1.- Investiga buenas prácticas en Java:
a) Busca 5 ejemplos de buenas prácticas o de refactorización en programación Java, con una imagen del código de cada una de ellas y explica cada una.
1)

```
public class Ej3_4 {
    private static final Scanner teclado = new Scanner(source: System.in);
    public static void main(String[] args) {
        int tam, num;
        boolean flag = false;
        System.out.println(x: "Introduzca tamaño array");
        tam = teclado.nextInt();
        int[] array = new int[tam];
        System.out.println(x: "Introduzca array original");
        for (int i = 0; i < tam; i++) {
            array[i] = teclado.nextInt();
        }
        System.out.println(x: "Introduzca numero");
        num = teclado.nextInt();
        for (int i = 0; i < tam; i++) {
            if (array[i] == num) {
                flag = true;
            }
        }
        System.out.println(flag? "Se ha encontrado" : "No se ha encontrado");
    }
}
```

Para evitar la reutilización de variables temporales, se crean dos variables, una para buscar el número y otra para definir el tamaño del array. (Diapositiva 12)

2)

```
 * @author DAW1-B
 */
public class Ej2 {

    private static final Scanner teclado = new Scanner(source: System.in);

    public static void main(String[] args) {
        int alto, ancho;
        System.out.println(x: "Introduzca tamaño de la matriz");
        alto = teclado.nextInt();
        ancho = teclado.nextInt();
        for (int i = 0; i < alto; i++) {
            for (int j = 0; j < ancho; j++) {
                System.out.print(s: "-");
            }
            System.out.println(x: "");
        }
    }
```

Para el nombre de las variables, si tienen nombres explicativos, como en este caso el alto y ancho de la matriz, el código se entiende mejor. (Diapositiva 11)

3)

El código embebido evita variables temporales, siendo más eficiente. (Diapositivas 9-10)

```java
class Calculadora {

    private double num1;
    private double num2;

    public Calculadora(double num1, double num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    public double suma() {
        return num1 + num2;
    }

    public double resta() {
        return num1 - num2;
    }

    public double multiplicacion() {
        return num1 * num2;
    }

    public double division() {
        return num1 / num2;
    }
}
```

4)

Uso de campos encapsulados, getters y setters (Diapositiva 18)

```java
class Cuenta{
    private String titular;
    private double cantidad;

    public Cuenta(String titular, double cantidad){
        this.titular = titular;
        this.cantidad = cantidad;
    }
    public Cuenta(String titular){
        this(titular, cantidad: 0); //sobrecarga de constructores
    }
    public String getTitular(){ //getter
        return titular;
    }
    public void setTitular(String titular){ //setter
        this.titular = titular;
    }
    public double getCantidad(){ //getter
        return cantidad;
    }
    public void setCantidad(double cantidad){ //setter
        this.cantidad = cantidad;
    }
    @Override
    public String toString(){
        return "Cuenta: titular = " + titular + ", cantidad = " + cantidad;
    }
```

5)

```java
public class Ej4 {

    public static Integer CONSTANTE = 60;

    public static void main(String[] args) {
        int i, check = -1;
        float media = 0;
        int[] cont = new int[10], array = new int[CONSTANTE];
        for (i = 0; i < CONSTANTE; i++) {
            array[i] = (int) (Math.random() * 101);
            media += array[i] / CONSTANTE;
            cont[array[i] % 10]++;
        }
```

Uso de constantes para incluir el menor número posible de números en el código. (Diapositiva 20)

2.- Ventajas de refactorizar
a) Define con tus palabras y explica al menos 3 ventajas de refactorizar.
Refactorizar código da una mayor claridad al código, esto otorga un elevado número de ventajas, destacando las tres siguientes:
Facilidad a la hora de corregir errores y mejorar el código, pues al estar delimitado por unas normas comunes, los errores se encuentran con mayor facilidad.
Reducción de la duración del desarrollo de un programa, debido a que, al encontrar los errores con mayor facilidad, los errores se encuentran más rápidos, reduciendo además el número de errores que se producen durante el desarrollo.
La eficiencia de los códigos será mayor, al tener normas establecidas pensando en la eficiencia del programa.
(Diapositiva 5)
b) ¿Qué beneficios le aporta a una empresa desarrolladora de Software?
Reducir costos en reparaciones y actualizaciones en caso de errores posteriores a la entrega del Software, debido a que hay menos errores y son más fáciles de encontrar.
Facilidad para mejorar el código al entenderse mejor.
(https://www.nimblework.com/es/agile/refactorizacion-en-agil/)

3.- Proyecto Apuesta.

a) Cambia el nombre de la variable “miApuesta” por "laApuesta”.

```
 */
package Apuesta;

/**
 *
 * @author DAW1-B
 */
public class Main {

    public static void main(String[] args) {
        Apuesta laApuesta;
        int mi_dinero;

        laApuesta = new Apuesta(dinero_disp:1000, goles_local:4, goles_visitante:2);
        try {
            System.out.println(x: "Apostando...");
            laApuesta.apostar(dinero: 25);
        } catch (Exception e) {
            System.out.println(x: "Fallo al realizar la Apuesta");
        }

        try {
            System.out.println(x: "Intento cobrar apuesta segun el resultado del partido");
            laApuesta.cobrar_apuesta(cantidad_goles_local: 2, cantidad_goles_visit: 3);
        } catch (Exception e) {
            System.out.println(x: "Fallo al cobrar la apuesta");
        }
        mi_dinero = laApuesta.getDinero_disp();
        System.out.println("El dinero que tengo tras las apuestas es " + mi_dinero);
    }
}
```

b) Introduce el método “operativa_Apuesta”, que englobe las sentencias de la clase Main que operan con el objeto laApuesta.

```
public class Main {

    public static void main(String[] args) {
        Apuesta laApuesta;
        int mi_dinero;
        laApuesta = new Apuesta(dinero_disp:1000, goles_local:4, goles_visitante:2);
        mi_dinero = operativa_Apuesta(laApuesta);
        System.out.println("El dinero que tengo tras las apuestas es " + mi_dinero);
    }

    private static int operativa_Apuesta(Apuesta laApuesta) {
        int mi_dinero;
        try {
            System.out.println(x: "Apostando...");
            laApuesta.apostar(dinero: 25);
        } catch (Exception e) {
            System.out.println(x: "Fallo al realizar la Apuesta");
        }

        try {
            System.out.println(x: "Intento cobrar apuesta segun el resultado del partido");
            laApuesta.cobrar_apuesta(cantidad_goles_local: 2, cantidad_goles_visit: 3);
        } catch (Exception e) {
            System.out.println(x: "Fallo al cobrar la apuesta");
        }
        mi_dinero = laApuesta.getDinero_disp();
        return mi_dinero;
    }
}
```

c) Encapsula todos los atributos de la clase Apuesta.

```
 */
public class Apuesta {

    private int dinero_disp;
    private int goles_local;
    private int goles_visitante;
    private int apostado;

    public int getGoles_local() {
        return goles_local;
    }

    public void setGoles_local(int goles_local) {
        this.goles_local = goles_local;
    }

    public int getGoles_visitante() {
        return goles_visitante;
    }

    public void setGoles_visitante(int goles_visitante) {
        this.goles_visitante = goles_visitante;
    }

    public int getApostado() {
        return apostado;
    }

    public void setApostado(int apostado) {
        this.apostado = apostado;
    }
```

d) Añadir un nuevo parámetro al método operativa_Apuesta, de nombre dinero y de tipo int.

```
private static int operativa_Apuesta(Apuesta laApuesta) {
    int dinero;
    try {
        System.out.println(x: "Apostando...");
        laApuesta.apostar(dinero: 25);
    } catch (Exception e) {
        System.out.println(x: "Fallo al realizar la Apuesta");
    }

    try {
        System.out.println(x: "Intento cobrar apuesta segun el resultado del
        laApuesta.cobrar_apuesta(cantidad_goles_local: 2, cantidad_goles_visit: 3);
    } catch (Exception e) {
        System.out.println(x: "Fallo al cobrar la apuesta");
    }
    dinero = laApuesta.getDinero_disp();
    return dinero;
}
```

e) ¿Sé puede aplicar otra refactorización? Indica cuál
No he encontrado otros patrones de refactorización aplicables en este código.