TEMA 1 ENCAPSULACIÓN

Clase → Atributos + Constructores + Métodos
Encapsulación → Atributos privados y solo se acceden mediante los métodos para leer y escribir sobre ellos.

Accesos → public, protected, paquete, private
Si no se especifica Java piensa q es public

Constructores → Java proporciona un constructor vacío si no hay otros creados. Cuantos más constructores mejor. Reservan memoria para un objeto mediante new

Valores por defecto → Primitivos 0, Objetos null

Cada atributo debería un getter en la clase, un setter depende del atributo

Java permite la sobrecarga de métodos

El método toString puede sobreescribirse para imprimir lo que se desee acerca del objeto, se puede usar texto formateado. ("Edad: %d".formatted(edad))

TEMA 2 CLASES Y TIPOS DE DATOS

Tipos de datos primitivos → No tienen métodos, ocupan poco en memoría
Clases Wrappers → Envuelven los datos primitivos a una clase con métodos, ocupan más en memoría. Son clases inmutables

Autoboxing → Dato primitivo a Clase (Integer a = 10;)
Unboxing → Clase a Dato primitivo
(suma(Integer a, Integer b); → suma(int aa, int bb){})

Referencia → Posición de memoria de un objeto, las referencias se comparan con ==
Aliasing → El operador = iguala un objeto a la misma posición de memoría, permitiendo modificarlo, lo cual interfiere con la encapsulación.

El aliasing no se debe evitar siempre, pero hay zonas del código en donde es recomendable. Para hacer una copia exacta de un objeto se usa el método clone.

Es importante tener en cuenta las clases mutables e inmutables para el aliasing. Como por ejemplo a la hora de hacer un getString o un getArrayList.

El almacenamiento se divide en pila y heap

Pila → Referencias a los objetos, datos primitivos (args, returns, var locales) y métodos

Heap → Objetos

El recolector de basura se encarga de eliminar de memoría aquellas referencias que apuntan a null en el heap.

Los objetos que más tiempo llevan en memoría tienen más probabilidades de seguir en memoría por localidad

El recolector de basura se divide en 3 gens
Young gen → Objetos new y sección pequeña. Cuando se llena RB minor muy rápida
Old gen → Todos los obj, sección grande. Cuando se llena RB major, más lenta.
Permanent gen → Clases y métodos usados

El recolector de basura en Young gen se resume en:

1. Objetos entran al edén, cuando se llena se procede a eliminar los no referenciados y los que sobreviven pasan a S0 con edad 1
2. Cuando se vuelve a llenar el edén los supervivientes pasan a S1 y los objetos de S0 pasan a S1 también incrementando su edad
3. Se repiten el paso 1 y 2 hasta que promocionan a old gen al alcanzar una edad concreta

Para asegurar que una rb minor o major no elimine un objeto que está apuntando a null se utilizan referencias débiles y suaves, además de phantom.

Ref débiles → Apuntan a referencias fuertes y el gc las borra solo si apuntan a null, sirve para cachés

Ref suaves → Apuntan a referencias fuertes, pero el gc no las borra si aputan a null sino solo cuando realmente se necesite memoria

Phantom refs → Se trata de una cola de referencias de un objeto

La clase String es inmutable, y existe una zona en el heap llamada string pool en la que 2 o más strings iguales apuntan al mismo objeto string. Ej:

String s1 = “abc”

String s2 = “abc” s1 y s2 apuntan al mismo objeto, solo 1 reserva de memoria

Con new se obliga a realizar una reserva de memoria
String s1 = “abc”
String s2 = “abc”
String s3 = new String(“abc”) → Ahora son 2 reservas y s3 apunta a otra posición

Otro ejemplo
String s1 = “abc”
String s2 = “abc”
String s3 = s2 + “d”

3 reservas de memoria → Una para s1/s2, otra para “d” y otra para s3

Con otras Clases como Integer también puede ocurrir si están en caché

2 objetos son iguales si tienen la misma referencia (==) o si son de la misma clase y el valor de todos sus atributos son iguales. Para comprobar si 2 objetos son iguales usamos el método equals.

El método equals se debe sobreescribir en muchas ocasiones para poder decidir los criterios que deciden si 2 objetos son iguales o no. Ej:

```
@Override
public Boolean equals(object o){
        if(this==o){ return true;}
        if(this==null){ return false;}
        if(getClass() != o.getClass() ){ return false;}
        ClaseA a = (Clase A) o
        if(!a.atributo.equals(o.atributo)){ return false;}
        return true;
}
```

Es importante para métodos como contains o remove

TEMA 3 CONJUNTOS DE DATOS

Array → Array convencionales, tienen tamaño fijo e índices. Se deben reservar memoria con new. No permiten borrar elementos, pues no permite nulos

Collection → Interfaz en la cual los objetos no están ordenados, no se pueden recorrer con un índice pero sí con un for-each o un iterator.
Métodos: add, remove, contains, iterator, isEmpty

Para crear un iterator a partir de una collection → Iterator<Clase> itC = C.iterator()
Alterar una Collection en un for-each produce errores (insertar y borrar)

Con un iterator podemos recorrer una collection y borrar algunos de sus elementos, métodos Iterator: hasNext, next, remove

Un iterator solo se puede usar una única vez y las veces que llame a next es equivalente al nº de veces que se mueve el puntero

List → Interfaz donde los elementos están ordenados por un índice en la posición en la que entraron. (Recordatorio de q list no es instanciable)

Métodos: get, set, remove

ArrayList → Clase que implementa List, permite redimensionar la lista. Al instanciar un ArrayList se reserva memoria para 10 objetos, al ocupar las 10 posiciones se ofrecen 10 nuevas. Puede contener nulos y es una clase mutable (Aliasing).

Set → Interfaz en la que los elementos no se repiten, solo hay un valor nulo. (Depende de hashcode y de equals)

HashSet → Implementación de Set, mediante claves hash

Map → Interfaz donde los objetos se guardan con un par de clave valor. Los valores pueden repetirse pero las claves no. Algunas implementaciones admiten valores y claves nulas. (Depende de equals)

Métodos: put, get, containsValue, containsKey, remove, values, keySet, entrySet

HashMap → Clase que implementa Map, mediante claves hash. Primero se comprueba el hashcode, si es igual se recurre al método equals

Se asume que aunque dos objetos no sean iguales, los hashcode pueden ser iguales; pero si dos objetos son diferentes, entonces los hashcode deben de ser diferentes

El método hashCode se puede y debería sobreescribirse para tener una buena función hash

TEMA 4 HERENCIA

La herencia es clave para la reutilización de código desde clases base a clases derivadas. Establece una relación de "ser" dentro de una jerarquía (Una moto es un vehículo). Para que una clase/interfaz herede de otra se utiliza extends

public class B extends A{}

Java no permite la herencia múltiple entre clases (Una Clase no puede tener 2 padres), lo que resulta en una jerarquía en forma de árbol.

En la herencia los métodos generales y la mayor parte de ellos deben implementarse lo más alto posible, y dejar los más específicos para las clases finales. Un problema de la herencia es que la jerarquía puede hacerse demasiado grande y que la modificación de métodos de las clases superiores interfieran con las de clases inferiores.

En herencia se presentan clases abstractas, que son aquellas las cuales tienen como mínimo un método abstracto o simplemente no son instanciables.
[!] Eso no quita que pueda tener constructores

Método abstracto → Método que no es posible implementar en las clases superiores y se delega su implementación a las clases inferiores, ejemplo:

public float calcArea();

Otra opción frente a la herencia es la composición, donde el objeto delega responsabilidades a los demás objetos que lo componen, de forma que es más simple la gestión de código, pero más escritura, pues no se reutiliza código. Aquí se dice que la clase "tiene" algo. (Un coche tiene un volante, donde la clase volante tiene las responsabilidades de girar el coche)

En Java una clase hereda todo acerca de su clase base, pero se restringe la accesibilidad en función del tipo de acceso que tenga en la clase base. Se heredan atributos privados, aunque no se tenga su acceso y se tenga que hacer mediante getters. (Tiene ojo, pero tiene un parche encima)

Para llamar a un constructor de una clase inferior es necesario antes llamar al de la clase padre anterior mediante super. Si no tiene argumentos ninguno de los 2 Java saca el constructor por defecto.

Super permite reutilizar código para llamar a métodos de la clase padre. (Solo un nivel arriba en la jerarquía)

Los métodos de una clase base pueden ser sobreescritos mediante @Override, cambiando así su comportamiento en clases inferiores

Lo más común es que clases abstractas hereden de otras abstractas y las finales de abstractas (pues las finales no pueden tener clases derivadas)

final = No se puede modificar

Un atributo primitivo final indica que una vez establecido un valor, este no puede cambiar. Usado para constantes (public final static)

Un atributo objeto final no puede reasignarse / cambiar posición de memoría

Un método final no puede ser sobreescrito

Una clase final indica que no puede tener descendencia, por lo que debería tener todos los métodos abstractos implementados y es instanciable

static = Disponible desde el principio del programa, no se necesita instanciar para usarlo. Se llama con Clase.A

TEMA 5 POLIMORFISMO

Upcasting → Clase inferior se comporta como una clase superior

Si una clase inferior tiene métodos heredados y los sobreescribe, entonces tendrán el comportamiento de la clase inferior aunque se haga upcasting, pues es el primero que se ha cargado en memoría

Esto permite que cada objeto pueda ejecutar sus métodos según a la clase que pertenezcan dentro de la jerarquía

El upcasting permite almacenar objetos de distintas clases en conjuntos de datos mutables como ArrayLists o Hashmaps si comparten la misma clase/interfaz padre

Downcasting → Clase superior se comporta como una clase inferior

El downcasting puede dar problemas pues puede que se realice el cast a un objeto del cual no coincide el tipo (ClassCastException) → Para evitarlo uso de instanceof

TEMA 6 INTERFACES

Interfaz → Contrato donde se establece qué métodos debe implementar la clase que la implemente con implements.

Una interfaz se compone de:
-Constantes (public static final)
-Métodos abstractos (no se pueden implementar y deben ser publicos)
-Métodos estáticos (Disponibles desde el inicio del programa)
-Métodos por defecto (Métodos implementados que heredan las clases implementadas para ahorrar código, pueden ser sobreescritos)
[!] NO hay constructores y NO es instanciable

[!] Una interfaz NO es una clase abstracta

Se permite la herencia múltiple entre interfaces
(public interface I extends H,J {H y J interfaces})
Y la implementación múltiple entre una clase e interfaces
(public class A implements B,C {B y C interfaces})
Una clase no puede extender una interfaz

[!] extends = Heredar contrato
[!] implements = Implementar/Aceptar contrato

Métodos estáticos → Se heredan entre clases, pero no entre interfaces. Además de que no pueden ser sobrescritos (no tiene sentido hacerlo, para ello se crea uno nuevo)

Métodos default→ Se heredan en las clases que implementan la interfaz y pueden ser sobreescritos. Para solucionar un conflicto entre métodos default en una clase al implementar 2 métodos default con el mismo nombre, se puede elegir sobreescribiendo el método y eligiendo la implementación con super:
ClaseB.super.met()

TEMA 7 EXCEPCIONES

La corrección de errores y la implementación de los métodos debe ir por separado.

Hay 2 tipos de excepciones, las chequeadas y las no chequedas, todas provienen de la clase Throwable
Chequeadas → Clases de error que se indica de forma explícita que puede ocurrir
No chequeadas → No se comprueban en tiempo de compilación donde sucede el error como NullPointerException

Para crear una excepción se crea una clase que extienda a Exception que tiene los métodos de Throwable getMessage y printStackTrace

En los métodos de una clase se debe comprobar si ocurre alguna condición para lanzar la excepcion por ejemplo con la division entre 0:

```
public float div (float a, float b) throws Div0Exc{
        if(a/b!=0){
                return a/b;
        }else
                throw new Div0Exc("Error división entre 0"); → Instancia
        }
}
```

En la clase principal ahora con try, intentamos capturar la excepción, si es así tenemos que tratarla con un bloque catch

```
Aritmetica a = new Aritmetica();
try{
        float t=5;
        t = a.div(5,0);
} catch(Div0Exc d){
        d.getMessage()
        {Tramiento error}
}
```

Se puede realizar un único bloque try para varias excepciones

El orden de los bloques catch es importante, pues si tenemos una jerarquía de excepciones y ponemos primero la más general las demás serán inalcanzables

Se puede realizar multi-catch, pero hay que estar atento con la jerarquía de excepciones, pues tienen que estar al mismo nivel de la jerarquía

Trás los bloques catch se puede poner un bloque finally, por el que siempre pasa el código después de haber ocurrido una excepción, suele usarse para liberar recursos.