Guia de Repaso

OOP Avanzado y Diseno de Clases

Semana 01 - Miercoles 11 de Febrero de 2026

Academia Java - Ciudad de Mexico

Temas del dia:

1. Herencia avanzada: clases abstractas e interfaces
2. Clases anonimas y expresiones lambda
3. Instruccion switch
4. Ejercicios del libro: capitulo 3
5. equals(), hashCode() y toString()
6. Encapsulamiento
7. Modificador final
8. Objetos inmutables y copias defensivas

Instructor: Miguel Rugerio

1. Herencia Avanzada: Clases Abstractas e Interfaces

Continuando con el proyecto herencia del dia anterior, se evoluciona la jerarquia de operaciones matematicas a traves de cuatro versiones que demuestran cuando usar clases abstractas vs interfaces.

Proyecto: herencia / v2 - Agregar Potencia

Se agrega una nueva operacion `Potencia` sin modificar el codigo existente. Esto demuestra el principio Open/Closed: abierto a extension, cerrado a modificacion.

Jerarquia de clases

```
Operacion (abstract)
    abstract int ejecuta(int x, int y)
    /     |        \         \
 Suma  Division  Multiplicacion  Potencia (NUEVA)
 x+y    x/y        x*y          x^y
```

Nueva subclase: Potencia.java

```
public class Potencia extends Operacion {
    @Override
    int ejecuta(int x, int y) {
        return (int)Math.pow(x, y);
    }
}
```

Principal.java - Se agrega sin cambiar show()

```
List<Operacion> operaciones = new ArrayList<>();
operaciones.add(new Suma());
operaciones.add(new Division());
operaciones.add(new Multiplicacion());
operaciones.add(new Potencia());       // Solo se agrega aqui
show(operaciones);                     // show() no cambia
```

> Open/Closed Principle: para agregar Potencia solo fue necesario crear una nueva clase y agregarla a la lista. El metodo show() no se modifico.

Proyecto: herencia / v3 - De Clase Abstracta a Interfaz

Se refactoriza `Operacion` de clase abstracta a interfaz. Las subclases ahora usan `implements` en lugar de `extends`. Se agrega una nueva operacion: `Resta`.

Interfaz Operacion.java

```
public interface Operacion {
    int ejecuta(int x, int y);       // public abstract por defecto
}
```

Implementaciones con implements

```
public class Suma implements Operacion {
    @Override
    public int ejecuta(int x, int y) {  // DEBE ser public
        return x + y;
    }
}

public class Resta implements Operacion {     // NUEVA
    @Override
    public int ejecuta(int x, int y) {
        return x - y;
    }
}
```

> ! Al implementar una interfaz, los metodos DEBEN ser public. En la clase abstracta (v2) tenian visibilidad default (package-private).

Principal.java - Identico patron

```
System.out.println("V3 Interface");
List<Operacion> operaciones = new ArrayList<>();
operaciones.add(new Suma());
operaciones.add(new Division());
operaciones.add(new Multiplicacion());
operaciones.add(new Resta());          // Nueva operacion
operaciones.add(new Potencia());
show(operaciones);
```

> Transparencia polimorfica: el codigo de Principal es practicamente identico en v2 y v3. El cambio de clase abstracta a interfaz no afecta al codigo que usa el tipo Operacion.

Proyecto: herencia / v4 - Clase Abstracta con Estado

Se retorna a usar clase abstracta porque ahora `Operacion` necesita mantener estado: atributos de instancia `x` e `y`, un constructor, y un contador estatico.

Clase abstracta con estado: Operacion.java

```
public abstract class Operacion {
    int x;                          // Atributo de instancia
    int y;                          // Atributo de instancia
    static int contador;            // Atributo estatico

    Operacion(int x, int y) {       // Constructor
        this.x = x;
        this.y = y;
        contador++;                 // Incrementa con cada instancia
    }

    abstract int ejecuta();         // Ya no recibe parametros

    static void showContador() {
        System.out.println("Contador: " + contador);
    }
}
```

Subclase Suma.java

```
public class Suma extends Operacion {
    Suma(int x, int y) {
        super(x, y);                // Llama al constructor padre
    }

    int ejecuta() {
        return x + y;               // Usa los atributos heredados
    }
}
```

> ! Se usa clase abstracta (no interfaz) porque se necesitan: atributos de instancia, constructor y contador estatico. Una interfaz NO puede tener ninguno de estos.

Proyecto: herencia / v5 - Interfaz con Funcionalidades Java 8+

Se demuestra que las interfaces modernas (Java 8+) pueden tener constantes estaticas, metodos default y metodos estaticos.

Interfaz moderna: Operacion.java

```
public interface Operacion {

    //public static final
    int contador = 0;                  // Constante (static final
implicito)

    //public abstract
    int ejecuta();                     // Metodo abstracto

    default public String show() {     // Metodo default (con
implementacion)
        return "Operacion";
    }

    static void showContador() {       // Metodo estatico
        System.out.println("Contador: " + contador);
    }
}
```

Suma.java - Debe manejar su propio estado

```
public class Suma implements Operacion {
    int x;                             // Cada clase maneja sus atributos
    int y;

    Suma(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int ejecuta() {
        return x + y;
    }
}
```

! La interfaz NO puede tener atributos de instancia ni constructor. El campo `contador` es `public static final` implicito (constante, no mutable). Si se necesita estado mutable compartido, se debe usar clase abstracta (v4).

Cuando usar cada uno

Necesitas...	Clase abstracta	Interfaz
Solo contrato (que hacer)	Si	Si (preferida)
Atributos de instancia	Si	No
Constructor	Si	No
Estado mutable compartido	Si (static)	No (solo final)
Herencia multiple	No (solo 1 extends)	Si (multiples implements)
Metodos con implementacion	Si	Si (default, Java 8+)

2. Clases Anonimas y Expresiones Lambda

Una clase anonima permite implementar una interfaz sin crear una clase con nombre. Las expresiones lambda (Java 8+) simplifican esta sintaxis cuando la interfaz tiene un solo metodo abstracto (interfaz funcional).

Proyecto: anonimas / v0 - Clases Anonimas

Interfaz funcional

```
public interface Operacion {
    int ejecuta(int x, int y);        // Un solo metodo abstracto
}
```

Implementacion con clases anonimas

```
List<Operacion> operaciones = new ArrayList<>();

Operacion ope1 = new Operacion() {    // Clase anonima
    @Override
    public int ejecuta(int x, int y) {
        System.out.println("Suma");
        return x + y;
    }
};

Operacion ope2 = new Operacion() {    // Otra clase anonima
    @Override
    public int ejecuta(int x, int y) {
        System.out.println("Potencia");
        return (int)Math.pow(x, y);
    }
};

operaciones.add(ope1);
operaciones.add(ope2);

operaciones.add(new Operacion() {     // Anonima inline
    @Override
    public int ejecuta(int x, int y) {
        System.out.println("Multi");
        return x * y;
    }
});
```

> Clase anonima: es una implementacion de la interfaz sin nombre. Se crea con `new Interfaz() { ... }`. Es util cuando solo se necesita una implementacion unica y local.

Proyecto: anonimas / v1 - Expresiones Lambda

Se refactoriza v0 reemplazando las clases anonimas con expresiones lambda. La sintaxis es mucho mas concisa.

Misma logica con lambdas

```
List<Operacion> operaciones = new ArrayList<>();

Operacion ope1 = (x, y) -> x + y;                    // Lambda: suma

Operacion ope2 = (w, q) -> (int)Math.pow(w, q);     // Lambda: potencia

operaciones.add(ope1);
operaciones.add(ope2);

operaciones.add((pato1, pato2) -> pato1 * pato2);  // Lambda inline
```

Comparacion de sintaxis

Clase anonima	Lambda equivalente

`new Operacion() {`
`  public int ejecuta(int x, int y) {`
`  return x + y;`
`  }`
`}`
	`(x, y) -> x + y`

> ! Las lambdas solo funcionan con interfaces funcionales (un solo metodo abstracto). Los nombres de los parametros son libres: (x,y), (w,q) o (pato1,pato2) son todos validos.

3. Instruccion Switch

La instruccion `switch` permite evaluar una variable contra multiples casos. Es importante entender el comportamiento de fall-through (cuando no hay break) y el uso de variables `final` en los cases.

Proyecto: switchExample / v0

```
String dia = "JUEVES";
final String domingo = "DOMINGO";    // final: se puede usar en case

switch(dia) {

default : System.out.println("Dia 6");    // default NO tiene que ir al
final

case domingo: System.out.println("Dia 0");
        break;
case "LUNES": System.out.println("Dia 1");
        break;
case "MARTES": System.out.println("Dia 2");
        break;
case "MIERCOLES": System.out.println("Dia 3");  // Sin break!
case "VIERNES": System.out.println("Dia 4");    // Sin break!

}
```

Salida con dia = "JUEVES"

```
Dia 6
Dia 0
```

Ningun case coincide con "JUEVES", entonces entra al `default`. Como el default NO tiene break, continua ejecutando el siguiente case (fall-through) hasta encontrar un break en el case `domingo`.

Reglas del switch

Regla	Detalle
default	Puede ir en cualquier posicion (no solo al final)
break	Sin break, la ejecucion cae al siguiente case (fall-through)
Variables en case	Solo se permiten constantes: literales o variables final
Tipos permitidos	int, byte, short, char, String, enum

! Fall-through es un error comun. Si dia = "MIERCOLES", imprime "Dia 3" Y "Dia 4" porque no hay break entre esos cases.

4. Ejercicios del Libro: Capitulo 3

Ejercicio de logica de control con bucles anidados. Se analiza paso a paso el flujo de ejecucion para predecir la salida.

Proyecto: chapter03Book / Question 26

```
int w = 0, r = 1;
String name = "";

while (w < 2) {
    name += "A";
    do {
        name += "B";
        if (name.length() > 0)
            name += "C";
        else
            break;
    } while (r <= 1);
    r++;
    w++;
}

System.out.println(name);
```

Traza de ejecucion

Paso	w	r	name	Que sucede
Inicio	0	1	""	Variables iniciales
while	0	1	"A"	w<2 es true, agrega "A"
do 1a	0	1	"ABC"	Agrega "B", length>0, agrega "C"
while(r<=1)	0	1	"ABC"	r<=1 es true, repite do
do 1b	0	1	"ABCBC"	Agrega "B", agrega "C" (bucle infinito)

! Este es un bucle infinito. Dentro del do-while, la variable `r` nunca cambia (r++ esta fuera del do-while), asi que la condicion `r <= 1` siempre es true. El programa nunca terminaria.

> Leccion: siempre verificar que la condicion de salida de un bucle sea alcanzable. Si la variable de control no se modifica dentro del bucle, es un bucle infinito.

5. equals(), hashCode() y toString()

Tres metodos de la clase Object que toda clase puede sobreescribir para definir como se comparan, identifican y representan los objetos.

Proyecto: hashcode / v0 - equals()

Se implementa `equals()` para comparar objetos por contenido en lugar de por referencia.

```
public class Pato extends Object {
    String name;
    int age;

    @Override
    public boolean equals(Object obj) {
        Pato other = (Pato) obj;
        return age == other.age
            && Objects.equals(name, other.name);
    }
}
```

Comparacion en Principal

```
Pato pato1 = new Pato("Donald", 5);
Pato pato2 = new Pato("Donald", 5);

System.out.println(pato1 == pato2);      // false (diferentes objetos)
System.out.println(pato1.equals(pato2)); // true  (mismo contenido)

String name1 = new String("Donald");
String name2 = new String("Donald");

System.out.println(name1 == name2);      // false (diferentes objetos)
System.out.println(name1.equals(name2)); // true  (mismo contenido)
```

Operador/Metodo	Que compara	Resultado
`==`	Referencias (direcciones de memoria)	false si son objetos distintos
`.equals()`	Contenido (segun implementacion)	true si los campos son iguales

Proyecto: hashcode / v1 - toString()

Se agrega `toString()` para una representacion legible del objeto.

```
@Override
public String toString() {
    return "Pato [name=" + name + ", age=" + age + "]";
}
```

```
Pato pato1 = new Pato("Donald", 5);
System.out.println(pato1.toString());  // Pato [name=Donald, age=5]
System.out.println(pato1);             // Pato [name=Donald, age=5]
```

> `System.out.println(objeto)` llama automaticamente a `toString()`. Sin sobreescribirlo, imprime algo como `com.curso.v0.Pato@1a2b3c`.

Proyecto: hashcode / v2 - hashCode() con HashSet

Se implementa `hashCode()` junto con `equals()` para el correcto funcionamiento en colecciones basadas en hash.

```
@Override
public int hashCode() {
    return Objects.hash(age, name);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) return true;                    // Misma referencia
    if (obj == null) return false;                   // Null check
    if (getClass() != obj.getClass()) return false;  // Mismo tipo
    Pato other = (Pato) obj;
    return age == other.age && Objects.equals(name, other.name);
}
```

Uso en HashSet

```
Pato pato1 = new Pato("Donald", 5);
Pato pato2 = new Pato("Donald", 5);
Pato pato3 = new Pato("Lucas", 8);
Pato pato4 = new Pato("Lucas", 8);

Set<Pato> patos = new HashSet<>();
patos.add(pato1);   // Se agrega
patos.add(pato2);   // NO se agrega (duplicado)
patos.add(pato3);   // Se agrega
patos.add(pato4);   // NO se agrega (duplicado)

// Solo imprime 2 patos: Donald y Lucas
```

! Contrato hashCode/equals: si dos objetos son iguales segun equals(), DEBEN tener el mismo hashCode(). Si solo implementas equals() sin hashCode(), el HashSet no detecta duplicados correctamente.

6. Encapsulamiento

El encapsulamiento protege los datos internos de un objeto controlando el acceso a traves de metodos. Permite validar datos y cambiar la implementacion interna sin afectar a los clientes.

Proyecto: encapsular / v0 - Sin encapsulamiento (problema)

```
public class Pato {
    String name;        // Acceso default (package-private)
    int age;            // Acceso default

    public Pato(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Cliente2 - Asignacion invalida

```
Pato pato = new Pato("Donald", 5);
pato.name = "Feo";
pato.age = -10;         // Edad negativa! No hay validacion
System.out.println(pato);  // Pato [name=Feo, age=-10]
```

> ! Sin encapsulamiento, cualquier cliente puede asignar valores invalidos directamente. No hay forma de proteger la integridad de los datos.

Proyecto: encapsular / v1 - Con encapsulamiento

```
public class Pato {
    private String name;      // private: solo accesible dentro de Pato
    private int age;          // private

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if (age < 0)
            throw new UnsupportedOperationException("Edad invalida");
        this.age = age;        // Solo se asigna si es valido
    }
}
```

Cliente2 - Ahora falla con edad invalida

```
Pato pato = new Pato("Donald", 5);
pato.setName("Feo");
pato.setAge(-10);      // UnsupportedOperationException: Edad invalida
```

Aspecto	v0 (sin encapsular)	v1 (encapsulado)
Atributos	default (accesibles)	private (protegidos)
Acceso	pato.age = -10	pato.setAge(-10) → excepcion
Validacion	Ninguna	En setters
Integridad	Comprometida	Garantizada

7. Modificador final

El modificador `final` en una variable impide que se le reasigne un nuevo valor. Sin embargo, si la variable apunta a un objeto mutable, el contenido del objeto SI puede modificarse.

Proyecto: finalDemo / v0

```
final StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");          // OK: modifica el contenido del objeto
//sb = sb.append(" World");   // ERROR: no se puede reasignar la
referencia
//sb = null;                  // ERROR: no se puede reasignar
//sb = new StringBuilder("Hola"); // ERROR: no se puede reasignar
System.out.println(sb);       // Hello World
```

Que protege final

Accion	Permitida?	Razon
`sb.append(" World")`	Si	Modifica el contenido, no la referencia
`sb = null`	No	Intenta reasignar la referencia
`sb = new StringBuilder()`	No	Intenta reasignar la referencia

> `final` protege la REFERENCIA (la flecha), no el OBJETO al que apunta. El objeto sigue siendo mutable si su clase lo permite (como StringBuilder).

8. Objetos Inmutables y Copias Defensivas

Un objeto inmutable no puede ser modificado despues de su creacion. Esto es fundamental para seguridad en concurrencia (multithreading) y para evitar efectos secundarios no deseados.

Proyecto: inmutable / v0 - Clase mutable

La clase `Estudiante` tiene setters que permiten modificar todos sus campos despues de la creacion.

```
public class Estudiante {
    private String nombre;
    private int edad;
    private List<String> materias;

    // Constructor, getters Y setters
    public void setNombre(String nombre) { this.nombre = nombre; }
    public void setEdad(int edad) { this.edad = edad; }
    public void setMaterias(List<String> materias) { this.materias =
materias; }
}
```

> ! Con setters, cualquier cliente puede modificar el estado del objeto en cualquier momento. No hay garantia de inmutabilidad.

Proyecto: inmutable / v2 - Intento sin copias defensivas

Se eliminan setters y se marcan los campos como `final`, pero NO se copian los objetos mutables en el constructor.

```
final public class Estudiante {
    final private StringBuilder matricula;   // MUTABLE
    final private List<String> materias;     // MUTABLE

    public Estudiante(..., StringBuilder matricula, List<String>
materias) {
        this.matricula = matricula;          // Guarda la MISMA
referencia
        this.materias = materias;            // Guarda la MISMA
referencia
    }

    public StringBuilder getMatricula() {
        return matricula;                    // Retorna la MISMA
referencia
    }
}
```

El problema

```
StringBuilder matricula = new StringBuilder("XYZ777");
Estudiante est1 = new Estudiante("Patrobas", 20, matricula,
listaMaterias);

System.out.println(est1);    // matricula=XYZ777

matricula.append(" 999");    // Modifica el StringBuilder ORIGINAL

System.out.println(est1);    // matricula=XYZ777 999  (CAMBIO!)
```

> ! El objeto "inmutable" fue modificado desde afuera porque el constructor guardo la misma referencia al StringBuilder. Modificar `matricula` externamente afecta al Estudiante.

Proyecto: inmutable / v3 - Inmutabilidad verdadera

Se aplican copias defensivas tanto en el constructor como en los getters para aislar completamente el objeto.

Reglas de inmutabilidad

1. Clase marcada como `final` (no se puede heredar)
2. Todos los atributos son `final` y `private`
3. No hay setters
4. Copias defensivas en el **constructor** (objetos mutables entrantes)
5. Copias defensivas en los **getters** (objetos mutables salientes)

```
final public class Estudiante {
    final private String nombre;
    final private int edad;
    final private StringBuilder matricula;
    final private List<String> materias;

    public Estudiante(String nombre, int edad,
            StringBuilder matricula, List<String> materias) {
        this.nombre = nombre;                          // String es
inmutable, OK
        this.edad = edad;                              // Primitivo, OK
        this.matricula = new StringBuilder(matricula); // COPIA defensiva
        this.materias = new ArrayList<>(materias);     // COPIA defensiva
    }

    public String getNombre() { return nombre; }     // Inmutable, OK
    public int getEdad() { return edad; }            // Primitivo, OK

    public StringBuilder getMatricula() {
        return new StringBuilder(matricula);          // COPIA defensiva
    }

    public List<String> getMaterias() {
        return new ArrayList<>(materias);             // COPIA defensiva
    }
}
```

Ahora es seguro

```
StringBuilder matricula = new StringBuilder("XYZ777");
Estudiante est1 = new Estudiante("Patrobas", 20, matricula,
listaMaterias);

System.out.println(est1);    // matricula=XYZ777

matricula.append(" 999");    // Modifica la variable LOCAL

System.out.println(est1);    // matricula=XYZ777  (NO CAMBIO!)
```

> Copias defensivas: el constructor crea copias NUEVAS de los objetos mutables, y los getters retornan copias NUEVAS. Asi, nadie externo puede afectar el estado interno del objeto.

Resumen: cuando se necesita copia defensiva

Tipo	Inmutable?	Necesita copia?
String	Si	No
int (primitivo)	Si (por valor)	No
StringBuilder	No	Si (constructor y getter)
List, Set, Map	No	Si (constructor y getter)

9. Resumen del Dia

Proyecto	Tema principal	Concepto clave
herencia v2	Agregar subclase	Open/Closed Principle
herencia v3	Clase abstracta → interfaz	implements, visibilidad public
herencia v4	Abstracta con estado	Atributos, constructor, static
herencia v5	Interfaz Java 8+	default, static, constantes
anonimas v0	Clases anonimas	new Interfaz() { ... }
anonimas v1	Expresiones lambda	(x,y) → x + y
switchExample	Instruccion switch	fall-through, final en case
chapter03Book	Logica de control	Bucles anidados, traza
hashcode v0	equals()	== vs .equals()
hashcode v1	toString()	Representacion legible
hashcode v2	hashCode()	Contrato con HashSet
encapsular v0-v1	Encapsulamiento	private + getters/setters
finalDemo	Modificador final	Referencia vs contenido
inmutable v0-v3	Inmutabilidad	Copias defensivas

Estructura de los proyectos

```
herencia/src/com/curso/
  v2/  -- Clase abstracta + Potencia
  v3/  -- Interfaz + Resta
  v4/  -- Abstracta con estado
  v5/  -- Interfaz Java 8+

anonimas/src/com/curso/
  v0/  -- Clases anonimas
  v1/  -- Expresiones lambda

switchExample/src/switchExample/
  Principal.java  -- Switch con fall-through

chapter03Book/src/com/curso/v0/
  Question26.java -- Bucles anidados

hashcode/src/com/curso/
  v0/  -- equals()
  v1/  -- toString()
  v2/  -- hashCode() + HashSet

encapsular/src/com/curso/
  v0/  -- Sin encapsulamiento
  v1/  -- Con encapsulamiento

finalDemo/src/com/curso/v0/
  Principal.java  -- Modificador final

inmutable/src/com/curso/
  v0/  -- Clase mutable
  v1/  -- Copia en constructor y getter (parcial)
  v2/  -- Sin copias (problema)
  v3/  -- Copias defensivas completas
```

Para repasar

1. Compara herencia v2 (extends) con v3 (implements): observa que cambios son necesarios al migrar a interfaz.

2. Intenta agregar una nueva operacion (ej. Modulo) a v3 sin modificar Principal.java.

3. Convierte las clases anonimas de anonimas/v0 a lambdas por tu cuenta antes de ver v1.

4. Traza manualmente el switch con dia="MIERCOLES" y predice la salida.

5. Crea una clase `Persona` con equals(), hashCode() y toString(). Pruebala en un HashSet.

6. Modifica inmutable/v2 para que sea verdaderamente inmutable (como v3) y verifica que los cambios externos ya no afectan al objeto.