**02/09/25 – BACKEND DE APLICACIONES**

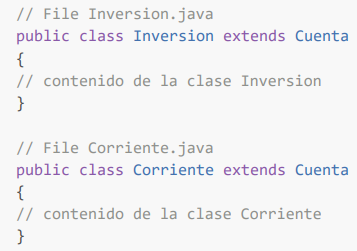
**HERENCIA Y POLIMORFISMO**

**HERENCIA**: mecanismo por el cual se define una clase B en términos de otra clase A definida, de forma en que la B obtiene todos los miembros definidos en la A sin necesidad de una Re-declaración. A, hace que la clase B incluya todos los miembros de A como propios (a los cuales podrá acceder en mayor o menor medida de acuerdo con el calificador de acceso [public, private, protected, "default"] que esos miembros tengan en A). La clase de la cual se hereda es llamada superclase y las que heredan son clases derivadas/subclases.

La jerarquía de clases es un conjunto de clases relacionadas por herencia. La clase base es de la que nace la jerarquía y esta reúne en ella características que son comunes a todas las clases de la jerarquía, y que por lo tanto todas ellas deberían compartir sin necesidad de hacer redeclaraciones de esas características. Existen dos tipos de herencia:

* Herencia Simple: Java solo permite herencia simple (una superclase directa). Sólo puede existir una flecha que parta desde la clase derivada hacia alguna super clase.
* Herencia múltiple: no está soportada. : Si se siguen reglas de herencia múltiple, entonces una clase puede tener tantas super clases directas como se desee.

Relación de uso: implica que un objeto de un objeto usa un objeto de otra. Pero una relación de herencia implica que un objeto b de una clase B, es a su vez objeto de la clase A. Se debería usar herencia cuando B es un A (p. ej., Cliente es un Persona); si solo “usa” otra clase, se elige composición.

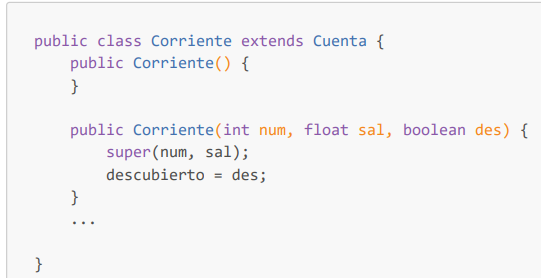
Forma de declaración y uso de constructores en un jerarquía: para indicar que una clase hereda de otra, se usa en Java la palabra reservada extends al declarar la clase derivada, nombrando luego de ella a la super clase de la misma.

Si al declarar una clase no se indica si la misma deriva de otra (o sea, no se escribe extends), entonces el lenguaje asume que esa clase hereda desde la clase Object, que es la base de toda la jerarquía de clases de Java. (public class Persona = public class Persona extends Object).

La clase Object provee los siguientes métodos (los nombramos sólo a los efectos documentales): toString() – finalize() – getClass() – clone() – equals() – hashCode() – wait() – notify() – notifyAll(). Y todos estos métodos están entonces presentes en toda clase Java predefinida o definida por el programador.

* Si un programador no incluye ningún constructor en una clase, el compilador incluirá un constructor nulo en el código de byte de la clase (archivo con extensión .class), y por ello cualquier creación de instancias invocando al mismo, compilará.
* Si el programador incluye algún constructor que NO sea el nulo, entonces el compilador no incluirá el constructor nulo en el .class, y cualquier intento de crear una instancia sin enviar parámetros a su constructor no compilará.

Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.Al invocar un constructor de una clase derivada, Java espera que ese constructor invoque a su vez a algún constructor de la super clase. palabra reservada super puede usarse para invocar explícitamente a un constructor de la super clase. Si el constructor de la clase derivada no incluye una llamada explícita a algún constructor de la super clase, entonces Java impone una llamada al constructor por default de la super clase. Si ese constructor no estuviera en la super clase y no hubiera sido insertado automáticamente por el compilador, se provocará un error de compilación. Por ese motivo la inclusión del constructor default o nulo es recomendable.

En resumen: super te deja referenciar a la superclase (la clase padre) desde una subclase. Se usa sobre todo para:

* Llamar a un constructor del padre: super(...)
* (Opcional) Reusar una implementación del padre en un método sobrescrito: p.ej., super.toString() y luego agregar lo propio.

Reglas Clave

* En un constructor de la subclase, la llamada a la superclase debe ir como primera línea. Si no se escribe, el compilador inserta super() (llamando al constructor por defecto del padre).
* Si la superclase no tiene constructor por defecto y la subclase no invoca explícitamente otro constructor válido del padre con super(...), la compilación falla.
* La sobrecarga del constructor del padre se elige por la lista de parámetros pasada a super(...).
* Al sobrescribir un método, puede invocarse la versión del padre (por ejemplo super.toString()) y agregar información propia.
* En un mismo constructor no puede combinarse this(...) y super(...); solo una llamada y debe ir primero.

Si la clase derivada incluye métodos propios, entonces esos métodos no pueden ser usados por la super clase: sólo existen en la derivada. Atención a las diferencias:

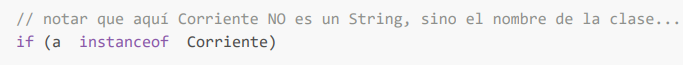
* Redefinir un método es la acción que tiene lugar en una clase derivada, para volver a definir un método heredado desde la super clase pero tal que su especificación no es adecuada a la derivada. Para redefinir un método, la clase derivada debe volver a escribir su cabecera pero TAL CUAL COMO FIGURA DECLARADA EN LA SUPER CLASE, excluyendo eventualmente el calificador de acceso. Al redefinir un método, en la clase derivada valdrá entonces el redefinido, y no el heredado.
* Sobrecargar un método es la acción que tiene lugar en una clase cualquiera, para agregar distintas versiones del mismo método en esa clase (caso: los constructores). Para sobrecargar un método, debe mantenerse el mismo nombre pero debe modificarse la forma de la lista de parámetros del mismo. No se toma como sobrecarga la diferenciación en el tipo de salida del método.

Si en la super clase los atributos o métodos son privados, entonces son inaccesibles incluso para las clases derivadas. En cambio, si en la super clase los atributos fueran protected, serían accesibles para las clases derivadas como si fueran públicos, aunque no lo serían para clases que no deriven de esa super clase.

Una vez creada la jerarquía, podemos crear instancias de cualquiera de las clases en ella. Al invocar métodos, Java seguirá la pista del método invocado hasta la definición hecha en la clase del objeto, siempre y cuando ese método aparezca definido en la base de la jerarquía. Un objeto de una clase más arriba en la jerarquía, no puede invocar un método cuya primera definición aparezca en alguna clase más abajo. Pero lo contrario es válido: un objeto de una clase más abajo, puede invocar métodos definidos más arriba.

POLIMORFISMO: es una referencia del tipo de la clase base puede apuntar a objetos de cualquiera de sus subclases; al invocar métodos, se ejecuta la versión del objeto real (enlace dinámico). Ej.: Cuenta x puede referir a Cuenta, Inversion o Corriente, y x.toString() llamará a la implementación correspondiente al objeto concreto.

Situaciones donde posiblemente se querrá determinar la clase a la que pertenece el objeto apuntado por una referencia polimórfica. En ese caso, tenemos dos vías:

* Usar el operador instanceof. La siguiente condición determina si el objeto referido por a es de la clase Corriente: 
* Usar el método getClass() que viene heredado desde Object. Este método devuelve un objeto de la clase Class (la cual está definida en java.lang). Los objetos de la clase Class representan a las clases de los objetos de la aplicación en curso. Si tenemos dos referencias (polimórficas o no) a y b, la siguiente condición determina si los objetos apuntados son de la misma clase "real":



**Polimorfismo con Object**

En Java, Object es la raíz de la jerarquía de clases. Por ello, una referencia de tipo Object puede apuntar a instancias de cualquier clase, y al invocar un método sobrescrito como toString(), se ejecuta la implementación correspondiente al tipo real del objeto. Esto habilita el máximo grado de polimorfismo.

**Arreglos Polimórficos**

Un arreglo declarado con el tipo de la superclase (por ejemplo, Cuenta[]) puede almacenar objetos de la superclase y de cualquiera de sus subclases (Inversion, Corriente). Cada casilla actúa como referencia polimórfica, permitiendo recorrer y operar la colección de manera uniforme.

**Invocación de Métodos Comunes**

Si un método está definido en la superclase (p. ej., retirar() en Cuenta), puede invocarse sobre cada elemento del arreglo polimórfico. En tiempo de ejecución, la JVM resuelve dinámicamente qué implementación concreta ejecutar según el tipo real del objeto contenido en cada posición.

**Métodos Exclusivos de la Subclase**

Cuando se necesita acceder a un método que existe sólo en la subclase (por ejemplo, actualizar() en Inversion), resulta obligatorio verificar el tipo real con instanceof y luego realizar el *downcast*. De ese modo, la referencia adopta el tipo específico y el compilador habilita la invocación del método exclusivo.

**Comparación de Clases Concretas**

Para filtrar u operar únicamente sobre los objetos de una clase exacta, se utiliza getClass() y se compara con la clase de referencia deseada. Esta técnica permite separar instancias por su tipo concreto, más allá del polimorfismo general de la jerarquía.

**Clases Envoltorio (Wrappers) de Primitivos**

Las clases envoltorio (Integer, Float, Character, etc.) representan valores primitivos como objetos inmutables. Este mecanismo posibilita usar valores primitivos dentro de colecciones genéricas y aprovechar conversión a/desde String mediante métodos como parseInt, parseFloat y sus contrapartes de extracción (intValue, floatValue, etc.).

**Autoboxing y Unboxing**

Desde Java 5, el compilador realiza conversiones automáticas entre tipos primitivos y sus wrappers. Así, asignaciones como Integer i = 23; o int j = i; se traducen internamente a creaciones de objetos y extracciones de valor, simplificando el código y reduciendo la verbosidad sin perder seguridad de tipos.

**Idea Central**

En polimorfismo, el **tipo de la referencia** determina **qué se puede invocar** en tiempo de compilación, mientras que el **tipo real del objeto** determina **qué implementación se ejecuta** en tiempo de ejecución. Esta separación favorece código extensible, menos acoplado y más fácil de mantener.

**Modificador static: miembros de clase vs. de instancia**

Se explica que un miembro **no estático** pertenece a **cada objeto**: cada instancia posee su propia copia de los atributos y sus métodos operan sobre esa copia. En cambio, un miembro **static** pertenece a la **clase**: existe **una única copia compartida** por todas las instancias y se carga en memoria al cargarse la clase, antes de crear objetos. Por ello, cambiar un atributo estático desde cualquier instancia afecta el **mismo valor compartido** por todas.

**Uso típico de static: contador de instancias**

Para contabilizar cuántos objetos se han creado, se sugiere un **atributo static contador** e incrementarlo en el **constructor**. De esta manera, no hay múltiples contadores (uno por objeto) sino **un único contador global a la clase**, cuyo valor refleja la cantidad total de instancias creadas. Resulta natural acompañar esto con un **método static** (por ejemplo, getInstanceCounter()) para consultar el valor sin necesidad de instanciar.

**Métodos static vs. métodos de instancia**

Se aclara que un **método static** existe aunque **no haya instancias** y por eso puede invocarse con **NombreDeLaClase.metodo()**. Un **método de instancia** requiere un objeto concreto para ejecutarse y no puede invocarse desde la clase. Aunque Java permite llamar a un método static vía una referencia de objeto, se considera **estilísticamente incorrecto**; la invocación recomendada es **por la clase**.

**Restricciones de acceso desde un método static**

Se detalla que un método static **no puede** acceder a **atributos de instancia** ni invocar **métodos de instancia** de la misma clase, porque estos dependen de un objeto que podría no existir. En consecuencia, dentro de un método static solo deben usarse **miembros static** de la clase o, si se necesitan datos de instancia, primero **crear** o **recibir** una instancia y acceder a través de ella.

**main como método static**

Se recuerda que el método **main** es static para que la **JVM** pueda ejecutarlo **sin instanciar** la clase contenedora. Por el mismo motivo, desde main no se puede acceder directamente a miembros **no estáticos** de esa clase; para hacerlo, debe **crearse** una instancia o convertir los miembros necesarios a static (cuando conceptualmente tengan sentido como miembros de clase).

**Modificador final: variables, métodos y clases**

Se establece que **final** indica **inmutabilidad en la referencia o la definición**:

* En **variables**: la **referencia** no puede reasignarse (si es un objeto, su estado interno puede cambiar salvo que también sea inmutable); las **constantes** se declaran típicamente public static final.
* En **métodos**: **no pueden ser sobrescritos** en subclases, fijando el comportamiento.
* En **clases**: **no pueden heredarse** (por ejemplo, String es final).  
  Se advierte que **final y abstract son incompatibles**: lo abstracto exige ser sobrescrito; lo final lo impide.

**Modificador abstract: clases y métodos**

Se explica que una **clase abstract** **no puede instanciarse** y sirve como **plantilla** parcial de comportamiento. Puede contener métodos **con implementación** y métodos **abstract** (sin cuerpo). Cualquier subclase **concreta** debe **implementar todos** los métodos abstractos pendientes; de lo contrario, también debe declararse abstract. Se recuerda que un **método static no puede ser abstract**, porque lo abstracto depende del **polimorfismo de instancia** y lo estático pertenece a la clase (no participa del despacho dinámico).

**Relación con polimorfismo y buenas prácticas**

Se enfatiza que los **métodos estáticos no son polimórficos**: no se **sobrescriben**, sino que se **ocultan** (method hiding). Por eso, para comportamiento polimórfico se utilizan **métodos de instancia** (posiblemente declarados abstract en la base y **sobrescritos** en las subclases). Asimismo, se recomienda reservar static para **estado o funciones verdaderamente globales a la clase**, final para **proteger contratos** o **constantes**, y abstract para **definir interfaces de comportamiento** sin forzar implementación en la superclase.

**Modificador abstract: propósito y alcance**

Se indica que una **clase abstracta** modela la **esencia común** de una familia de clases y **no debe instanciarse**. Sirve para agrupar comportamientos y habilitar **herencia y polimorfismo**. Por ejemplo, Cuenta es abstracta porque una “cuenta genérica” carece de datos operativos completos; en cambio, Inversion o Corriente son **concretas** y sí se instancian. Declarar una referencia a una clase abstracta es válido; lo que está prohibido es **crear objetos** de ella con new.

**Métodos abstractos: contrato obligatorio**

Cuando parte del comportamiento depende de cada subclase, la clase abstracta puede declarar **métodos abstractos** (firma sin cuerpo). Esto establece un **contrato** que las subclases concretas deben **implementar**. Si una subclase no implementa todos los métodos abstractos heredados, también debe declararse abstracta. En el ejemplo, Cuenta declara public abstract void retirar(float imp); y Inversion **provee la implementación** específica.

**Polimorfismo sobre clases abstractas**

El polimorfismo sigue activo aunque la base sea abstracta: una referencia de tipo Cuenta puede apuntar a instancias de Inversion o Corriente y ejecutar el **método adecuado** según el **tipo real**. Así, “se programa contra el contrato” definido por la clase abstracta, mientras las subclases aportan la variación de comportamiento.

**Modificador final: cierre de extensión y cambio**

El modificador final expresa **cierre**:

* En **clases**, impide la **herencia** (no se pueden derivar subclases).
* En **métodos**, impide la **sobrescritura** (fija la implementación).
* En **variables** (atributos, parámetros o locales), impide **reasignar** la referencia o el valor una vez establecido.  
  Es semánticamente **opuesto** a abstract: lo abstracto **exige** sobrescribir; lo final **prohíbe** sobrescribir.

**Constantes y reglas de inicialización (static, final, static final)**

Se detalla que un **atributo static** es **único y compartido** por todas las instancias; su valor puede cambiar mientras no sea final. Un **atributo final** (no estático) debe asignarse **una sola vez** (en la **declaración** o en el **constructor**, pero no en ambos). Un **atributo static final** actúa como **constante de clase** y **debe inicializarse en la declaración**; por convención suele ser public, dado que no existe riesgo de modificación y su uso es global. Si se marca final a **parámetros o variables locales**, cualquier intento de reasignación provocará error de compilación.

**Interfaces y polimorfismo “horizontal”**

Para habilitar operaciones comunes entre clases **sin parentesco** de herencia, se emplean **interfaces**. Una clase **implementa** una o varias interfaces y queda obligada a **definir** sus métodos. Comparable es el ejemplo canónico: expone compareTo para ordenar objetos. De este modo, se logra polimorfismo entre tipos dispares que comparten la **misma interfaz**, superando la limitación de herencia simple.

**Arreglos y referencias genéricas: Object vs. interfaz**

Un arreglo de Object permite almacenar **cualquier** objeto, pero para invocar comportamientos **específicos** obliga a **chequear tipo** y **castear**. Si lo que se necesita es **comparar u ordenar**, conviene declarar el arreglo con el **tipo de interfaz** pertinente (por ejemplo, Comparable[]) y así **evitar downcasting** y condicionar en tiempo de compilación qué operaciones están disponibles.

**Cadenas: String (inmutable) vs. StringBuilder**

La clase **String es final e inmutable**: su contenido no cambia; cada concatenación crea **nuevas instancias**, lo que puede ser **ineficiente**. Para construir cadenas de manera intensiva, se recomienda **StringBuilder** (o StringBuffer en ambientes multihilo), que permite **modificar** el contenido y luego obtener un String con toString(). Para comparar cadenas, String implementa **Comparable** (compareTo) y provee **equals** para igualdad exacta.

**Interacción entre static, abstract y polimorfismo**

Se recuerda que los **métodos static no son polimórficos**: pertenecen a la **clase**, no a la instancia, y se **ocultan** (no se sobrescriben). Por tanto, los contratos polimórficos deben expresarse con **métodos de instancia** (abstractos o concretos) en jerarquías o interfaces. El método main es static porque debe poder ejecutarse **sin instanciar** la clase contenedora.

**Anexo: Lombok en jerarquías**

Al usar Lombok en herencia, se advierte que por defecto **no incluye campos de la superclase** en los métodos generados (toString, equals, hashCode). Para incluirlos, se debe especificar **callSuper = true** en anotaciones como @ToString y @EqualsAndHashCode. Esta decisión debe alinearse con qué campos del padre forman parte de la **identidad** o la **representación** del objeto.

**Lista sobre Arrays**

Una lista necesita crecer dinámicamente, pero los arrays en Java tienen tamaño fijo. Para resolverlo:

* Se usa un Object[] interno como almacenamiento.
* Un atributo count lleva la cuenta de cuántos elementos hay.
* Un método asegurarCapacidad() crea un nuevo array más grande y copia los elementos cuando se llena.

Con esto podés simular el comportamiento de un ArrayList. Lo importante es aprender a encapsular y no exponer el array al usuario.

Iteración sobre la Lista

Para recorrer los elementos sin depender de la estructura interna se usa el patrón Iterator.

* Versión trivial: métodos en la misma clase (iniciarIterador, hayMas, getActual, siguiente). Solo permite un recorrido.
* Versión mejorada: clase interna IteradorLineal que puede acceder a los atributos privados de la lista, soportando múltiples recorridos a la vez.
* Versión idiomática en Java: implementar Iterable<E> y declarar un Iterator<E> con hasNext(), next() y remove(). Esto permite usar for-each.

Lo que debés aprender es cómo crear tu propio iterador interno y cómo funciona el mecanismo fail-fast (cuando la lista cambia durante un recorrido, lanza excepción).

**Clases Internas y Anónimas**

Las clases internas permiten declarar un iterador dentro de la propia lista, con acceso directo a sus campos privados Las clases anónimas (o inline) son útiles para crear un objeto rápido de una interfaz o clase abstracta sin necesidad de declarar una clase aparte. Se usan para tareas simples, como callbacks.

**Clases Genéricas (Generics)**

Una limitación de usar Object[] es que se pierde el control de tipos y hay que hacer casting. Con generics (class ListaArray<E>) podés:

* Declarar que la lista maneja un tipo E.
* Evitar casting en cada get().
* Asegurar homogeneidad.
* Incluso restringir E (ejemplo: E extends Comparable) si querés ordenar elementos.
* Framework de Colecciones en Java
* El paquete java.util provee interfaces y clases listas para usar:
* Collection: raíz de List, Set, Queue, Deque.
* Map: pares clave–valor.
* ArrayList: lista sobre array, acceso O(1).
* LinkedList: lista doblemente encadenada, acceso secuencial O(n).

Tenés que entender cuándo conviene cada una: ArrayList para accesos rápidos por índice, LinkedList para muchas inserciones o borrados en medio.

**Clases Abstractas de Apoyo**

Java provee plantillas que facilitan implementar colecciones:

* AbstractList: para listas basadas en array.
* AbstractSequentialList: para listas ligadas.Al extenderlas, solo debés sobreescribir algunos métodos clave (get, size, add, remove) y manejar modCount para soportar iteradores fail-fast.

**Paquetes, import y JA**

Las clases deben organizarse en packages (carpetas). Con import se accede a otras librerías. Finalmente, el código puede empaquetarse en un .jar (ejecutable o librería) para distribuirlo.

**Formas de Recorrer Colecciones**

Hay tres enfoques:

* for con índice: rápido en arrays, pero acoplado a la estructura.
* Iterator: desacoplado, permite eliminar elementos durante el recorrido.
* for-each: sintaxis moderna, simple, pero no soporta remove().

Lo esencial: usar for-each en la mayoría de los casos y Iterator cuando necesites modificar la colección durante el recorrido.