Guia de Repaso

Tipos de Datos, Arrays y Sobrecarga

Semana 01 - Jueves 12 de Febrero de 2026

Academia Java - Ciudad de Mexico

Temas del dia:

1. API de Fechas: LocalDate, LocalTime y Period
2. Boxing, Autoboxing e Integer Cache
3. BigDecimal vs double
4. Arrays: declaracion, List y clone
5. Sobrecarga de metodos (Overloading)

Instructor: Miguel Rugerio

1. API de Fechas: LocalDate, LocalTime y Period

Java 8 introdujo una nueva API de fechas en el paquete `java.time`. Las clases principales son **inmutables**: cada operacion retorna un objeto nuevo sin modificar el original (igual que String).

Proyecto: dateTime / v0 - Clases principales

Se demuestran las 4 clases principales para representar fechas y horas:

```
System.out.println(LocalDate.now());      // 2026-02-12 (solo fecha)
System.out.println(LocalTime.now());      // 14:30:15.123 (solo hora)
System.out.println(LocalDateTime.now());  // 2026-02-12T14:30:15
System.out.println(ZonedDateTime.now());  //
2026-02-12T14:30:15-06:00[America/Mexico_City]
```

Clase	Contiene	Ejemplo
LocalDate	Solo fecha	2026-02-12
LocalTime	Solo hora	14:30:15
LocalDateTime	Fecha + hora	2026-02-12T14:30:15
ZonedDateTime	Fecha + hora + zona	2026-02-12T14:30:15-06:00

Proyecto: dateTime / v0 - Creacion y manipulacion

Se crean fechas con `of()` y se manipulan con metodos `minus/plus`. Como son inmutables, cada operacion retorna un objeto nuevo:

```
LocalDate date = LocalDate.of(2024, Month.JANUARY, 20);
LocalTime time = LocalTime.of(5, 15);

LocalDateTime dateTime = LocalDateTime.of(date, time)  // Crea objeto
        .minusDays(1)       // Nuevo objeto: resta 1 dia
        .minusHours(10)     // Nuevo objeto: resta 10 horas
        .minusSeconds(30);  // Nuevo objeto: resta 30 segundos
```

```
System.out.println(dateTime);  // 2024-01-18T19:14:30
```

> Las clases de `java.time` son **inmutables**, igual que String. Los metodos `minusDays()`, `plusHours()`, etc. retornan un NUEVO objeto. Si no capturas el retorno, el resultado se pierde.

Inmutabilidad de LocalDate

```
LocalDate date = LocalDate.of(2022, Month.JANUARY, 20);
date = date.plusDays(8);       // DEBE reasignarse
System.out.println(date);      // 2022-01-28

// Mismo patron que String:
String name = "Andronico";
name = name.concat(" Patrobas");  // DEBE reasignarse
System.out.println(name);         // Andronico Patrobas
```

Proyecto: dateTime / v0 - Period (la trampa del chaining)

`Period` representa una cantidad de tiempo en anios, meses y dias. Pero tiene una trampa: sus metodos `ofYears()`, `ofWeeks()`, etc. son **static**, asi que el method chaining NO funciona como se espera.

```
// Period.of() - funciona correctamente
Period period = Period.of(1, 2, 15);  // 1 anio, 2 meses, 15 dias
LocalDate start = LocalDate.of(2022, Month.JANUARY, 1);
LocalDate end = start.plus(period);
System.out.println(end);  // 2023-03-16
```

La trampa del method chaining

```
// CUIDADO: esto NO crea "1 anio y 1 semana"
Period period2 = Period.ofYears(1)
                       .ofWeeks(1);     // ofWeeks es STATIC!

// Equivale a: Period.ofWeeks(1)  (se pierde ofYears)
LocalDate start2 = LocalDate.of(2022, Month.JANUARY, 1);
start2 = start2.plus(period2);
System.out.println(start2);  // 2022-01-08 (solo 1 semana, NO 1 anio)
```

! Los metodos `ofYears()`, `ofMonths()`, `ofWeeks()` son **static**. Encadenarlos con el operador punto NO acumula valores: cada llamada crea un Period NUEVO e independiente. Solo el ultimo se asigna.

2. Boxing, Autoboxing e Integer Cache

Java permite convertir automaticamente entre tipos primitivos y sus clases wrapper. Esto se llama **autoboxing** (primitivo → wrapper) y **unboxing** (wrapper → primitivo).

Proyecto: boxing / v0 - Literales numericos

Java soporta diferentes bases para literales enteros:

```
int decimal     = 129;     // Base 10 (default)
int octal       = 010;     // Base 8 (prefijo 0) = 8 en decimal
int binario     = 0B1010;  // Base 2 (prefijo 0B o 0b)
int hexadecimal = 0xFF;    // Base 16 (prefijo 0X o 0x)

System.out.println(1 + 010);  // 9 (no 11! porque 010 es octal = 8)
```

Base	Prefijo	Ejemplo	Valor decimal
Binario	0B o 0b	0B1010	10
Octal	0	010	8
Decimal	(ninguno)	129	129
Hexadecimal	0X o 0x	0xFF	255

Autoboxing y la trampa del == con Integer

```
int int1 = 129;
int int2 = 129;
System.out.println(int1 == int2);      // true (primitivos: compara valores)

Integer integer1 = 129;  // Autoboxing: int -> Integer
Integer integer2 = 129;  // Autoboxing: int -> Integer
System.out.println(integer1 == integer2);  // false (objetos: compara
referencias)

Integer integer3 = 127;  // Autoboxing
Integer integer4 = 127;  // Autoboxing
System.out.println(integer3 == integer4);  // true (!?)
```

! Java cachea objetos Integer en el rango **-128 a 127** (Integer Cache). Dentro de ese rango, el autoboxing reutiliza el mismo objeto, por lo que == da `true`. Fuera del rango, se crean objetos nuevos y == da `false`. Siempre usa `.equals()` para comparar wrappers.

Casting implicito con byte

```
byte b1 = 5;
byte b2 = 4;

// byte b3 = b1 + b2;         // ERROR: b1+b2 se promueve a int
byte b3 = (byte)(b1 + b2);    // OK: cast explicito

System.out.println(b2 += b1); // 9 - el operador += hace cast implicito
```

Cuando se suman dos `byte`, Java los promueve a `int`. El resultado es `int` y no se puede asignar a `byte` sin cast. Sin embargo, los operadores compuestos (+=, -=) hacen el cast de forma implicita.

Sufijos de literales

```
long l = 10;          // OK: int cabe en long (widening)
Long longx = 10l;     // Necesita sufijo 'l' o 'L' para autoboxing a Long

float f = 5.0f;       // Necesita sufijo 'f' o 'F'
Float floatx = 5.0f; // Autoboxing con sufijo

// double d D  (default para decimales)
// float  f F
// long   l L
```

3. BigDecimal vs double

Los tipos `double` y `float` usan representacion binaria (IEEE 754) que **no puede representar ciertos decimales de forma exacta**. `BigDecimal` ofrece precision arbitraria.

Proyecto: boxing / v0 - Precision decimal

```
double d1 = 0.3;
double d2 = 0.2;
System.out.println("Double:\t 0,3 - 0,2 = " + (d1 - d2));
// Double:  0,3 - 0,2 = 0.09999999999999998

float f1 = 0.3f;
float f2 = 0.2f;
System.out.println("Float:\t 0,3 - 0,2 = " + (f1 - f2));
// Float:   0,3 - 0,2 = 0.10000001
```

Solucion con BigDecimal

```
BigDecimal bd1 = BigDecimal.valueOf(0.02);
BigDecimal bd2 = BigDecimal.valueOf(0.03);
System.out.println(bd1.subtract(bd2));  // -0.01 (exacto)
```

Workaround con double (multiplicar y dividir)

```
double d3 = 0.3;
double d4 = 0.2;
double d5 = (d3*1000 - d4*1000)/1000;
System.out.println("Double:\t 0,3 - 0,2 = " + d5);  // 0.1 (truco)
```

Situacion	Usa	Razon
Dinero, finanzas	BigDecimal	Precision exacta
Graficos, juegos	double	Rendimiento
Calculos cientificos	Depende	Evaluar tolerancia al error

> **!** Nunca uses `new BigDecimal(0.1)` (constructor con double). Usa `new BigDecimal("0.1")` (String) o `BigDecimal.valueOf(0.1)` para evitar que el error del double se propague al BigDecimal.

4. Arrays: Declaracion, List y Clone

Los arrays en Java son objetos de tamano fijo. Se pueden convertir a listas y viceversa, pero hay diferencias importantes entre listas mutables e inmutables.

Proyecto: arrays / v1 - Formas de declarar

```
int[] arreglo1 = new int[3];           // Tamano fijo, valores default (0)
int[] arreglo2 = {0, 1, 2};            // Inicializacion directa
int[] arreglo3 = new int[]{0, 1, 2};   // Inicializacion explicita

System.out.println(Arrays.toString(arreglo2));  // [0, 1, 2]
System.out.println(arreglo3);                    // [I@1a2b3c (referencia!)
```

> `System.out.println(array)` imprime la referencia, no el contenido. Para ver el contenido se usa `Arrays.toString(array)`.

Conversion entre Array y List

```
// Array de primitivos -> NO se puede hacer List<int>
int[] arregloPrim = {0, 1, 2};
List<int[]> listaRara = Arrays.asList(arregloPrim);  // Lista de UN array

// Array de wrappers -> SI funciona correctamente
Integer[] arregloObj = {0, 1, 2};
List<Integer> lista1 = Arrays.asList(arregloObj);        // Vista (tamano
fijo)
List<Integer> lista2 = List.of(arregloObj);               // Inmutable
List<Integer> lista3 = new ArrayList<>(Arrays.asList(arregloObj)); // Mutable

// List -> Array
Integer[] arreglo4 = (Integer[])lista3.toArray();
```

Metodo	Resultado	add/remove?	set?
Arrays.asList()	Lista tamano fijo	NO	SI
List.of()	Lista inmutable	NO	NO

Metodo	Resultado	add/remove?	set?
new ArrayList<>()	Lista mutable	SI	SI

Proyecto: arrays / v1 - Listas inmutables vs mutables

```
List<Integer> lista = List.of(1, 2, 3, 4);  // Inmutable!

lista.add(5);       // UnsupportedOperationException!
lista.remove(0);    // UnsupportedOperationException!
lista.set(1, 22);   // UnsupportedOperationException!
```

! `List.of()` y `Arrays.asList()` crean listas que NO soportan `add()` ni `remove()`. Si necesitas una lista completamente mutable, envuelvela en `new ArrayList<>()`.

Proyecto: arrays / v1 - Clone de arrays bidimensionales

El metodo `clone()` en arrays hace una copia **shallow** (superficial): copia las referencias del primer nivel, pero NO clona los subarrays.

```
int[][] orig = { {1, 2, 3}, {4, 5, 6, 7} };
int[][] dup = orig.clone();
int[] copy = dup[0].clone();

System.out.println(orig == dup);             // false (diferentes arrays)
System.out.println(orig.equals(dup));        // false (equals no
sobreescrito)
System.out.println(orig[0] == dup[0]);       // true  (MISMA referencia!)
System.out.println(dup[0] == copy);          // false (clone creo copia)
System.out.println(dup[0].equals(copy));     // false (equals por referencia)
System.out.println(Arrays.equals(dup[0], copy));  // true (compara contenido)
```

Comparacion	Resultado	Razon
orig == dup	false	clone crea nuevo array exterior
orig[0] == dup[0]	true	Shallow copy: misma referencia interna
dup[0] == copy	false	clone() del subarray crea copia nueva
Arrays.equals(dup[0], copy)	true	Compara contenido elemento por elemento

> Para comparar contenido de arrays usa `Arrays.equals()` (1D) o `Arrays.deepEquals()` (multidimensional). El operador == y el metodo `.equals()` de arrays comparan referencias, no contenido.

Ejercicio: Matriz con compute()

```
public class TestClass {
    int[][] matrix = new int[2][3];
    int[] a = new int[]{1, 2, 3};
    int[] b = new int[]{4, 5, 6};

    public int compute(int x, int y) {
        return a[x] * b[y];     // Multiplica elementos de a y b
    }

    public void loadMatrix() {
        for (int x = 0; x < matrix.length; x++) {
            for (int y = 0; y < matrix[x].length; y++) {
                matrix[x][y] = compute(x, y);  // Llena la matriz
```

```
                    }
                }
            }
        }
```

5. Sobrecarga de Metodos (Overloading)

La sobrecarga permite definir multiples metodos con el **mismo nombre** pero **diferentes parametros**. Java decide cual invocar en **tiempo de compilacion** siguiendo reglas de prioridad.

Proyecto: overloading / v0 - Reglas de resolucion

La clase `Alumno` demuestra la prioridad que sigue Java al resolver cual metodo sobrecargado invocar:

```
public class Alumno {
    String nombre;

    public Alumno(String nombre) {
        this.nombre = nombre;
    }

    // 1. Coincidencia exacta (int, int)        -- COMENTADO
    // void getPromedio(int x, int y) { ... }

    // 2. Widening (int -> double)               -- COMENTADO
    // void getPromedio(double x, double y) { ... }

    // 3. Autoboxing (int -> Integer)            -- COMENTADO
    // void getPromedio(Integer x, Integer y) { ... }

    // 4. Varargs (int...)                       -- ACTIVO
    void getPromedio(int... xs) {
        System.out.println("varargs...");
        int suma = 0;
        for (int x : xs) suma += x;
        System.out.println(suma / xs.length);
    }

    void getPromedio(Double x, Double y) {
        System.out.println((x + y) / 2);
    }
}
```

Orden de prioridad del compilador

Prioridad	Mecanismo	Ejemplo
1 (maxima)	Coincidencia exacta	getPromedio(int, int)
2	Widening (ampliacion)	int → double
3	Autoboxing	int → Integer
4 (minima)	Varargs	int...

! Si hay coincidencia exacta, Java la elige. Si no, intenta widening, luego autoboxing, y como ultimo recurso varargs. Nunca combina widening + autoboxing en un mismo paso.

Proyecto: overloading / v1 - Resolucion con null

Cuando se pasa `null` como argumento, Java elige el metodo con el tipo **mas especifico** en la jerarquia de herencia.

Ejemplo con excepciones

```
public class TestClass {
    public void method(Object o) {
        System.out.println("Object Version");
    }
    public void method(java.io.FileNotFoundException s) {
        System.out.println("FileNotFoundException Version");
    }
    public void method(java.io.IOException s) {
        System.out.println("IOException Version");
    }

    public static void main(String args[]) {
        TestClass tc = new TestClass();
        tc.method(null);  // FileNotFoundException Version
    }
}
```

> Jerarquia: Object → IOException → FileNotFoundException. Con `null`, Java elige el tipo mas especifico (FileNotFoundException), porque es la subclase mas profunda.

Ejemplo con clases propias

```
class Ave extends Object {}
class Pato extends Ave {}

public class TestClass2 {
    public void method(Object o)  { System.out.println("Object Version"); }
    public void method(Ave s)     { System.out.println("Ave Version"); }
    public void method(Pato s)    { System.out.println("Pato Version"); }

    public static void main(String args[]) {
        TestClass2 tc = new TestClass2();
        tc.method(null);  // Pato Version (el mas especifico)
    }
}
```

> **!** Si se agrega `method(String)`, se vuelve **ambiguo** porque String y Pato estan en ramas diferentes de la jerarquia. Ninguno es mas especifico que el otro, y Java da error de compilacion.

Static vs instancia en overloading

```
public class TestClass {
    public static void main(String[] args) {
        new TestClass().sayHello();  // "Hello World " (metodo de instancia)
    }

    public static void sayHelloX() {
        System.out.println("Static Hello World");
    }

    public void sayHello() {
        System.out.println("Hello World ");
    }
}
```

> Un metodo static y uno de instancia pueden coexistir con nombres similares. Desde una instancia se llama al metodo de instancia; el static se invoca con el nombre de la clase.

6. Resumen del Dia

Proyecto	Tema principal	Concepto clave
dateTime v0	API de fechas	LocalDate, LocalTime, inmutabilidad
dateTime v0	Period	Trampa del method chaining static
boxing v0	Boxing / Autoboxing	Integer Cache -128 a 127
boxing v0	BigDecimal vs double	Precision en operaciones decimales
arrays v1	Arrays y listas	Inmutable vs mutable, clone shallow
overloading v0	Sobrecarga	Prioridad: exacta > widening > boxing > varargs
overloading v1	Overloading con null	Tipo mas especifico en jerarquia

Estructura de los proyectos

```
dateTime/src/com/curso/v0/
  Principal.java       -- LocalDate.now(), LocalTime.now(), etc.
  Principal0.java      -- LocalDateTime.of(), minus(), Period
  Principal2.java      -- Inmutabilidad de LocalDate (plusDays)
  Principal3.java      -- Period.of() vs chaining static

boxing/src/com/curso/v0/
  Principal.java       -- Literales, autoboxing, Integer cache, byte cast
  Principal2.java      -- BigDecimal vs double vs float

arrays/src/com/curso/v1/
  Principal.java       -- Declaracion, Arrays.asList, List.of (primitivos)
  Principal2.java      -- Wrappers, ArrayList, toArray
  Principal4.java      -- List.of inmutable (UnsupportedOperationException)
  Principal5.java      -- Clone shallow, Arrays.equals
  TestClass.java       -- Ejercicio de matriz con compute()

overloading/src/com/curso/
  v0/Alumno.java       -- Sobrecarga con varargs y prioridades
  v0/Principal.java    -- Demo autoboxing, sufijos literales
  v0/TestClass.java    -- Static vs instancia
  v1/TestClass.java    -- Overloading con null (jerarquia excepciones)
  v1/TestClass2.java   -- Overloading con null (jerarquia propia)
```

Para repasar

1. Crea un LocalDate y encadena `plusDays()` sin reasignar. Verifica que el original no cambio.

2. Compara dos Integer con valor 127 usando ==, luego cambia a 128. Explica por que el resultado cambia.

3. Crea una `List.of(1,2,3)` e intenta agregar un elemento. Luego envuelvela en `new ArrayList<>()` y reintenta.

4. Descomenta los metodos de `Alumno.java` uno por uno y observa como cambia el metodo que se invoca al llamar `getPromedio(5, 4)`.

5. En `TestClass2.java`, descomenta el metodo `method(String)` y predice que pasa al llamar `tc.method(null)`.

6. Calcula `0.1 + 0.2` con double y con BigDecimal. Compara los resultados.