer cuando lo necesitemos. Este proceso se conoce como **refinamiento**. **Refinamiento:** Implica realizar modificaciones y mejoras en una clase existente para lograr un código más eficiente, mantenible y de calidad.

Interface

¿Es posible plantear un escenario donde una clase hereda más de una clase, es decir, herencia múltiple? La respuesta es que si, pero a la vez no. Esto se debe a que una clase puede heredar una única clase, pero puede heredar diferentes interfaces “Interface”. Es como una especie de trampa, porque Java no posee herencia múltiple, pero puede heredar una única clase y diferentes interfaces (que simula la herencia múltiple o implementación limitada).

Interface: Es una construcción similar a una clase abstracta de Java, pero con las siguientes diferencias:

1. En el encabezado utiliza la palabra clave **interface** en lugar de class o abstract class. Ejemplo: public interface NombreInterface{...}
2. Todo metodo es abstracto y público sin necesidad de declararlo, no hace falta poner *abstract public* porque por defecto todos son métodos abstractos y públicos. Por tanto, una **interface** no implementa ninguno de los métodos que declara (solo declara la firma), ya que ninguno tiene cuerpo.
3. Las interfaces no tienen constructor.
4. Una **interface** solo admite campos del tipo “public static final”, es decir, campos de clase, públicos y constantes. No hace falta agregar estas palabras, porque todos los campos se comportaran como tal. static = de clase, final = constante. Las interfaces pueden dar lugar para declarar constantes usadas por diferentes clases en nuestro programa.
5. Una clase puede derivar de una interface, de la misma manera que puede derivar de otra clase. Una clase se implementa (implements), no se extiende (extends).

Una clase puede implementar distintas interfaces de la siguiente manera: public class MyClass implements IRenderizar, Comparable, Cloneable…

En una interface se pueden declarar variables, pero se deben inicializar en la clase concreta que la implemente. Ejemplo:

```
miLista = new List <String>();
        → No se puede, porque no se puede crear un objeto definido en una interfaz.
miLista = new LinkedList<String>();
        → Si se puede.
```

Una interface en Java define un tipo de métodos que están sin implementar y equivalen a la "herencia múltiple" (de clases abstractas). Si una clase implementa una interface puede pasar lo siguiente:

- Implementa los métodos de la interface sobreescribiéndolos (puede ser una clase concreta).
- No implementa los métodos de la interface, obligatoriamente será una clase abstracta y deberá tener la palabra clave abstract en su encabezado.

Una clase puede heredar de otra e implementar diversas interfaces. Primero se coloca la relación de herencia (extends) y luego las implementaciones de interfaces (implements).

¿Como saber si una clase es candidata a ser definida como una interface?

- Si necesitamos algún metodo con cuerpo ya sabemos que no será una interfaz porque todos los métodos de una interface tienen que ser abstractos.
- Si necesitamos que una clase "herede" más de una superclase, estas superclases suelen ser interfaces.
- En otros casos es igual de viable definir una interfaz o una clase abstracta, pero si se da esta situación, optaremos por una interface porque es más flexible y extensible, ya que permite la "herencia múltiple". En cambio, una clase abstracta no.

El interés principal de una interface es el polimorfismo. Ejemplo: reunir una colección de objetivos de tipos interface implementados en diferentes clases.

Para qué sirven las interfaces en Java

¿Si una interface define un tipo (igual que una clase) pero no se provee de ningún metodo, que se gana? La implementación (herencia) de una interfaz no quiere decir que evite la duplicidad de código o favorezca la reutilización de código porque realmente no aportan código.

Si se puede decir, que posee dos ventajas de herencia. Favorece el mantenimiento y extensión de aplicaciones. Esto se debe a que permite la existencia de variables polimorfismo y la invocación polimórfica de métodos.

Un aspecto fundamental de las interfaces es que, de manera genérica, separa la especificación de una clase (que hace) de la implementación (como lo hace). Esto consigue que un programa sea más robusto y tenga menos errores.

Se puede utilizar una u otra implementación para comprobar diferentes rendimientos del programa. Ejemplo: ArrayList es más rápida buscando objetos que no LinkedList. Si declaramos las variables de nuestro programa como List, podemos ir seleccionando entre una u otra cuando sea necesario. Esto provoca que solo nos tengamos que preocupar de especificar el tipo en la declaración de la variable, pero no en el resto del código, y por esto resulta muy útil utilizar interfaces. List tiene todos los métodos abstractos, no tiene constructores y no es instanciable.

La ventaja clara de las interfaces es que permiten declarar constantes que estarán disponibles para todas las clases que les deseemos implementar una interface. Nos ahorra código y el tener que escribir declaraciones constantes de otras clases.

Implementar una interface del API de Java

La API de Java define diferentes interfaces que, aparte de definir tipos, se pueden utilizar para implementar una clase propia en nuestro código. Esto tiene cierta similitud con la redefinición de un metodo, pero no es lo mismo exactamente. Para poder utilizar las interfaces, como el Comparable, habrá que escribirla en el encabezado de la clase. Ejemplo: public class NombreDeLaClase implements Comparable <NombreDeLaClase>

Interesa implementar interfaces de la API de Java, porque nos dotan de instrucciones respecto a características de las clases que las implementen y define que métodos se deben incluir para cumplir esta interface y para qué servirán estos métodos. Si se implementa una interface, lo que hacemos es ajustarnos a la norma. Esto es útil para cuando se trabaja en conjunto, es decir, un programador programa algo, y otro lo retoma. Si se necesita saber como implementar una interface, se debe mirar en la documentación de Java (Oracle).

Muchas API's tiene implementada la interfaz Comparable, que cuenta con el metodo comparableTo().

No se deben conocer todas las clases ni interfaces de Java, pero pueden resultar útiles, ya que algunas suelen ser muy utilizadas. Hay otras que no se usan tanto.

Diferencia entre clase abstracta e interface

La **abstracción** es el proceso de ocultar la implementación y solo proporcionar detalles esenciales al usuario. Esta se consigue a través de clases abstractas e interfaces. Las clases abstractas y las interfaces tiene algunas cosas en común, pero existen diferencias entre ellas.

Clase Abstracta	**Interface**
La palabra clave **abstract** se utiliza para crear o declarar una clase abstracta.	La palabra clave **interface** se utiliza para crear o declarar una interface
Una clase puede heredar las propiedades y métodos de una clase abstracta utilizando la palabra **extends**.	Para implementar una interfaz, se utiliza la palabra clave **implements**.

Una clase abstracta puede tener métodos abstractos y no abstractos. Los abstractos son los que no se tiene prevista ninguna implementación sobre ellos.	Una interface solo tiene métodos abstractos. Solo se puede definir la firma del metodo, pero no su implementación. Después de Java 8 se pueden tener métodos predeterminados y estáticos en interfaces.
Una clase abstracta puede contener variables finales o no (atributos). Pueden también ser estáticos o no.	Una interface solo puede tener miembros estáticos y finales, no se permite ninguno otro.
Una clase abstracta puede implementar una interface e implementar métodos de la interface.	Una interface no se puede extender de otra clase, y no puede anular o implementar métodos de la clase abstracta.
Una clase abstracta puede ampliar otras clases y también puede implementar interfaces.	No pueden extender de otras clases, pero si pueden implementar interfaces.
Java no admite herencias múltiples. Este tipo de clase, tampoco.	Con interfaces proporciona soporte. Esto se debe a que proporcionan una abstracción completa.
Los miembros (atributos) de la clase abstracta pueden ser privados, protegidos o públicos.	Los miembros (atributos) de una interfaz siempre son públicos.

Similitudes entre clase abstracta e interface

Las clases y las interfaces no se pueden instanciar, es decir, no se pueden crear objetos a partir de ellas.

Las subclases (clases hijas) deben invalidar los métodos abstractos definidos en la clase o interfaz abstracta.

¿Cuándo se utiliza una u otra?

Las clases abstractas proporcionan abstracción parcial o total. Las interfaces proporcionan una abstracción completa. Se puede crear una clase padre abstracta para que algunas clases tengan funcionalidades comunes. Las clases abstractas poseen más libertad de acción.

Se prefieren las interfaces cuando se quiere definir una estructura básica. El programador puede construir cualquier cosa con esa estructura. Por otro lado, admiten herencias “múltiples”. Una clase puede implementar múltiples interfaces.

En general, es cuestión de elección y de la tarea que se busca realizar. Ambas son adecuadas para diferentes propósitos, y se deben usar en consecuencia.

Relaciones - Composición y agregación

En el contexto de la programación en Java, los conceptos de composición y agregación se refieren a la relación entre dos clases.

Composición: La composición es una relación fuerte entre dos clases, donde una clase está compuesta por una o más instancias de otra clase. La clase compuesta es responsable de la creación y destrucción de las instancias de la clase componente.

> *Ejemplo: Supongamos que tenemos una clase "Coche" que está compuesta por una instancia de la clase "Motor". El coche tiene un único motor y es responsable de crear y controlar su funcionamiento. Si el coche se destruye, el motor también se destruirá.*

Agregación: La agregación es una relación más débil entre dos clases, donde una clase tiene una referencia a otra clase, pero la clase agregada puede existir de forma independiente. La clase que contiene la referencia no es responsable de la creación o destrucción de la clase agregada.

> *Ejemplo: Supongamos que tenemos una clase "Universidad" que tiene una lista de estudiantes. Los estudiantes existen de forma independiente y pueden ser parte de la universidad, pero no dependen de ella para existir.*

En resumen, la composición implica una relación más fuerte, donde una clase está compuesta por otra clase y es responsable de su ciclo de vida, mientras que la agregación implica una relación más débil, donde una clase tiene una referencia a otra clase, pero la segunda clase puede existir de forma independiente.