Contenido

[TEMA 1: Objetos y memoria 4](#_Toc119930741)

[1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE. 4](#_Toc119930742)

[1.2 Sentencias de control. 5](#_Toc119930743)

[ If...else 5](#_Toc119930744)

[ Switch 5](#_Toc119930745)

[ for & for-each 5](#_Toc119930746)

[ while & do...while 6](#_Toc119930747)

[ break & continue 6](#_Toc119930748)

[ Arrays 6](#_Toc119930749)

[ Entrada y salida de datos 7](#_Toc119930750)

[1.3 Abstracción y encapsulamiento 7](#_Toc119930751)

[1.4 Sobrecarga de métodos 8](#_Toc119930752)

[ Paquetes y module-info.java 8](#_Toc119930753)

[ Estructuras de datos 9](#_Toc119930754)

[ Java Enums 9](#_Toc119930755)

[ Conversión de tipos: parse/cast 9](#_Toc119930756)

[TEMA 2: Otros conceptos fundamentales de la POO 10](#_Toc119930757)

[2.1 Herencia. Interfaces y clases abstractas. Agregación. 10](#_Toc119930758)

[ Clases abstractas 10](#_Toc119930759)

[ Interfaces 10](#_Toc119930760)

[ Clases abstractas vs interfaces 11](#_Toc119930761)

[ Agregación y composición 11](#_Toc119930762)

[2.2 Polimorfismo 12](#_Toc119930763)

[2.3 Gestión de excepciones 12](#_Toc119930764)

[ Uso de los bloques try y catch [&finally] 13](#_Toc119930765)

[ Creación de excepciones propias 13](#_Toc119930766)

[2.4 Genericidad y Plantillas 13](#_Toc119930767)

[2.5 Utilidades entrada y salida 14](#_Toc119930768)

[ Cambiar entrada estándar 14](#_Toc119930769)

[ Archivos, lectura – java.io.Reader 14](#_Toc119930770)

[ Archivos, escritura – java.io.Writer 15](#_Toc119930771)

[2.6 Anotaciones 15](#_Toc119930772)

[ JavaDocs 15](#_Toc119930773)

[TEMA 3: Patrones de diseño 16](#_Toc119930774)

[3.0 Prácticas de programación 16](#_Toc119930775)

[ Malas prácticas de programación – Síntomas 16](#_Toc119930776)

[ Buenas prácticas de programación 16](#_Toc119930777)

[3.1 Concepto de patrones de diseño 18](#_Toc119930778)

[3.2 Patrones creacionales 18](#_Toc119930779)

[ Factory Method 19](#_Toc119930780)

[ Abstract Factory Method 20](#_Toc119930781)

[ Singleton 20](#_Toc119930782)

[3.3 Patrones estructurales 21](#_Toc119930783)

[ Adapter 22](#_Toc119930784)

[ Decorator 22](#_Toc119930785)

[3.4 Patrones de comportamiento 23](#_Toc119930786)

[ Chain of responsability 24](#_Toc119930787)

[ Observer 24](#_Toc119930788)

[ State 25](#_Toc119930789)

[ Template 25](#_Toc119930790)

[3.5 Refactorización 25](#_Toc119930791)

[TEMA 4: Programación interfaces 26](#_Toc119930793)

[4.1 Interfaces gráficas de Usuario 26](#_Toc119930794)

[ Clases Swing 26](#_Toc119930795)

[ Componentes básicos (Swing) 26](#_Toc119930796)

[ JComponents 27](#_Toc119930797)

[ Gestión de eventos 28](#_Toc119930798)

[TEMA 5: Temas avanzados 30](#_Toc119930799)

[5.1 Concurrencia 30](#_Toc119930800)

[ Threads (hilos) 30](#_Toc119930801)

[ Concurrency vs Parallelism 31](#_Toc119930802)

[31](#_Toc119930803)

[ Ejemplo de implementación 31](#_Toc119930804)

[ Thread Safety 32](#_Toc119930805)

[ Ejemplo de sincronización 32](#_Toc119930806)

[5.2 Inversión de control 33](#_Toc119930807)

[ Inyección de dependencias 33](#_Toc119930808)

[5.3 Expresiones avanzadas del lenguaje 33](#_Toc119930809)

[ Collections (java.util.Collection) 33](#_Toc119930810)

[ Clases anónimas 34](#_Toc119930811)

[ Clases anónimas - Ejemplo Listener/Eventos 34](#_Toc119930812)

[ Lambda expressions 34](#_Toc119930813)

[ Clases serializables – guardar y extraer objetos 35](#_Toc119930814)

# TEMA 1: Objetos y memoria

## Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.

* Java se basa en clases (class). Sólo hay clases (métodos, atributos) e interacciones entre ellas.
* Debe haber al menos una defición de clase en el programa.
* El programa principal (main) es una función/método público de una clase (public class). Sólo puede haber un main(String[] args) en el programa (podría haber otros si se cambian los argumentos de entrada, sobrecarga).

Imagen que contiene Diagrama

Descripción generada automáticamente

* **Público/Public (+):** cualquiera tiene acceso.
* **Privado/Private (-):** únicamente la clase puede acceder a la propiedad o método.
* **Protegido/Protected (#):** las clases del mismo paquete y que heredan de la clase pueden acceder a la propiedad o método.
* **Paquete / Package private (~)** (valor por defecto si no se indica ninguno): solo las clases en el mismo paquete pueden acceder a la propiedad o método.
* **Derivado/Derived property (/):** producido o calculado a partir del valor de otro atributo o método.
* **Estático/ static (subrayado):** Se puede acceder directamente a una variable estática por el nombre de clase y no necesita ningún objeto.

## Sentencias de control.

### If...else

Interfaz de usuario gráfica

Descripción generada automáticamente

### Switch

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

### for & for-each

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

### while & do...while

Interfaz de usuario gráfica

Descripción generada automáticamente

### break & continue

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

### Arrays

Texto

Descripción generada automáticamente

### Entrada y salida de datos

Texto

Descripción generada automáticamente

## Abstracción y encapsulamiento

Tabla

Descripción generada automáticamente

## Sobrecarga de métodos

- Mismo método (nombre) dentro de una clase, pero: diferentes argumentos, en diferente orden.

- Similar a sobrecarga de constructores.

- Polimorfismo: igual, pero desde clases hijas a clase padre (super).

Texto

Descripción generada automáticamente

### Paquetes y module-info.java

Paquete: agrupación de clases que comparten una temática o funcionalidad similar. Evitar conflictos de nombres entre clases (diferentes paquetes pueden tener clases que se llamen igual, pero para acceder a ellas hay que poner a qué paquete pertenecen).

Una clase puede acceder a todas las clases públicas que están en su mismo paquete, sin necesidad de indicar el nombre de dicho paquete. Si se desea acceder a otras que no están en su mismo paquete, se puede importar éste o indicar el nombre completo.

*Ejemplo:* import java.awt.image.BufferStrategy;

**¿Para qué sirve el module-info.java? (Se crea por defecto)**

Para exportar o importar paquetes, este fichero descriptor nos será de mucha utilidad para saber que exportamos o bien que importamos. Ejemplo:

Texto

Descripción generada automáticamente

### Estructuras de datos

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

### Java Enums

- Similar a una clase, pero no puede ser instanciado y no puede extender otras clases (aunque sí implementar interfaces).

- Para representar valores constantes. Los elementos son públicos, estáticos y finales.

- Nos permite asegurar que los parámetros pasados a un método están dentro de una lista de posibilidades.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

### Conversión de tipos: parse/cast

Texto

Descripción generada automáticamente

# TEMA 2: Otros conceptos fundamentales de la POO

## 2.1 Herencia. Interfaces y clases abstractas. Agregación.

Diagrama

Descripción generada automáticamente

### Clases abstractas

Diagrama

Descripción generada automáticamente

### Interfaces

Diagrama

Descripción generada automáticamente

### Clases abstractas vs interfaces

Interfaz de usuario gráfica

Descripción generada automáticamente

### Agregación y composición

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

## 2.2 Polimorfismo

Captura de pantalla de computadora

Descripción generada automáticamente

## 2.3 Gestión de excepciones

Tabla

Descripción generada automáticamente

Todos los errores y excepciones son subclases de Throwable, por lo que podrán acceder a sus

métodos. Los métodos más utilizados son los siguientes:

**- getMessage( )** Se usa para obtener un mensaje de error asociado con una excepción.

**- printStackTrace(PrintStream s )** Se utiliza para imprimir el registro del stack2 donde se ha iniciado la excepción. Para recoger la info también se puede usar **getStackTrace ()**.

**- toString( )** Se utiliza para mostrar el nombre de una excepción junto con el mensaje que devuelve getMessage().

### Uso de los bloques try y catch [&finally]

Texto

Descripción generada automáticamente

### Creación de excepciones propias

Texto

Descripción generada automáticamente

## 2.4 Genericidad y Plantillas

Texto

Descripción generada automáticamente

## 2.5 Utilidades entrada y salida

### Cambiar entrada estándar

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

### Archivos, lectura – java.io.Reader

Texto

Descripción generada automáticamente

### Archivos, escritura – java.io.Writer

Texto

Descripción generada automáticamente

## 2.6 Anotaciones

- No modifican la actividad de un programa ordenado → Pero si tienen efecto, pueden cambiar la forma en la que el compilador trata el programa. Ayudan a relacionar metadatos (info) con componentes del programa.

- Si no cumplimos lo mencionado en las @notaciones, el programa puede dar warning o errores.

*(Más info en las diapos del tema 2)*

### JavaDocs

- Genera documentación automática para nuestro programa si hacemos uso de los tags

- Podemos acompañar a las anotaciones de su correspondiente JavaDocs para dar más información

Texto

Descripción generada automáticamente

# TEMA 3: Patrones de diseño

## 3.0 Prácticas de programación

### Malas prácticas de programación – Síntomas

**Fragilidad:** al realizar un simple cambio, se rompe el código de otras muchas partes, incluso aunque no está relacionado en modo alguno. Incrementa el coste de cambiar el código, porque se pueden generar muchos problemas inesperados.

**Rigidez:** código difícil de cambiar, o extender. Requiere la creación de otros sistemas externos.

**Inmovilidad:** falta de flexibilidad, es difícil separar el Sistema en componentes para reusar el código en otras partes. Lleva a la duplicidad de Código.

**Viscosidad:** los problemas presentes en el código llevan a seguir empleando métodos malos, pues es más conveniente y fácil que aplicar buenas estructuras. Ciclo vicioso. Parches.

### Buenas prácticas de programación

**Abstracción:** La abstracción es un proceso de interpretación y diseño que implica reconocer y enfocarse en las características importantes de una situación u objeto, y filtrar o ignorar todas las particularidades no esenciales. Dejar a un lado los detalles de un objeto y definir las características específicas de éste, aquellas que lo distingan de los demás tipos de objetos. Hay que centrarse en lo que es y lo que hace un objeto, antes de decidir cómo debería ser implementado.

**Encapsulación:** Es la propiedad que permite asegurar que la información de un objeto está oculta del mundo exterior. Consiste en agrupar en una Clase las características (atributos) con un acceso privado y los comportamientos (métodos) con un acceso público → Acceder o modificar los miembros de una clase a través de sus métodos.

**Modularidad:** La modularidad es la propiedad que permite dividir una aplicación en partes más pequeñas, cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.

**Herencia/Generalización:** permite la reusabilidad de Código extendiendo clases y añadiendo nuevos métodos o atributos. Evita la repetición de código similar (redundancia).

* Diseño modular, encapsulación a nivel de método:
  + Código reutilizable (modular, con clases sencillas)
  + Extensible (mejor con clases abstractas e interfaces)
  + Núcleo del código protegido (**abstracción y encapsulamiento**) (**Abstraction & Encapsulation**)
* Programar a una interfaz no a una implementación
  + En lugar de hacer dependencias entre clases, introducir interfaz intermedia (**Generalization - Interface**)
* Favorecer la composición frente a la herencia (**Decomposition**)
  + Asociación (baja dependencia)
  + Agregación
  + Composición (alta dependencia) → Generalización
* Principios **SOLID**

Texto

Descripción generada automáticamente

Tabla

Descripción generada automáticamente

**Comentarios: no abusar**

- No deben sustituir a una funcionalidad. A veces se pueden sustituir por un método.

- Si el código es claro y el nombre de las variables y funciones está bien elegido, no son necesarios (GOAL).

- Si el Código cambia, hay que actualizar los comentarios relacionados. Un exceso de ellos aumenta el tiempo requerido para esta tarea.

**@annotations & JavaDocs NO SON COMENTARIOS! USAR SIEMPRE**

## 3.1 Concepto de patrones de diseño

Los patrones de diseño son soluciones habituales a problemas que ocurren con frecuencia en el diseño de software. No son plantillas, son consejos.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

*(En estos apuntes solo se tratarán los patrones en negrita, los más importantes)*

## 3.2 Patrones creacionales

Texto

Descripción generada automáticamente con confianza media

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

### Factory Method

Imagen que contiene Escala de tiempo

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

### Abstract Factory Method

Texto

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

### Singleton

Texto

Descripción generada automáticamente

## 3.3 Patrones estructurales

Texto

Descripción generada automáticamente con confianza baja

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

### Adapter

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

### Decorator



Diagrama

Descripción generada automáticamente

## 3.4 Patrones de comportamiento

Texto

Descripción generada automáticamente con confianza media

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación, Sitio web

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

### Chain of responsability

Texto

Descripción generada automáticamente

### Observer

Diagrama, Texto

Descripción generada automáticamente con confianza media

### State

Diagrama, Texto

Descripción generada automáticamente

### Template

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

## 3.5 Refactorización

## Diagrama, Gráfico de burbujas Descripción generada automáticamente

# TEMA 4: Programación interfaces

## 4.1 Interfaces gráficas de Usuario

**Java Swing** es una herramienta de interfaz gráfica de usuario (GUI) ligera que incluye un amplio conjunto de widgets. Incluye paquete que le permite crear componentes de GUI para sus aplicaciones Java, y es independiente de la plataforma.

### Clases Swing

Interfaz de usuario gráfica

Descripción generada automáticamente

### Componentes básicos (Swing)

Captura de pantalla de un celular

Descripción generada automáticamente

### JComponents

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

*(Código específico para JPanel, JFrame, etc. en el archivo de código)*

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

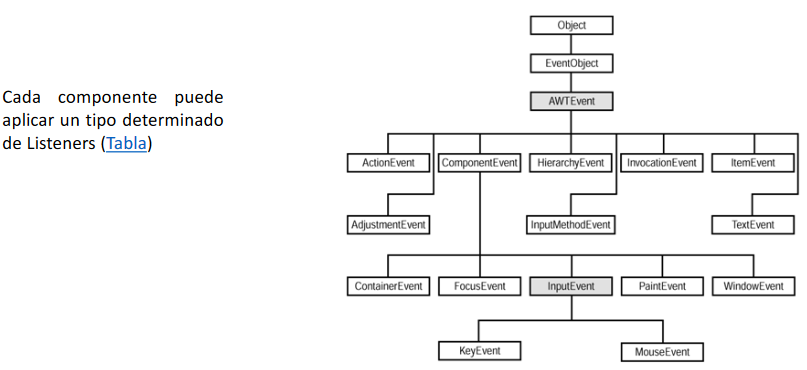
**Swing:** Crear clase que extienda JPanel/JComponent y sobrescribir paintComponent.

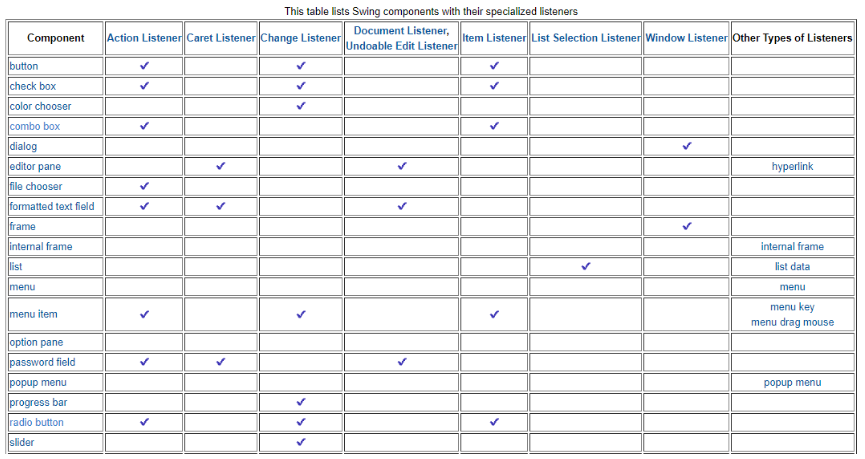
**AWT:** Crear clase que extienda Canvas y sobrescribir paint. También sobrescribir update para evitar flickering.

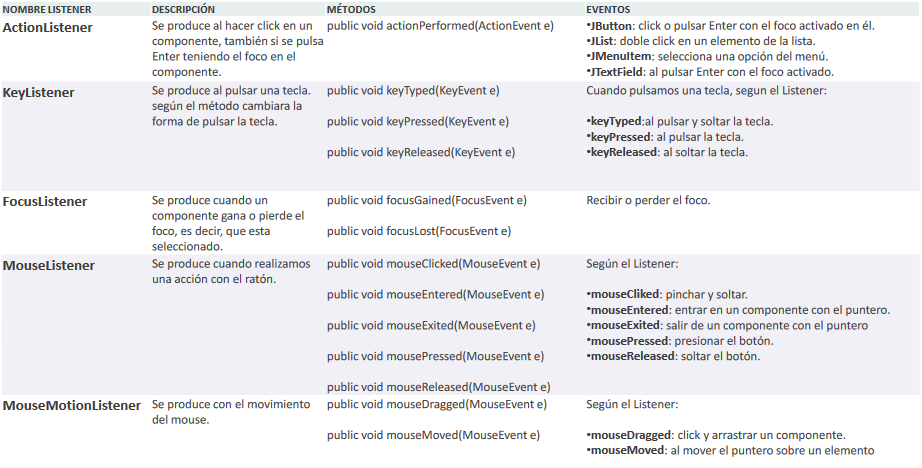
En cualquier caso, un contenedor Top Level (ej.: JFrame/Frame) es necesario.

*(Código específico para Frame y Graphics en el archivo de código)*

### Gestión de eventos







*(Código específico para Listeners en el archivo de código)*

# TEMA 5: Temas avanzados

## 5.1 Concurrencia

Ejecuciones en paralelo (multithreading) para ahorrar recursos y memoria, aislar actividades en diferentes hilos de ejecución, etc. En java, siempre hay al menos 1 Thread (main).

**Clase Thread – Métodos:**

* run() actividad del thread (active desde start)
* start() activa run() y vuelve al llamante
* join() espera por la terminación (timeout opcional)
* interrupt() sale de un wait, sleep o join
* isInterrupted()
* yield()

**Métodos estáticos:**

* sleep(milisegundos)
* currentTread()

**Métodos de la clase Object que controlan la suspensión de threads:**

* wait(), wait(milisegundos)
* notify(), notifyAll()

### Threads (hilos)

Unidades básicas de código que pueden ser ejecutadas al mismo tiempo.

Cómo implementar un hilo:

1. Extendiendo la clase Thread

2. Implementando la interfaz Runnable

En ambos casos, tendremos que instanciar después un Thread e invocar al método “start”.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

### Concurrency vs Parallelism

Gráfico, Gráfico de cajas y bigotes

Descripción generada automáticamente

### Gráfico Descripción generada automáticamente

### Ejemplo de implementación

Captura de pantalla con letras

Descripción generada automáticamente

### Thread Safety

-Si varios hilos tienen acceso a la misma variable o sección de código, deben establecerse métodos

de acceso sincronizados.

- Si dos o más hilos compiten por la misma información, y el orden puede afectar al resultado, se

genera una condición de carrera (race condition). Las partes del Código que generan estas

situaciones se denominan secciones críticas (critical section).

- Synchronized: puede usarse para métodos, bloques de Código (estáticos o no).

### Ejemplo de sincronización

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

## 5.2 Inversión de control

* Estilo de programación en el cual un framework o librería controla el flujo de un programa.
* Muy útil cuando se usan frameworks de desarrollo. Es el framework el que toma el control, el que define el flujo de actuación o el ciclo de vida de una petición. Es decir, es el framework quien ejecuta el código de usuario.
* La Inversión de control puede implementarse mediante:
  + Inyección de dependencias (se puede crear un contenedor que gestione las instancias u objetos)
  + Eventos (como en las aplicaciones con GUI)

### Inyección de dependencias

La inyección de dependencias es un patrón de diseño que permite construir software con poco acoplamiento.

El patrón funciona con un objeto que se encarga de construir las dependencias que una clase necesita y se las suministra (“inyecta”).

La clase no crea directamente los objetos que necesita, sino que los recibe de otra clase.

## 5.3 Expresiones avanzadas del lenguaje

### Collections (java.util.Collection)

Diagrama

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

### Clases anónimas

Las clases anónimas son expresiones, por lo que deben ser parte de una declaración. Se emplean como solución rápida para implementar una clase que se va a utilizar una vez y de forma inmediata. Se emplean mucho en listeners, eventos...

### Clases anónimas - Ejemplo Listener/Eventos

Texto

Descripción generada automáticamente

### Lambda expressions

Una expresión lambda es un bloque corto de código que toma parámetros y devuelve un valor. Las expresiones lambda son similares a los métodos, pero no necesitan un nombre y se pueden implementar directamente en el cuerpo de un método.

Son interfaces funcionales, solo tienen un método abstracto.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

### Clases serializables – guardar y extraer objetos

Diagrama

Descripción generada automáticamente

Para serializar un objeto, llamamos al método writeObject () de la clase ObjectOutputStream, y para la deserialización llamamos al método readObject () de la clase ObjectInputStream.

String y Wrappers: implementan Serializable. Los datos primitivos pueden ser escritos directamente por el ObjectOutputStream.

Algunas clases como JPanel lo implementan también.

Si no desea serializar ningún miembro de datos de una clase, puede marcarlo como transitorio (transient). Tras serializar y deserializar, se obtendrá un valor por defecto de dicha variable.

Si una clase implementa el marcador, todas las clases hijas lo tendrán también (herencia). Si una clase contiene a otra por agregación, está deberá ser también Serializable, o el proceso no podrá llevarse a cabo.

En general, todos los datos de la clase deben ser Serializables para que pueda ejecutarse sin excepciones.

*Código específico sobre [ObjectOutputStream.writeObject(Obj)] y [ObjectInputStream.readObject()] en el archivo de código.*