

Contenido

| | |
|--|----|
| TEMA 1: Objetos y memoria | 4 |
| 1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.4 | |
| 1.2 Sentencias de control..... | 5 |
| • If...else | 5 |
| • Switch | 5 |
| • for & for-each..... | 5 |
| • while & do...while | 6 |
| • break & continue..... | 6 |
| • Arrays | 6 |
| • Entrada y salida de datos | 7 |
| 1.3 Abstracción y encapsulamiento | 7 |
| 1.4 Sobrecarga de métodos | 8 |
| • Paquetes y module-info.java..... | 8 |
| • Estructuras de datos..... | 9 |
| • Java Enums | 9 |
| • Conversión de tipos: parse/cast..... | 9 |
| TEMA 2: Otros conceptos fundamentales de la POO | 10 |
| 2.1 Herencia. Interfaces y clases abstractas. Agregación. | 10 |
| • Clases abstractas | 10 |
| • Interfaces..... | 10 |
| • Clases abstractas vs interfaces | 11 |
| • Agregación y composición..... | 11 |
| 2.2 Polimorfismo | 12 |
| 2.3 Gestión de excepciones..... | 12 |
| • Uso de los bloques try y catch [&finally]..... | 13 |
| • Creación de excepciones propias | 13 |
| 2.4 Genericidad y Plantillas | 13 |
| 2.5 Utilidades entrada y salida | 14 |
| • Cambiar entrada estándar | 14 |
| • Archivos, lectura – java.io.Reader..... | 14 |
| • Archivos, escritura – java.io.Writer | 15 |
| 2.6 Anotaciones..... | 15 |
| • JavaDocs | 15 |

| | |
|---|----|
| TEMA 3: Patrones de diseño | 16 |
| 3.0 Prácticas de programación..... | 16 |
| • Malas prácticas de programación – Síntomas | 16 |
| • Buenas prácticas de programación | 16 |
| 3.1 Concepto de patrones de diseño | 18 |
| 3.2 Patrones creacionales | 18 |
| • Factory Method..... | 19 |
| • Abstract Factory Method | 20 |
| • Singleton..... | 20 |
| 3.3 Patrones estructurales | 21 |
| • Adapter..... | 22 |
| • Decorator | 22 |
| 3.4 Patrones de comportamiento | 23 |
| • Chain of responsability..... | 24 |
| • Observer | 24 |
| • State | 25 |
| • Template | 25 |
| 3.5 Refactorización..... | 25 |
| TEMA 4: Programación interfaces..... | 26 |
| 4.1 Interfaces gráficas de Usuario..... | 26 |
| • Clases Swing | 26 |
| • Componentes básicos (Swing)..... | 26 |
| • JComponents..... | 27 |
| • Gestión de eventos..... | 28 |
| TEMA 5: Temas avanzados..... | 30 |
| 5.1 Concurrencia | 30 |
| • Threads (hilos)..... | 30 |
| • Concurrency vs Parallelism | 31 |
| | 31 |
| • Ejemplo de implementación | 31 |
| • Thread Safety | 32 |
| • Ejemplo de sincronización..... | 32 |
| 5.2 Inversión de control | 33 |
| • Inyección de dependencias | 33 |

| | |
|---|----|
| 5.3 Expresiones avanzadas del lenguaje | 33 |
| • Collections (java.util.Collection)..... | 33 |
| • Clases anónimas | 34 |
| • Clases anónimas - Ejemplo Listener/Eventos..... | 34 |
| • Lambda expressions | 34 |
| • Clases serializables – guardar y extraer objetos | 35 |

TEMA 1: Objetos y memoria

1.1 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 definición 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).

| Genérico | Ejemplo |
|--|---|
| Clase | Dog |
| + attribute1:type = defaultValue + attribute2:type - attribute3:type | - name:String - age:int = 0 - attribute3:type |
| + operation1(params):returnType - operation2(params) - operation3() | + getAge():int + setName(String name) + getName(): String |

- **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.

1.2 Sentencias de control.

- **If...else**

```
if (condition) {  
    // code in if block  
}  
else {  
    // code in else block  
}
```

- ❖ Igual que en C++.
- ❖ Admite "if else" y anidación

```
if (condition1) {  
    // code  
} else if (condition2) {  
    // code  
} else if (condition3) {  
    // code  
} ... else {  
    // code  
}
```

```
if (x<100) {  
    System.out.println("n<100");  
    if (x<75) {  
        System.out.println("n<75");  
        if (x<50) {  
            System.out.println("n<50");  
        }  
    }  
}
```

```
result = (condition) ? return_if_True : return_if_False;
```

OPERADOR TERNARIO

"inline if statement"

```
int x=7, y=5;  
int mayor=(x>y)?x:y;
```

- **Switch**

```
switch (expression) {  
    case value1:  
        // code  
        break;  
    case value2:  
        // code  
        break;  
    ...  
    ...  
    default:  
        // default statements  
}
```

- ❖ Igual que en C++.
- ❖ Si no se añade el 'break', el resto de casos (tras elegir el valor verdadero) se ejecutarán.
- ❖ 'default': Cuando no se ha cumplido ninguna opción anterior

- **for & for-each**

- ❖ Igual que en C++. Se usa cuando el número de iteraciones es conocido.

```
for (initialExpression; testExpression; updateExpression) {  
    // body of the loop  
}
```

- ❖ FOR each...

```
for (dataType item : array) {  
    ...  
}
```

```
class Main {  
    public static void main(String[] args) {  
        int n = 5;  
  
        // for loop  
        for (int i = 1; i <= n; ++i) {  
            System.out.println("n is " + n);  
        }  
    }  
}
```

```
// create an array  
int[] numbers = {3, 7, 5, -5};  
  
// iterating through the array  
for (int i: numbers) {  
    System.out.println(i);  
}
```

Equivalente a (para recorrer arrays y colecciones):

```
for (int i = 0; i < numbers.length; ++ i)
```

- **while & do...while**

❖ Igual que en C++

```
while (testExpression) {
    // body of loop
}
```

```
do {
    // body of loop
} while (testExpression)
```

```
while(i <= n) {
    System.out.println(i);
    i++;
}
```

```
do{
    System.out.println(i);
    i++;
} while(i <= n);
```

do...while() al menos se ejecuta una vez

- **break & continue**

```
while(true) {
    System.out.println(i);
    i++;

    if (i == 5) {
        i++;
        continue;
    }
}
```

- ❖ Elementos terminación bucles:
- **break**: sale del bucle
 - **continue**: sale de la iteración en curso

- ❖ **labeled break/continue:**
- Útil para bucles anidados (Avanzado)

```
while(i<10) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
    i++;
}
```

¿i?

```
while(i<10) {
    if (i == 5) {
        i++;
        continue;
    }
    System.out.println(i);
    i++;
}
```

- **Arrays**



Reserva de espacio: **new**

- Devuelve una dirección de memoria, la referencia (puntero) a la información creada (objeto).
- Invoca al constructor de la clase

```
String [] nombres = new String [7];
nombres[0] = "Mary";

String [] nombres2 = {"Mary", "Angel", "Joy", "Monica"};

System.out.println(nombres2.length);

for (String item:nombres2) {
    System.out.println(item);
}

int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
int x = myNumbers[1][2];
System.out.println(x)

object[] mixedArray = new object[10];
```

- Entrada y salida de datos

```

import java.util.Scanner;

public class ClaseMain {

    public static void main(String[] args)
    {
        System.out.print("Enter a number: ");

        // create an object of Scanner
        Scanner input = new Scanner(System.in);

        // take input from the user
        int number = input.nextInt();

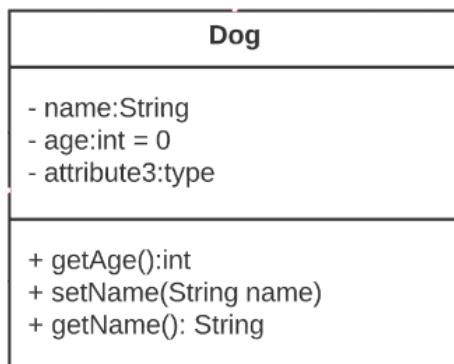
        System.out.println("You entered: " + number);

        // closes the scanner
        input.close();
    }
}

```

- Clase `Scanner` de `java.util` se usa para obtener entradas del teclado, archivos, usuarios, etc.
- `System.out [PrintStream]` indica salida standard (nuestra pantalla), por consola.
- `print` vs `println (new Line)`
 - Llamada implícita a `toString()`
- `System.in [InputStream]` indica que la entrada es standard (nuestro teclado).

1.3 Abstracción y encapsulamiento



1.4 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).

```
class Demo
{
    void multiply(int a, int b)
    {
        System.out.printIn("Result is"+(a*b));
    }

    void multiply(int a, int b, int c)
    {
        System.out.printIn("Result is"+(a*b*c));
    }

    public static void main(String[] args)
    {
        Demo obj = new Demo();
        obj.multiply(8,5);
        obj.multiply(4,6,2);
    }
}
```

- **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:

```
module rpg2D {
    requires module.name;
    exports package.name;
}
```

- **Estructuras de datos**

- **Array (built-in array)**: tamaño no se puede modificar. Hace falta crear otro array.
 - String [] cats = new String(5);
cats[0] = "Kitty";
- **ArrayList**: Se pueden añadir y eliminar elementos (set, get, remove, size, sort,)
 - import java.util.ArrayList;
import java.util.Collections; //Para usar sort
ArrayList<String> cats = new ArrayList<String>(); ← Objects, Wrappers
cats.add("Kitty");
- **LinkedList**: similar a ArrayList (ambos implementan la interfaz List), pero son elementos enlazados (nodos) en orden (addFirst, addLast, removeFirst, removeLast, getFirst, getLast, pop, peek, ..), **no se puede acceder de forma random**.
 - import java.util.LinkedList;
LinkedList<String> cats = new LinkedList<String>();
cats.add("Kitty");
- **HashMap**: como los diccionarios <key/value>
- **HashSet**: como los pools, saco con ítems únicos

- **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.

```
enum Race{ (←) Implementación simple o completa (↓)
    ELF,
    ORC,
    DWARF
}
enum Races{
    ELF("Elf"), DWARF("Dwarf"), ORC("Orc");
}
private final String Race;

Races (String Race) {
    this.Race = Race;
}

public String getRace() {
    return Race;
}

public static void setCharRace(Race race) {
    System.out.println(race);
}

public static void main(String[] args) {
    setCharRace(Race.DWARF);
}
```

- **Conversión de tipos: parse/cast**

String → int

public static int parseInt(String s)
public static int valueOf(s);

int i=Integer.parseInt("200");
int i=Integer.valueOf("200");

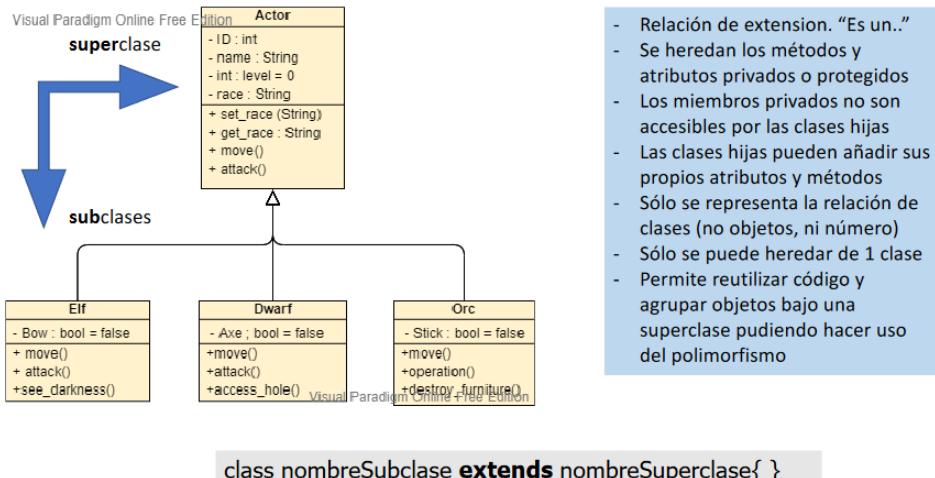
int → String

public static int parseInt(String s)
public static String valueOf(s);

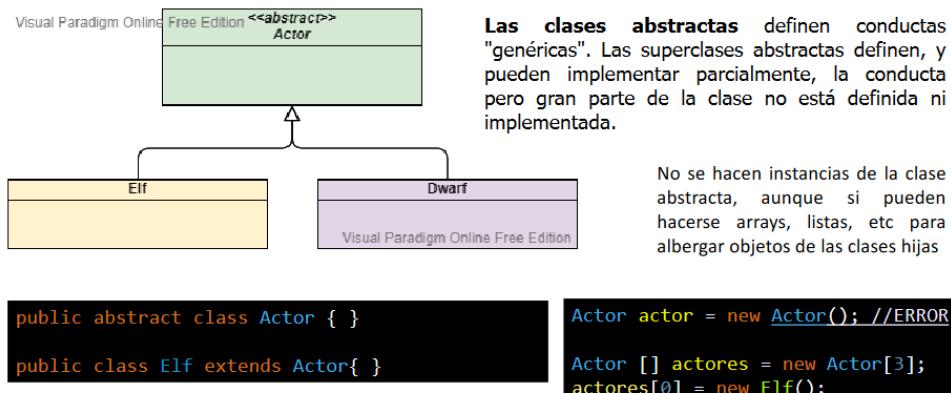
int i=Integer.parseInt("200");
String s=String.valueOf(Integer(200));

TEMA 2: Otros conceptos fundamentales de la POO

2.1 Herencia. Interfaces y clases abstractas. Agregación.

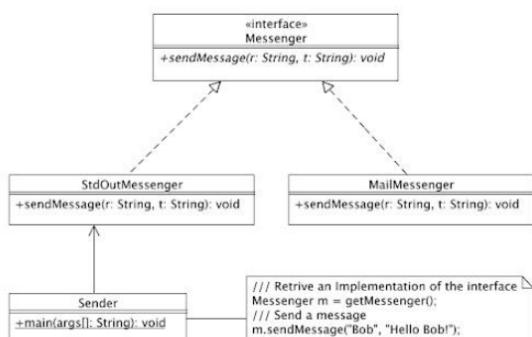


• Clases abstractas



• Interfaces

- Las interfaces son clases puramente abstractas
- No tienen atributos (new: puede tener constantes: static final)
- Poseen métodos sin implementar
- Para usarlas hay que implementarlas y definir todos sus métodos (si no se implementan todos, debe ser una clase abstracta)
- Se pueden implementar varias interfaces (separando nombres con comas)
- Las interfaces SI pueden heredar de varias interfaces. E implementar varias interfaces



```
class nombreSubclase implements nombreInterfaz{ }
```

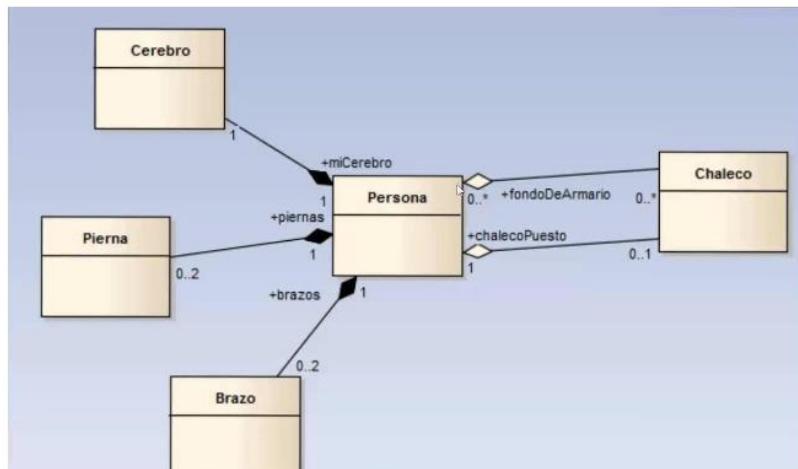
- Clases abstractas vs interfaces

| Interface | Abstract class |
|--|--|
| Interface support multiple inheritance | Abstract class does not support multiple inheritance |
| Interface does'n Contains Data Member | Abstract class contains Data Member |
| Interface does'n contains Constructors | Abstract class contains Constructors |
| An interface Contains only incomplete member (signature of member) | An abstract class Contains both incomplete (abstract) and complete member |
| An interface cannot have access modifiers by default everything is assumed as public | An abstract class can contain access modifiers for the subs, functions, properties |
| Member of interface can not be Static | Only Complete Member of abstract class can be Static |

Aunque no se instancie una clase abstracta, las clases hijas pueden llamar a su constructor en sus propios constructores.

Util para inicializar valores.

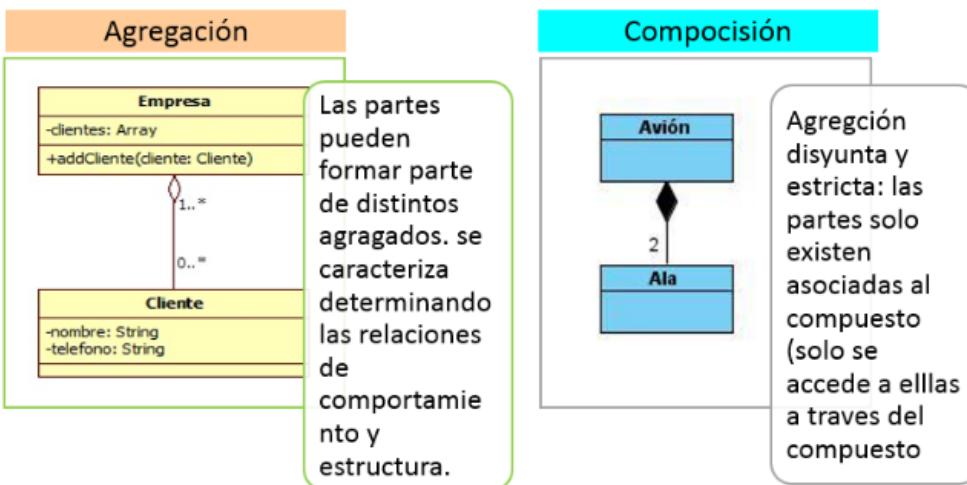
- Agregación y composición



| Persona |
|--------------------------|
| - miCerebro:Cerebro |
| - piernas[]: Pierna |
| - brazos[]: Brazo |
| + abrigar(chal:Chaleco); |

Agregación
El chaleco puede existir sin la persona, y después de que esta desaparezca.

Composición
Sin embargo, el cerebro, brazos y piernas...



2.2 Polimorfismo

```
class Animal {  
    public void makeSound() {  
        System.out.println("Grr...");  
    }  
}  
class Cat extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Meow");  
    }  
}  
class Dog extends Animal {  
    @Override  
    public void makeSound() {  
        System.out.println("Woof");  
    }  
}
```

Recordad: En la sobrecarga, ambos métodos siguen vigentes. Aquí el método de la clase hija anula a la clase padre, se sobreescribe.

```
public class ClaseMain {  
  
    public static void main(String[] args) {  
  
        Animal [] animales = new Animal[2];  
  
        animales[0] = new Dog();  
        animales[1] = new Cat();  
  
        for (Animal item:animales) {  
            item.makeSound();  
        }  
    }  
}
```

Polimorfismo, muchas formas. Allí donde se pueda usar el objeto padre, también se podrá hacer uso de cualquiera de sus hijas.

Principio de sustitución de Liskov (SOLID)

2.3 Gestión de excepciones

| Excepción | Error | |
|--|---|--|
| Situaciones que deberían solucionarse. El programa debería incluir bloques de try and catch para gestionarlos y recuperarse. | Problema grave debido a falta de recursos del sistema (run out memory, JVM error). | |
| RuntimeException | IOException | |
| Errores del programador (division por cero, acceso fuera de los límites de un array..). Gestión opcional | Errores que no puede evitar el programados, relacionados con E/S del programa. Gestión obligada. | El programa no debería "coger" estos errores. No es recuperable. Terminación del programa. |
| ArrayIndexOutOfBoundsException, NullPointerException | | OutOfMemoryError ,IOError |

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]

```
package ejemplos;

public class ClaseMain {
    public static void main(String[] args)
    {
        try{
            System.out.println(13/0);
        }
        catch(ArithmException myExcep) {
            System.err.println("Error aritmético");
        }
        finally{
            System.out.println("FIN");
        }
    }
}
```

Error aritmético
FIN

- Creación de excepciones propias

```
public class ValorNoValidoException extends Exception {
    public ValorNoValidoException() {
        super("El valor introducido no es válido");
    }
}

public class Checker{
    public double checkValue(double valor) throws ValorNoValidoException {
        if(valor < 0) throw new ValorNoValidoException ();
        return valor;
    }
}

Checker checker = new Checker();
try {
    checker.checkValue(-20);
} catch (ValorNoValidoException e) {
    e.printStackTrace();
}
```

2.4 Genericidad y Plantillas

```
static class Pair<T> {
    private T left, right;
    public Pair(T left, T right) {
        this.left = left;
        this.right = right;
    }
    public T getLeft() {return left;}
    public T getRight() {return right;}
}

public static void main (String[] args)  {
    Pair<Integer> i = new Pair(1,2);
    Pair<String> s = new Pair("hello", "world");
    Pair<Float> f = new Pair(1,2);

    System.out.println(i.getLeft());
    System.out.println(s.getRight());
    System.out.println(f.getRight());
}
```

2.5 Utilidades entrada y salida

- Cambiar entrada estándar

Cambio de entrada standard: `System.setIn`

Ejemplo: cambiamos la entrada por defecto a un archivo ➔

Importante: uso de gestión de excepciones con `try` y `catch`, también en el `finally`

De igual manera puede modificarse la salida estándar de error (`System.err()`).

```
FileInputStream fis = null;
Scanner scanner = new Scanner(System.in);

try {
    fis = new FileInputStream("entrada.txt");
    System.setIn(fis);

    while(scanner.hasNext()) {
        String s = scanner.next();
        System.out.println(s);
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    scanner.close();
    if (fis!=null) {
        try {
            fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Cambio de salida standard: `System.setOut`

Ejemplo: cambiamos la salida por defecto a un archivo:

```
import java.io.FileOutputStream;

FileOutputStream fos = new FileOutputStream("salida.txt");
System.setOut(new PrintStream(fos));

System.out.println("Hola Mundo!");

fos.close();
```

- Archivos, lectura – `java.io.Reader`

```
File archivo = null;
FileReader fr = null;
BufferedReader br = null;

try {
    archivo = new File ("C:\\archivo.txt");
    fr = new FileReader (archivo);
    br = new BufferedReader(fr);

    String linea;
    while((linea=br.readLine())!=null)
        System.out.println(linea);
}
catch(FileNotFoundException e){
    e.printStackTrace();
}finally{
    try{
        if( null != fr ){
            fr.close();
        }
    }catch (IOException e2){
        e2.printStackTrace();
    }
}
```



- Archivos, escritura – java.io.Writer

```

FileWriter fichero = null;
PrintWriter pw = null;
try {
    fichero = new FileWriter("c:/Prueba.txt");
    pw = new PrintWriter(fichero);

    for (int i = 0; i < 10; i++)
        pw.println("Linea " + i);

} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
        if (null != fichero)
            fichero.close();
    } catch (Exception e2) {
        e2.printStackTrace();
    }
}

```

Si queremos añadir al final de un fichero ya existente, simplemente debemos poner un *flag* a true como segundo parámetro del constructor de [FileWriter](#).

```
FileWriter fichero =
new FileWriter("c:/prueba.txt",true);
```

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

```

package rpg_packg;

/**
 * Main Class RPG game
 *
 * @author Nicuma3
 * @version %I%, %G%
 * @since 01/08/2021
 */
public abstract class Actor {

    //Member attributes
    /**
     * Character ID.
     */
    int ID;

    /**
     * Character Race. Elf, Dwarf, Orc, ...
     */
    private String race;
    //private Races ER;

    /**
     * Character Role. Wizard, Warrior, Explorer...
     */
    private String role;
}

//Member methods
/**
 * Actor no-parametric constructor
 */
public Actor() {

}

/**
 * Actor parametric constructor
 * @param name Actor's name
 * @param race Actor's race (Elf, dwarf, orc, ...)
 * @param role Actor's role (Wizard, explorer, warrior,
 * [...])
 */
public Actor(String name, String race, String role) {
    this.name = name;
    this.race = race;
    this.role = role;
}

```

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**

| Inicial | Acrónimo | Concepto |
|---------|----------|---|
| S | SRP | Principio de responsabilidad única (<i>Single responsibility principle</i>) la noción de que un <u>objeto</u> solo debería tener una única responsabilidad. |
| O | OCP | Principio de abierto/cerrado (<i>Open/closed principle</i>) la noción de que las “entidades de software ... deben estar abiertas para su extensión, pero cerradas para su modificación”. |
| L | LSP | Principio de sustitución de Liskov (<i>Liskov substitution principle</i>) la noción de que los “objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa”. Véase también diseño por contrato . |
| I | ISP | Principio de segregación de la interfaz (<i>Interface segregation principle</i>) la noción de que “muchas interfaces cliente específicas son mejores que una interfaz de propósito general”. ⁴ |
| D | DIP | Principio de inversión de la dependencia (<i>Dependency inversion principle</i>) la noción de que se debe “depender de abstracciones, no depender de implementaciones”. ⁴ La Inyección de Dependencias es uno de los métodos/patrones que siguen este principio |

| to DO – Keep it simple and easy | NOT to do |
|--|--|
| - Lanzar programa desde una clase creada específicamente para ello | - Tener todo mi programa en una misma clase |
| - Llamar a métodos desde dentro de los switch | - Poner dentro de los switch muchas líneas de código |
| - Separar acciones en métodos | - Tener métodos que realizan diferentes funciones |
| - Usar polimorfismo en vez de condicionales en las clases | - Heredar de una clase y repetir el mismo código en las clases hijas |
| - Usar comentarios cuando sea necesario para explicar por qué se implementa así el código o similar | - Abusar de comentarios y emplearlos como sustitutos de funciones/métodos |
| - Si usamos Código de otra persona, debemos dar atribuciones y referenciarlo. | - Usar variables con nombres genéricos, letras, |
| - Seguir las convenciones para nombres de clases, paquetes, módulos, métodos, variables, constantes... (mayúsculas/minúsculas, camelCase, snake_case | - Mezclar estilos. Check: https://en.wikipedia.org/wiki/Naming_convention_(programming) |

<

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.

| Familias | Se dedican a... | Ejemplos de patrones |
|-------------------|--|---|
| Creacionales | Proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización del código existente. | - Factory Method - Abstract Factory - Builder - Prototype - Singleton |
| Estructurales | Explican cómo ensamblar objetos y clases en estructuras más grandes, mientras se mantiene la flexibilidad y eficiencia de la estructura. | - Adapter - Bridge - Composite - Decorator - Facade - Flyweight - Proxy |
| De comportamiento | Tratan con algoritmos y la asignación de responsabilidades entre objetos. | - Chain of responsibility / - Command - Iterator / - Mediator - Memento / - Observer - State / - Strategy - Template method / - Visitor |

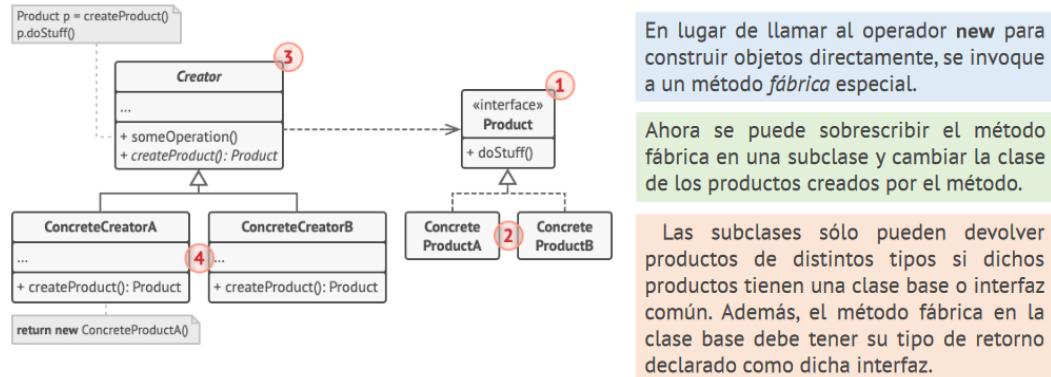
(En estos apuntes solo se tratarán los patrones en negrita, los más importantes)

3.2 Patrones creacionales

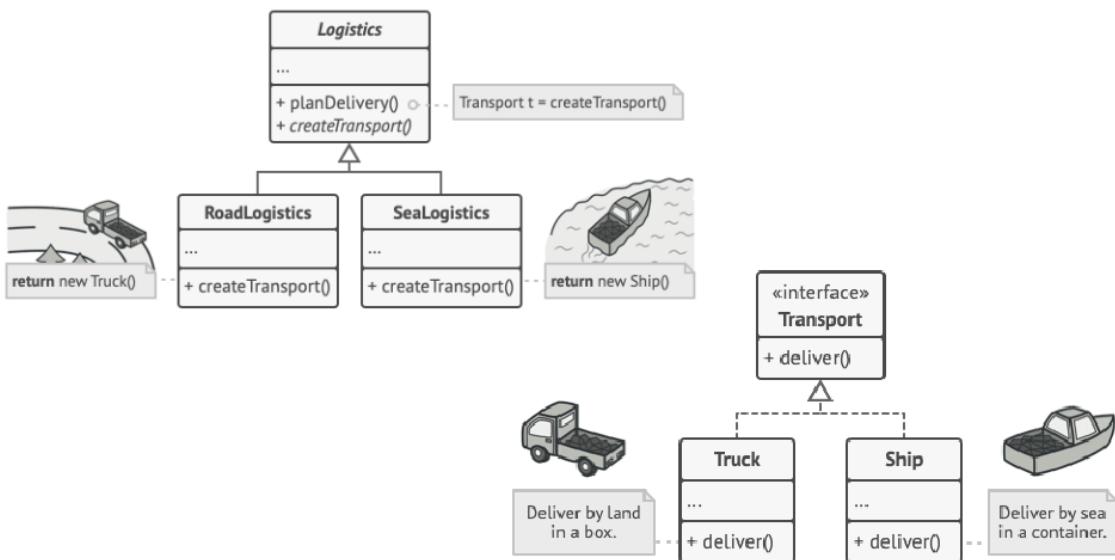
| Patrón | Finalidad | |
|-------------------------|--|--|
| Factory Method | Proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán. | |
| Abstract Factory | Nos permite producir familias de objetos relacionados sin especificar sus clases concretas. | |
| Builder | Nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción. | |
| Prototype | Nos permite copiar objetos existentes sin que el código dependa de sus clases | |
| Singleton | Nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia. | |



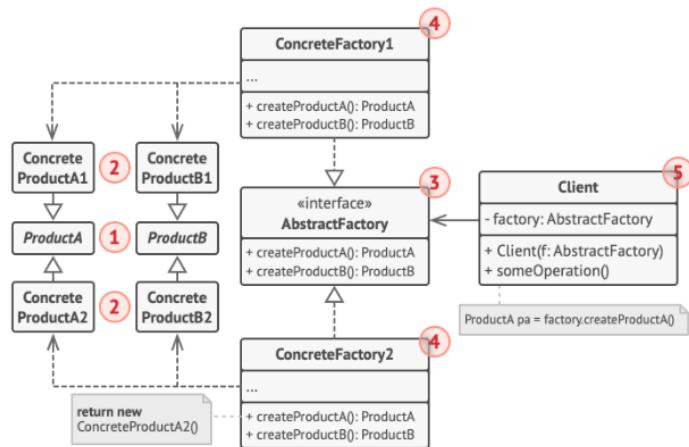
• Factory Method



Factory Method es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.



- **Abstract Factory Method**



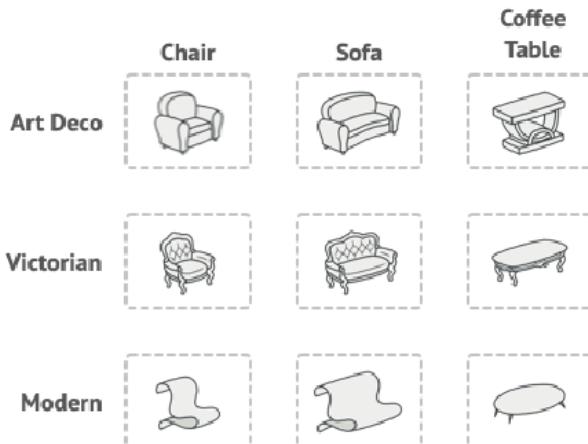
Declaramos de forma explícita interfaces para cada producto diferente de la familia de productos

Después podemos hacer que todas las variantes de los productos sigan esas interfaces.

El siguiente paso consiste en declarar la *Fábrica abstracta*: una interfaz con una lista de métodos de creación para todos los productos que son parte de la familia de productos. Estos métodos deben devolver productos **abstractos** representados por las interfaces que extrajimos previamente.

Para cada variante de una familia de productos, creamos una clase de fábrica independiente basada en la interfaz FábricaAbstracta

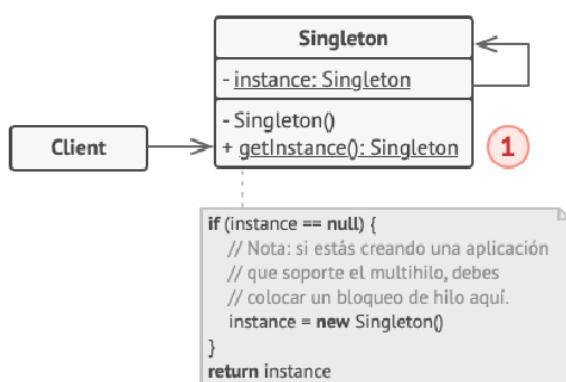
Abstract Factory es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.



Ejemplo:

- JComponents para diferentes sistemas operativos (diseño)

- **Singleton**



Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton.

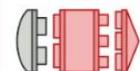
Crear un método de creación estático que actúe como constructor(este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático). Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Si el código tiene acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

3.3 Patrones estructurales

| Patrón | Finalidad |
|------------------|--|
| Adapter | Permite la colaboración entre objetos con interfaces incompatibles. |
| Bridge | Permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra. |
| Composite | Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales. |
| Decorator | Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades. |
| Facade | Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases. |
| Flyweight | Permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto. |
| Proxy | Permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original. |



Adapter

Permite la colaboración entre objetos con interfaces incompatibles.



Bridge

Permite dividir una clase grande o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.



Composite

Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.



Decorator

Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.



Facade

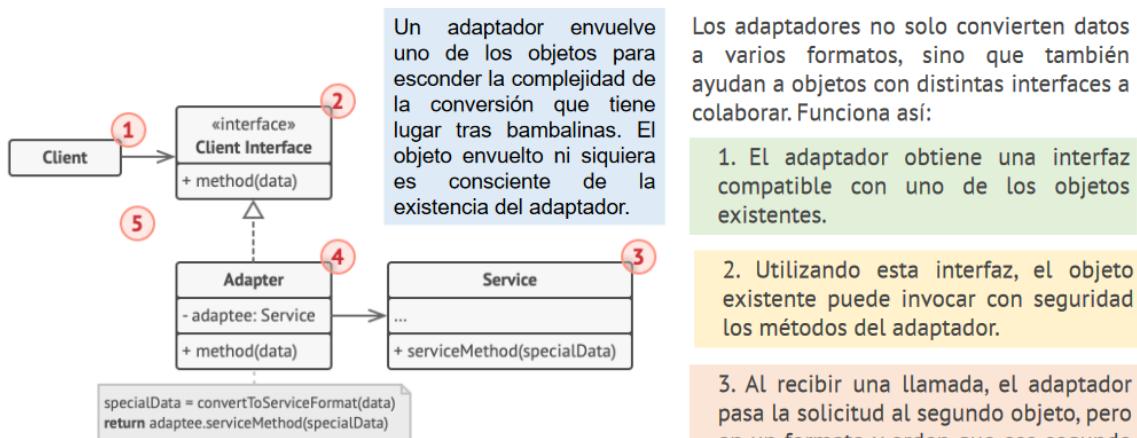
Proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.



Flyweight

Permite mantener más objetos dentro de la cantidad disponible de memoria RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.

• Adapter

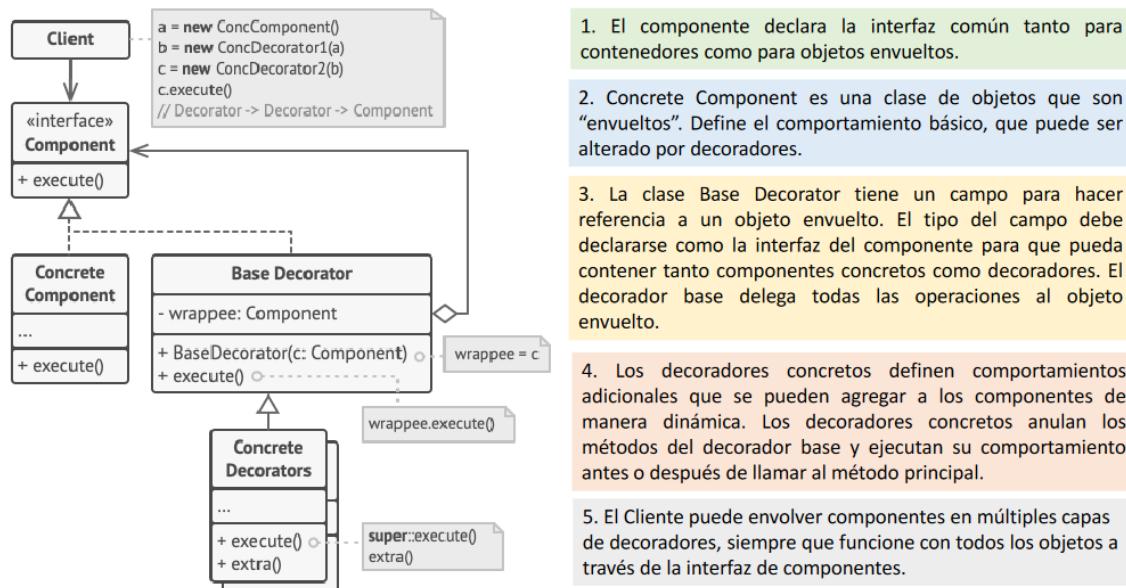


Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

En ocasiones se puede incluso crear un adaptador de dos direcciones que pueda convertir las llamadas en ambos sentidos.

• Decorator

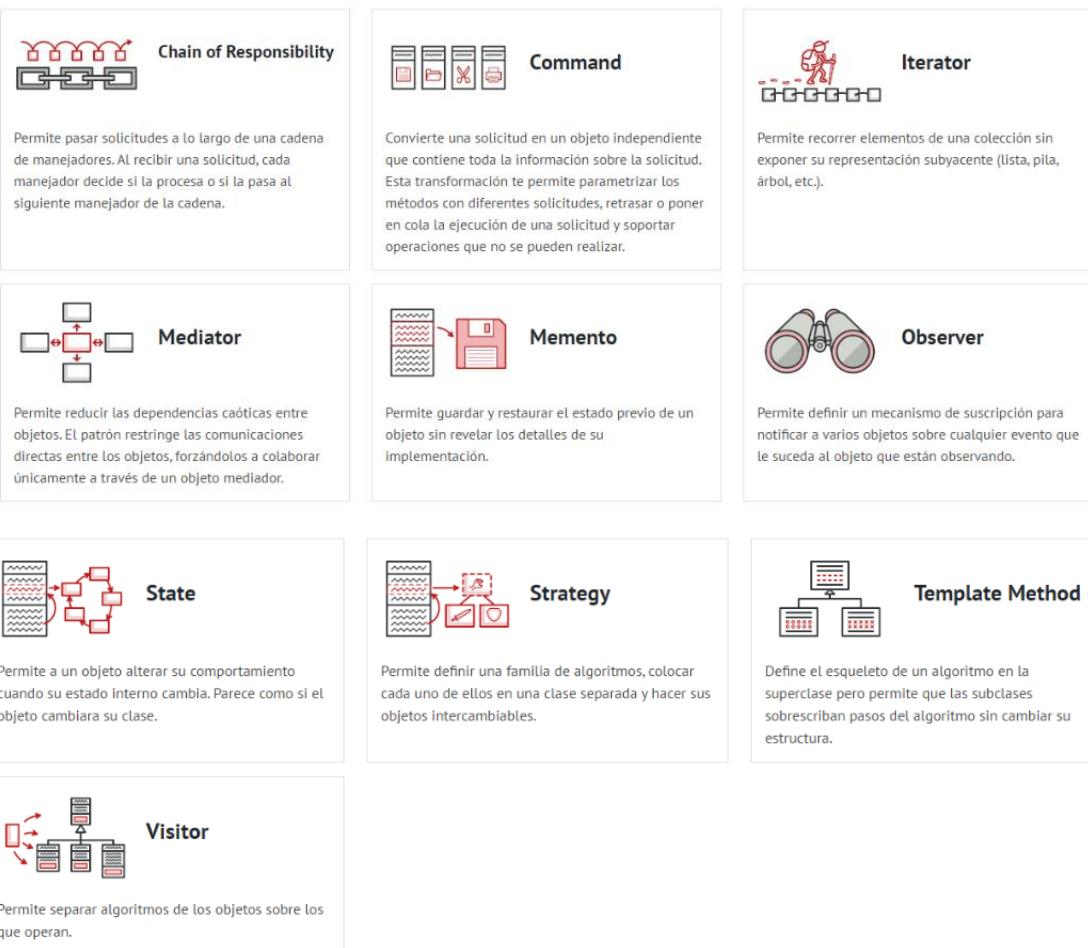
Decorator es un patrón de diseño estructural que te permite añadir funcionalidades sin necesidad de usar herencia (apila comportamientos)



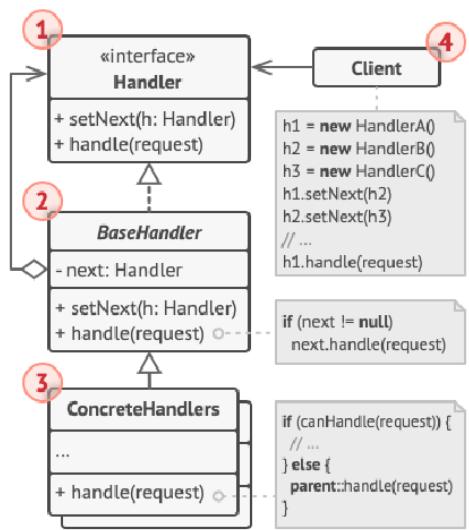
3.4 Patrones de comportamiento

| Patrón | Finalidad |
|--------------------------------|--|
| Chain of Responsibility | Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena. |
| Command | Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar. |
| Iterator | Permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.). |
| Mediator | Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador. |
| Memento | Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación. |
| Observer | Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando. |

| | |
|------------------------|---|
| State | Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase |
| Strategy | Permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables. |
| Template Method | Define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura. |
| Visitor | Permite separar algoritmos de los objetos sobre los que operan. |



- **Chain of responsibility**



1. El Handler declara la interfaz, común para todos los manipuladores concretos. Por lo general, contiene un solo método para manejar solicitudes, pero a veces también puede tener otro método para configurar el siguiente controlador de la cadena.

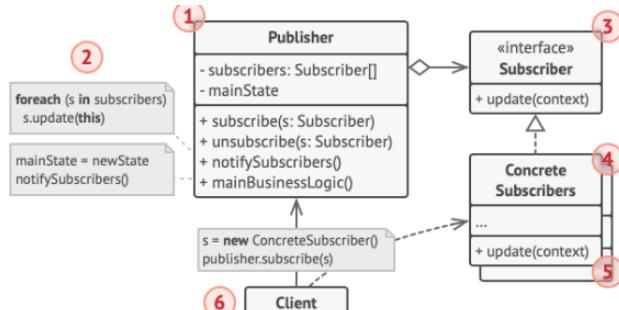
2. El controlador base es una clase opcional en la que puede colocar el código repetitivo que es común a todas las clases de controladores.

Por lo general, esta clase define un campo para almacenar una referencia al siguiente controlador. Los clientes pueden construir una cadena pasando un manipulador al constructor o definidor del manipulador anterior. La clase también puede implementar el comportamiento de manejo predeterminado: puede pasar la ejecución al siguiente controlador después de verificar su existencia.

3. Los manipuladores concretos contienen el código real para procesar las solicitudes. Al recibir una solicitud, cada manipulador debe decidir si la procesa y, además, si la pasa a lo largo de la cadena. Los controladores suelen ser autónomos e inmutables, y aceptan todos los datos necesarios solo una vez a través del constructor.

4. El Cliente puede componer cadenas solo una vez o componerlas dinámicamente, dependiendo de la lógica de la aplicación. Tenga en cuenta que se puede enviar una solicitud a cualquier controlador de la

- **Observer**



1. El editor emite eventos de interés para otros objetos. Estos eventos ocurren cuando el editor cambia su estado o ejecuta algunos comportamientos. Los editores contienen una infraestructura de suscripción que permite que se unan nuevos suscriptores y que los suscriptores actuales abandonen la lista.

2. Cuando ocurre un nuevo evento, el editor revisa la lista de suscripción y llama al método de notificación declarado en la interfaz del suscriptor en cada objeto de suscriptor.

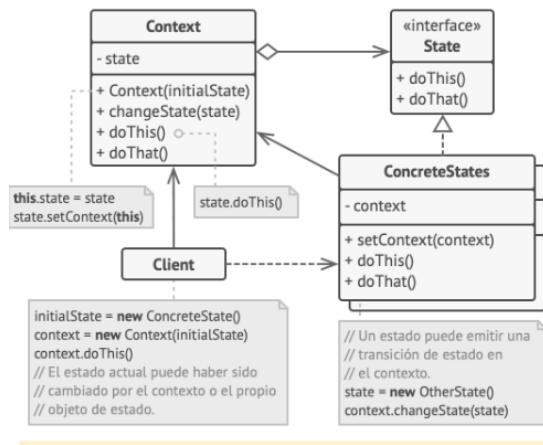
3. La interfaz del suscriptor declara la interfaz de notificación. En la mayoría de los casos, consta de un único método de actualización. El método puede tener varios parámetros que permiten al editor pasar algunos detalles del evento junto con la actualización.

4. Los suscriptores concretos realizan algunas acciones en respuesta a las notificaciones emitidas por el editor. Todas estas clases deben implementar la misma interfaz para que el editor no esté asociado a clases concretas.

5. Por lo general, los suscriptores necesitan información contextual para manejar la actualización correctamente. Por esta razón, los editores suelen pasar algunos datos de contexto como argumentos del método de notificación. El editor puede hacerse pasar por un argumento, lo que permite que el suscriptor obtenga los datos necesarios directamente.

6. El Cliente crea objetos de publicador y de suscriptor por separado y luego registra a los suscriptores para las actualizaciones.

- **State**



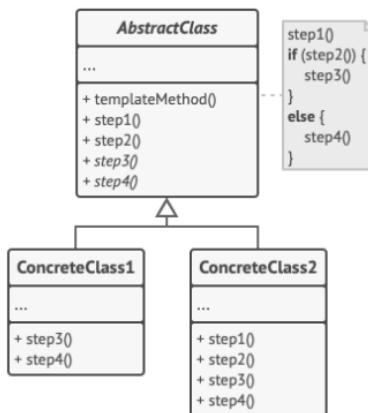
4. Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.

1. La clase **Contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (*setter*) para pasárselle un nuevo objeto de estado.

2. La interfaz **Estado** declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.

3. Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común. Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.

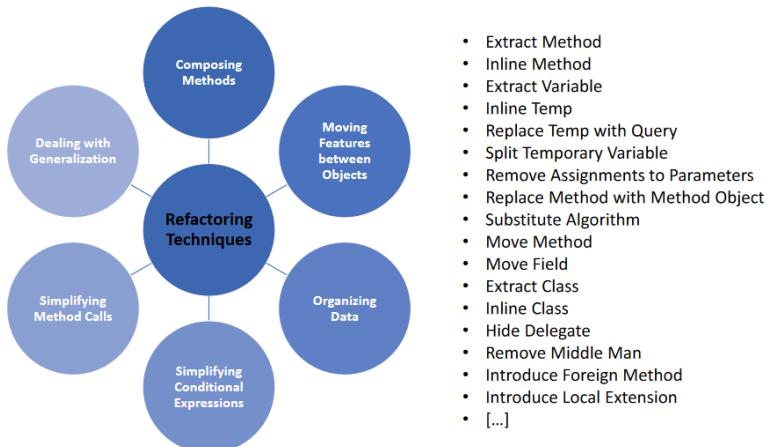
- **Template**



1. Clase Abstracta declara métodos que actúan como pasos de un algoritmo, así como el propio método plantilla que invoca estos métodos en un orden específico. Los pasos pueden declararse abstractos o contar con una implementación por defecto.

2. Las **Clases Concretas** pueden sobreescibir todos los pasos, pero no el propio método plantilla.

3.5 Refactorización

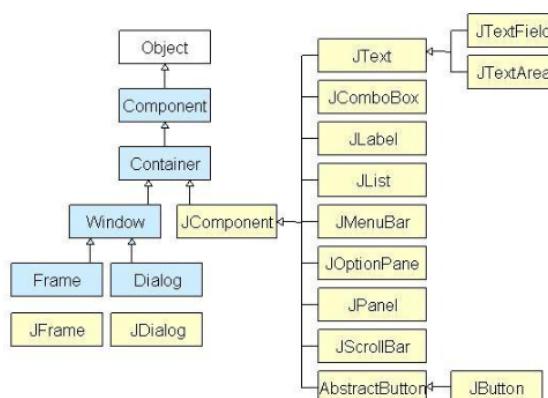


TEMA 4: Programación interfaces

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**



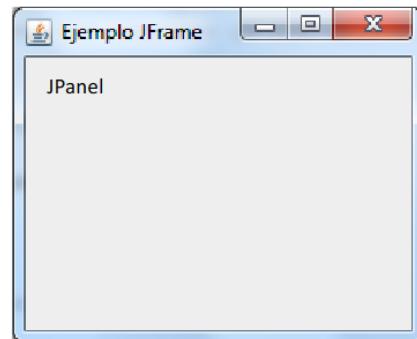
Top level containers (ventanas):

- JFrame (desktop)
- JApplet (browser)

Dentro:

JComponents

<https://docs.oracle.com/javase/tutorial/uiswing/components/rootpane.html>



- **Componentes básicos (Swing)**

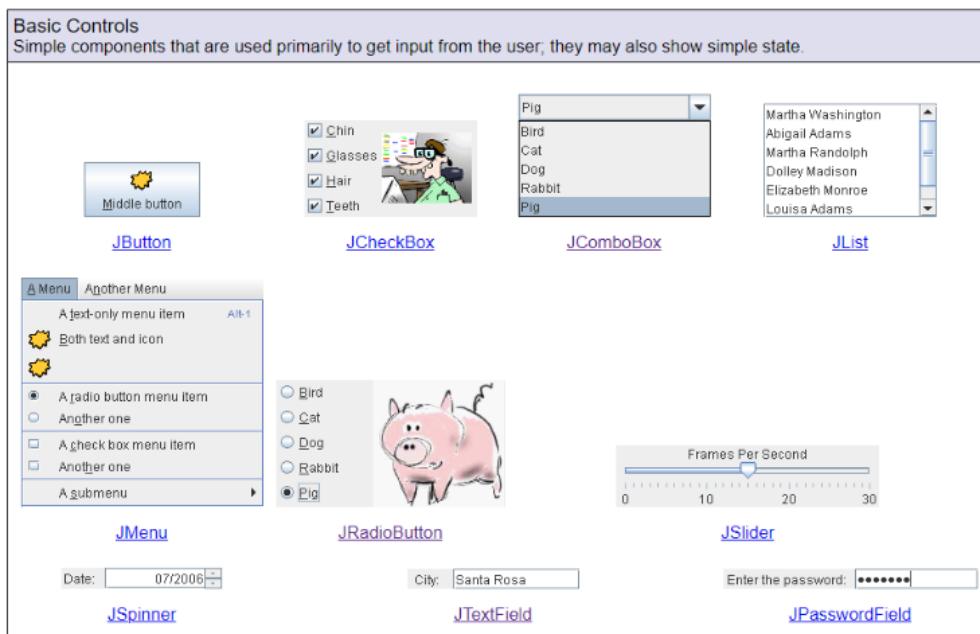
***Contenedores:**

- **JFrame** es la ventana principal del programa, **debe haber sólo uno**. Tiene icono, y sale en la barra de tareas.
- **JDialog** pertenece a un *JFrame*, debe tener un parent. No tiene icono ni sale en la barra de tareas. **Se pueden abrir muchos diálogos**. Si se configura como "modal" no deja tocar el resto de ventanas hasta cerrar/terminar esta.
- **JInternalFrame** corre dentro de otro panel (+ habitual: *JDesktopPane*) **Se pueden abrir muchas ventanas internas**.

***Componentes:**

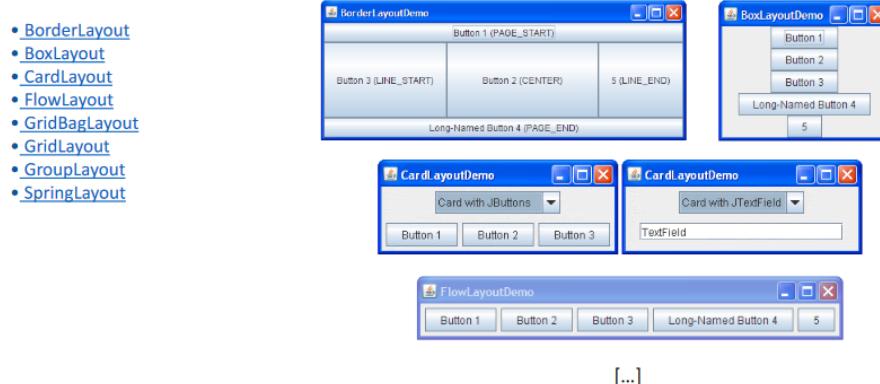
[AbstractButton](#), [BasicInternalFrameTitlePane](#), [Box](#), [Box.Filler](#), [JColorChooser](#), [JComboBox](#), [JFileChooser](#), [JInternalFrame](#), [JInternalFrame.JDesktopIcon](#), [JLabel](#), [JLayer](#), [JLayeredPane](#), [JList](#), [JMenuBar](#), [JOptionPane](#), [JPanel](#), [JPopupMenu](#), [JProgressBar](#), [JRootPane](#), [JScrollBar](#), [JScrollPane](#), [JSeparator](#), [JSlider](#), [JSpinner](#), [JSplitPane](#), [JTabbedPane](#), [JTable](#), [JTableHeader](#), [JTextComponent](#), [JToolBar](#), [JToolTip](#), [JTree](#), [JViewport](#)

- **JComponents**

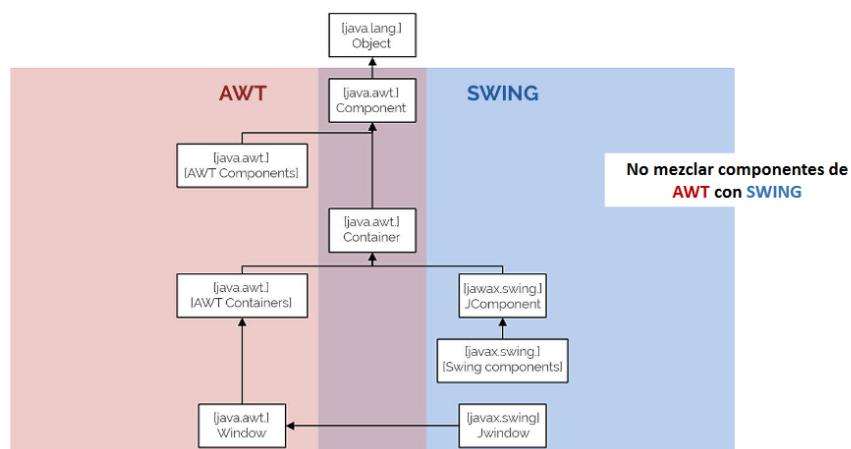


(Código específico para JPanel, JFrame, etc. en el archivo de código)

Si no desea colocar cada elemento manualmente mediante coordenadas, también se pueden emplear gestores de capas para realizar colocaciones automáticas. Existen diferentes gestores de Layouts de AWT & Swing:



[...]



| No. | Java AWT | Java Swing |
|-----|--|--|
| 1) | AWT components are platform-dependent . | Java swing components are platform-independent . |
| 2) | AWT components are heavyweight . | Swing components are lightweight . |
| 3) | AWT doesn't support pluggable look and feel . | Swing supports pluggable look and feel . |
| 4) | AWT provides less components than Swing. | Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc. |
| 5) | AWT doesn't follows MVC (Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view. | Swing follows MVC . |

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

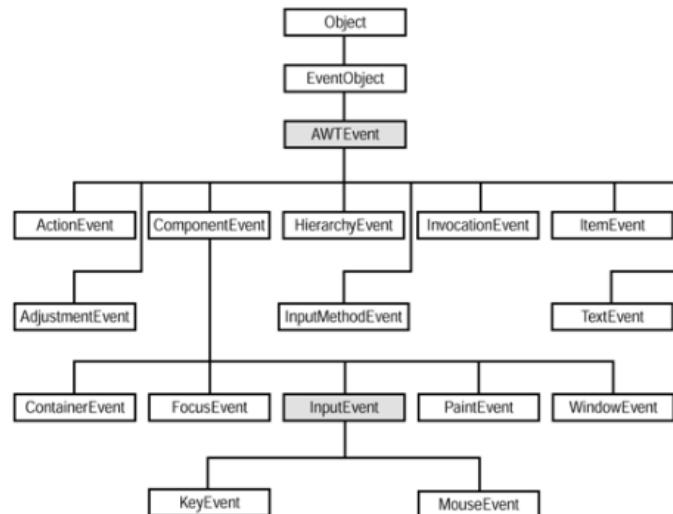
AWT: Crear clase que extienda Canvas y sobrescribir paint. También sobrescribir update para evitar flickering.

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**

Cada componente puede aplicar un tipo determinado de Listeners ([Tabla](#))



This table lists Swing components with their specialized listeners

| Component | Action Listener | Caret Listener | Change Listener | Document Listener, Undoable Edit Listener | Item Listener | List Selection Listener | Window Listener | Other Types of Listeners |
|----------------------|-----------------|----------------|-----------------|--|---------------|-------------------------|-----------------|-----------------------------|
| button | ✓ | | ✓ | | ✓ | | | |
| check box | ✓ | | ✓ | | ✓ | | | |
| color chooser | | | ✓ | | | | | |
| combo box | ✓ | | | | ✓ | | | |
| dialog | | | | | | | ✓ | |
| editor pane | | ✓ | | ✓ | | | | hyperlink |
| file chooser | ✓ | | | | | | | |
| formatted text field | ✓ | ✓ | | ✓ | | | | |
| frame | | | | | | | ✓ | |
| internal frame | | | | | | | | internal frame |
| list | | | | | | ✓ | | list data |
| menu | | | | | | | | menu |
| menu item | ✓ | | ✓ | | ✓ | | | menu key menu drag mouse |
| option pane | | | | | | | | |
| password field | ✓ | ✓ | | ✓ | | | | |
| popup menu | | | | | | | | popup menu |
| progress bar | | | ✓ | | | | | |
| radio button | ✓ | | ✓ | | ✓ | | | |
| slider | | | ✓ | | | | | |

| NOMBRE LISTENER | DESCRIPCIÓN | MÉTODOS | EVENTOS |
|---------------------|--|--|--|
| ActionListener | Se produce al hacer click en un componente, también si se pulsa Enter teniendo el foco en el componente. | public void actionPerformed(ActionEvent e) | • JButton: click o pulsar Enter con el foco activado en él. • JList: doble click en un elemento de la lista. • JMenuItem: selecciona una opción del menú. • JTextField: al pulsar Enter con el foco activado. |
| KeyListener | Se produce al pulsar una tecla, según el método cambiara la forma de pulsar la tecla. | public void keyTyped(KeyEvent e) public void keyPressed(KeyEvent e) public void keyReleased(KeyEvent e) | Cuando pulsamos una tecla, segun el Listener: •keyTyped:al pulsar y soltar la tecla. •keyPressed: al pulsar la tecla. •keyReleased: al soltar la tecla. |
| FocusListener | Se produce cuando un componente gana o pierde el foco, es decir, que esta seleccionado. | public void focusGained(FocusEvent e) public void focusLost(FocusEvent e) | Recibir o perder el foco. |
| MouseListener | Se produce cuando realizamos una acción con el ratón. | public void mouseClicked(MouseEvent e) public void mouseEntered(MouseEvent e) public void mouseExited(MouseEvent e) public void mousePressed(MouseEvent e) public void mouseReleased(MouseEvent e) | Según el Listener: •mouseClicked: pinchar y soltar. •mouseEntered: entrar en un componente con el puntero. •mouseExited: salir de un componente con el puntero. •mousePressed: presionar el botón. •mouseReleased: soltar el botón. |
| MouseMotionListener | Se produce con el movimiento del mouse. | public void mouseDragged(MouseEvent e) public void mouseMoved(MouseEvent e) | Según el Listener: •mouseDragged: click y arrastrar un componente. •mouseMoved: al mover el puntero sobre un elemento |

(Código específico para Listeners en el archivo de código)

TEMA 5: Temas avanzados

5.1 Concurrency

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)
- currentThread()

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

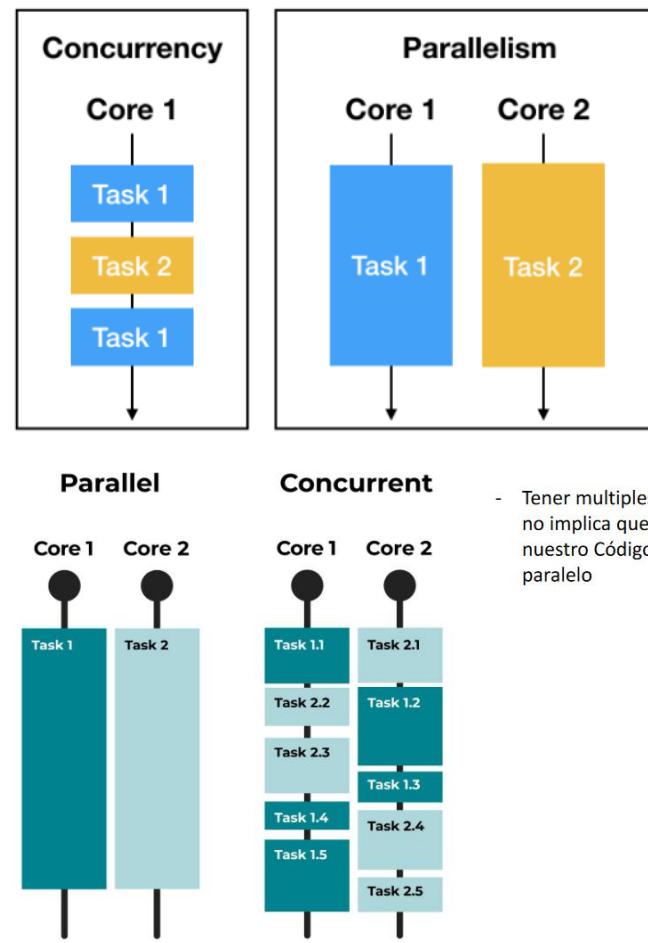
2. Implementando la interfaz Runnable

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

```
Public class myThread extends Thread {  
  
    @Override  
    public void run() {  
        //whatever  
    }  
  
    myThread t = new myThread(143);  
    t.start();
```

```
public class A implements Runnable{  
  
    @Override  
    public void run(){  
        //whatever  
    }  
  
    Thread t= new Thread(new A());  
    t.start();
```

- Concurrency vs Parallelism



- Ejemplo de implementación

```
import java.lang.Math ;

class EjemploThread extends Thread {
    int numero;
    EjemploThread (int n) { numero = n; }

    @Override
    public void run() {
        try {
            while (true) {
                System.out.println (numero);
                sleep((Long)(1000*Math.random()));
            }
        } catch (InterruptedException e) { return; } // acaba este thread
    }

    public static void main (String args[]) {
        for (int i=0; i<10; i++)
            new EjemploThread(i).start();
    }
}
```

Importante: se lanza con START para crear un nuevo hilo.

- **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**

```
class ThreadDemo extends Thread {  
    private Thread t;  
    private String threadName;  
    PrintDemo PD;  
  
    ThreadDemo( String name,  PrintDemo pd) {  
        threadName = name;  
        PD = pd;  
    }  
  
    public void run() {  
        synchronized(PD) { //Si elimino esto..  
            PD.printCount();  
        }  
        System.out.println("Thread " +  threadName + " exiting.");  
    }  
  
    public void start () {  
        System.out.println("Starting " +  threadName );  
        if (t == null) {  
            t = new Thread (this, threadName);  
            t.start ();  
        }  
    }  
}
```



```
class PrintDemo {  
    public void printCount() {  
        try {  
            for(int i = 5; i > 0; i--) {  
                System.out.println("Counter --- " + i );  
            }  
        } catch (Exception e) {  
            System.out.println("Thread interrupted.");  
        }  
    }  
}
```

```
//synchronized(PD) {
```

```
Starting Thread - 1  
Starting Thread - 2  
Counter --- 5  
Counter --- 4  
Counter --- 3  
Counter --- 5  
Counter --- 2  
Counter --- 1  
Counter --- 4  
Thread Thread - 1 exiting.  
Counter --- 3  
Counter --- 2  
Counter --- 1  
Thread Thread - 2 exiting.
```

```
synchronized(PD) {
```

```
Starting Thread - 1  
Starting Thread - 2  
Counter --- 5  
Counter --- 4  
Counter --- 3  
Counter --- 2  
Counter --- 1  
Thread Thread - 1 exiting.  
Counter --- 5  
Counter --- 4  
Counter --- 3  
Counter --- 2  
Counter --- 1  
Thread Thread - 2 exiting.
```

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**

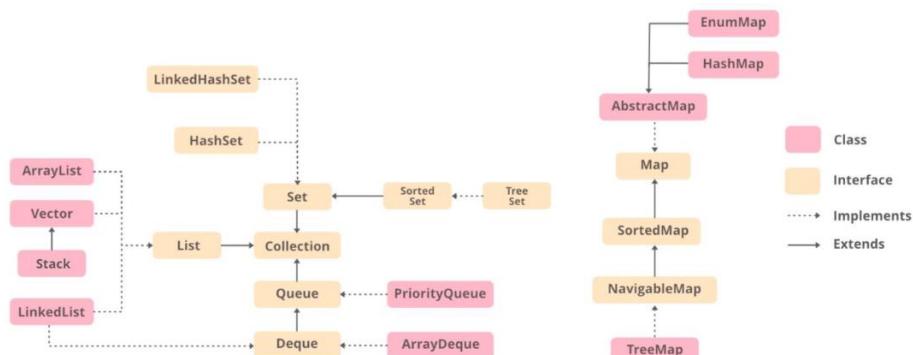
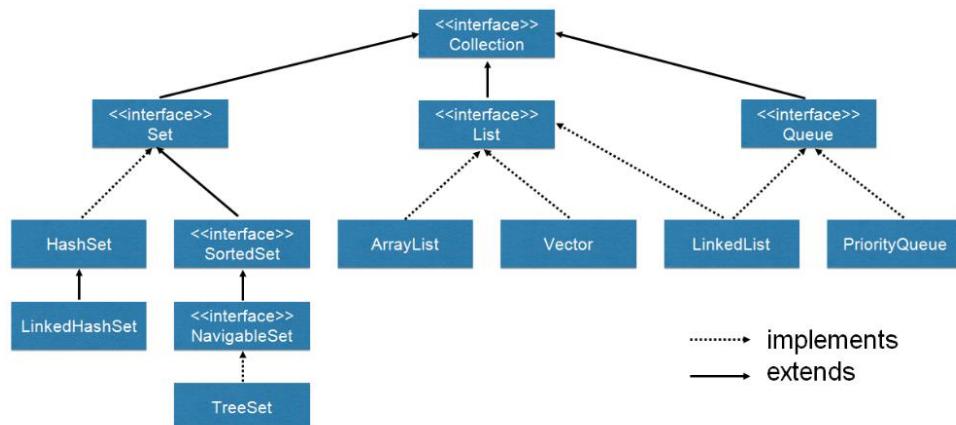
La inyección de dependencias es un patrón de diseño que permite construir software con poco acoplamiento.

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)**



- 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

```
public class Button1ActionListener implements ActionListener {  
    private JTextField jTextField;  
  
    public Button1ActionListener(JTextField jTextField) {  
        this.jTextField = jTextField;  
    }  
  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        jTextField.setText("button 1 clicked");  
    }  
}
```

Con clases anónimas:

```
// Previously defined: jButton1; jTextField;  
  
jButton1.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        jTextField.setText("button 1 clicked");  
    }  
});
```

Con exp. Lambda:

```
jButton1.addActionListener(e -> jTextField.setText("button 1 clicked"));
```

- 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.

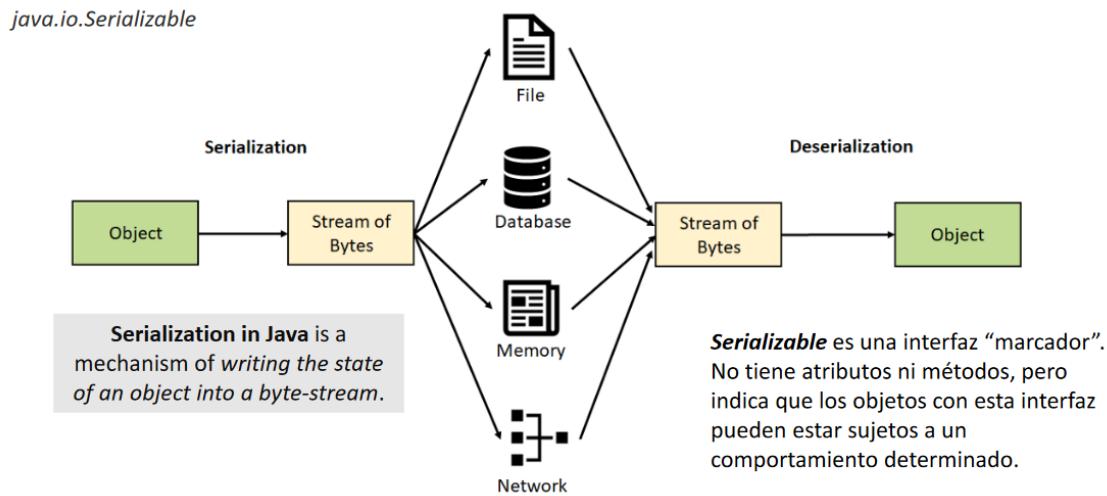
Son interfaces funcionales, solo tienen un método abstracto.

```
ArrayList<Integer> numbers = new  
ArrayList<Integer>();  
numbers.add(5);  
numbers.add(9);  
numbers.add(8);  
numbers.add(1);  
  
for (int n:numbers)  
System.out.println(n);
```

Sencillo para simplificar bucles con arraylist, usando el método forEach y pasando una función como argumento

```
numbers.forEach( (n) -> { System.out.println(n); } );
```

- Clases serializables – guardar y extraer objetos



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, ésta deberá ser también `Serializable`, o el proceso no podrá llevarse a cabo.

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

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