

# Introducción

- ▶ Materia Algoritmos y Estructuras de Datos
- ▶ Cátedra Schmidt
- ▶ Sintaxis básica del lenguaje

# Contenido de la materia

- Abstracción en el diseño de estructuras de datos y tipos abstractos de datos.
- Análisis de implementaciones sobre estructuras en arreglo y estructuras enlazadas. Criterios de redimensión en estructuras en arreglo. Estructuras de datos básicas: Vector, Pila, Cola, Listas enlazadas, diccionarios.
- Complejidad computacional: cálculo de complejidad computacional para algoritmos iterativos y recursivos simples. Teorema Maestro. Análisis de complejidad amortizado.
- División y Conquista, Algoritmos de ordenamiento no comparativos.
- Tablas de Hashing y resolución de colisiones en tablas de hashing.
- Árboles, árboles binarios de búsqueda, árboles autobalanceados, colas de prioridad.
- Grafos. Características y representaciones de grafos. Implementaciones eficientes de grafos. Recorridos BFS y DFS de grafos. Ordenamiento topológico. Algoritmos de cálculo de caminos mínimos en grafos. Algoritmos de cálculo de árboles de tendido mínimo.

# Calendario

Adjunto en el material de la semana

# Contenido

- ▶ Características
- ▶ Bibliografía
- ▶ Compiladores e IDEs
- ▶ Tipos de datos
- ▶ Comentarios
- ▶ Variables
- ▶ Tipos estructurados
- ▶ Operadores
- ▶ Estructuras de control de flujo
- ▶ Metodos / Funciones
- ▶ Programas
- ▶ Strings
- ▶ Entrada / Salida

# Características

El lenguaje de programación Java fue desarrollado por un equipo de ingenieros de Sun Microsystems, liderado por James Gosling, a principios de la década de 1990. La historia de Java se originó a partir de un proyecto llamado *Green Project*, que tenía como objetivo crear un lenguaje de programación para dispositivos electrónicos pequeños, como televisores, decodificadores y electrodomésticos.

Inicialmente, el lenguaje se llamó "Oak" (roble en inglés), inspirado en un roble que estaba fuera de la oficina de Gosling. Sin embargo, debido a problemas de derechos de autor con un lenguaje ya existente, el nombre fue cambiado a "Java", inspirado en el café de Java, una isla en Indonesia, que era muy popular entre los desarrolladores del proyecto.

El equipo detrás de Java buscaba crear un lenguaje que fuera:

1. **Simple y familiar:** Java se diseñó para ser fácil de aprender y utilizar, especialmente para programadores familiarizados con C y C++.
  2. **Orientado a objetos y TDA:** Java adopta la programación orientada a objetos para permitir la modularidad, reutilización y mantenibilidad del código.
  3. **Robusto y seguro:** Java fue diseñado para minimizar errores en tiempo de ejecución y ofrecer características de seguridad que lo hicieran adecuado para aplicaciones en red.
  4. **Independiente de la plataforma:** El lema de Java, "escribe una vez, ejecuta en cualquier lugar" (*Write Once, Run Anywhere*), reflejaba la intención de que los programas escritos en Java pudieran ejecutarse en cualquier plataforma que tuviera una Máquina Virtual de Java (JVM).
  5. **Eficiente y de alto rendimiento:** Aunque en sus primeras versiones Java fue criticado por su rendimiento, el lenguaje ha mejorado significativamente en este aspecto con el tiempo.
- Java fue lanzado oficialmente en 1995 y rápidamente ganó popularidad, especialmente en el desarrollo de aplicaciones web y empresariales debido a su portabilidad y características avanzadas de seguridad. A lo largo de los años, Java ha evolucionado y se ha convertido en uno de los lenguajes de programación más utilizados en el mundo.

# Bibliografía

- ▶ *Referencia del lenguaje*
  - ▶ <https://dev.java>
  - ▶ <https://docs.oracle.com/en/>
- ▶ *Guía para aprender el lenguaje*
  - ▶ <https://dev.java/learn/getting-started/>

# Compiladores e IDEs

- ▶ Compilador
  - ▶ Cualquier JDK con versión 8 o superior.
- ▶ IDE
  - ▶ Eclipse
  - ▶ IntelliJ
  - ▶ Visual studio code



# Compiladores e IDEs

## Compilador

- Un compilador traduce directamente el código fuente en instrucciones de máquina.

# Compiladores e IDEs

## IDE

- ▶ *Integrated Development Environment*: entorno integrado de desarrollo
- ▶ Aplicación que integra un conjunto de herramientas para el desarrollo de software.
- ▶ Está compuesto por un editor de código, un compilador, un debugger, etc.

# Compiladores e IDEs

## Eclipse

- ▶ IDE: Eclipse
- ▶ Compilador
  - ▶ JDK
- ▶ Descargas
  - ▶ Eclipse:  
<http://www.eclipse.org/downloads/>
  - ▶ JAVA:  
<https://www.oracle.com/ar/java/technologies/downloads/>

# JAVA – Sintaxis basica

## Comentarios

Antes que nada aquí esta como se escriben los comentarios en Java:

```
1 //Comentario de una línea para algo rápido
2
3 /** Comentario largo,
4  *ideal para la Documentación
5  */
```

## API de Java

Podemos entenderlo como un conjunto de muchísimas clases que ya trae hecha Java para ahorrarnos el trabajo. También las conocen como las librerías de Java.

```
1 | import java.Bla.Bla
```

No se preocupen si no le encuentran ningún sentido a esto, solo sigan leyendo.

## Atributos

En Java, hay varias clases de «variables» que podemos ocupar, por eso en Java se usa el término “Tipos de datos” para englobar a cualquier cosa que ocupa un espacio de memoria y que puede ir tomando distintos valores o características durante la ejecución del programa.

En Java diferenciamos dos tipos de datos:

- **Tipos primitivos**, que se corresponden con los tipos de variables en lenguajes como C y que son los datos elementales que hemos citado.
- **Los Tipos Objeto**, son mucho más inteligente y te permiten realizar muchas cosas más fácilmente.



# Tipos Primitivos

Sin métodos, no son objetos, no necesitan una invocación para ser creados. Fáciles y sencillitos.

A continuación te muestro todos los tipos de datos primitivos disponibles en Java:

Nombre	Tipo	Ocupa	Rango
<b>byte</b>	Entero	1 byte	-128 a 127
<b>short</b>	Entero	2 bytes	-32768 a 32767
<b>int</b>	Entero	4 bytes	$2^{31}$
<b>long</b>	Entero	8 bytes	Muy grande
<b>float</b>	Decimal simple	4 bytes	Muy grande
<b>double</b>	Decimal doble	8 bytes	Muy grande
<b>char</b>	Carácter  Usa comillas simples	2 bytes	—
<b>boolean</b>	Valor:  true o false	1 byte	—

Una vez que hayas decidido cual de todos estos vas a ocupar, tienes que seguir este sintaxis (*si, es muy muy parecida a C*).

## Sintaxis

```
1 //FORMA BASICA
2 TipoDeDato NombreDeDato;
3
4 //FORMA DECLARAR VARIAS
5 TipoDeDato Dato1, Dato2, Dato3;
6
7 //FORMA DECLARAR Y INICIALIZAR
8 TipoDeDato Dato1 = Valor;
9 TipoDeDato Dato2 = Valor2, Dato3 = Valor3;
```



# Tipos Objeto

Con métodos, necesitan una invocación para ser creados.

<b>Tipos de la biblioteca estándar de Java</b>	<b>String</b> (cadenas de texto)  Muchos otros (p.ej. Scanner, TreeSet, ArrayList...)
<b>Tipos definidos por el programador</b>	Cualquiera que se nos ocurra, por ejemplo Taxi, Autobus, Tranvia.
<b>Arrays</b>	Serie de elementos o formación tipo vector o matriz.  Lo consideraremos un objeto especial que carece de métodos.
<b>Tipos envoltorio o wrapper</b> (Equivalentes a los tipos primitivos pero como objetos...Cool)	Byte Short Integer Long Float Double Character Boolean

# Operadores

## Matemáticos

Símbolo	Operación
+	Suma
-	Resta
*	Multiplicación
%	Módulo (Resto de división)

# Simplificaciones

Forma Simple	Forma Compleja
$x += y$	$x = x + y$
$x -= y$	$x = x - y$
$x *= y$	$x = x * y$
$x /= y$	$x = x / y$
$x++$	$x = x + 1$
$x--$	$x = x - 1$

# Lógicos

Operador	Significado
!	NOT
>	Mayor que
<	Menor que
==	Igual
&&	AND
	OR
^	XOR
>=	Mayor o igual que
<=	Menor igual que
!=	No igual

## Sentencias de Control

Las sentencias de control son una manera *facil* de hablar de los bucles (*secciones de nuestro código que se repiten muchas veces*) y las selectivas (*cuando el programa decide ejecutar una u otra sección dependiendo de una decisión*) así que empecemos por la primera ¿va?

## Sentencias Selectivas

### IF

La más famosa y fundamental es los **IF's**, estos permiten al programa con base si una condición es cierta ejecutar un bloque de código o no hacerlo.

Se ven así:

### Diferentes sintaxis de IF

```
1  //FORMA BÁSICA
2  if(condicion){
3      sentenciasSiEsVerdad;
4  }
5
6  //COMO OPERADOR TERNARIO
7  (condicion)? sentenciaVerdad:sentenciaFalsa;
8
9  //FORMA IF-ELSE
10 if(condicion){
11     sentenciasSiEsVerdad;
12 }
13 else{
14     sentenciasSiEsFalso;
15 }
16
17 //FORMA IF-ELSE IF- ELSE
18 if(condicion1){
19     sentenciasSiEsVerdad1;
20 }
21 else if(condicion2){
22     sentenciasSiEsVerdad2;
23 }
24 else if(condicion3){
25     sentenciasSiEsVerdad3;
26 }
27 else{
28     sentenciasSiEsFalsoTodo;
29 }
```

## Operadores de Comparación

Lo que va dentro de condición, es muy fácil, es una pregunta que tu le harás a la máquina que podrá contestar como un sí o un no.

Deja primero mostrarte qué operadores pueden comparar:

- **!=** No igual
- **==** Igual
- **<** Menor que
- **>** Mayor que
- **&&** And
- **||** Or
- **<=** Menor o igual que
- **>=** Mayor o igual que

Otra cosa muy útil de saber es que para C/C++, cualquier valor que sea distinto de cero es verdadero, siendo por tanto falso sólo si el valor cero. En JAVA la condición debe ser booleana.

Por lo tanto podemos terminar con algunos ejemplos de condiciones:

```
1  //Ejemplo 1
2  double pi = 3;
3  int variableBasura;
4
5  if(pi < 4){
6      variableBasura = 0;
7  }
8
9  //Ejemplo 2
10 int numero = 3;
11
12 if( numero != 3 ){
13     printf("Nunca se va a ejecutar esto");
14 }
15
```



El condicional switch case es una estructura que evalúa más de un caso y se caracteriza por:

- Selección de una opción entre varias.
- Switch recibe un “caso” y lo evalúa hasta encontrar el caso que corresponda.
- Se puede usar la opción “default” para cuando no se encuentra el caso dado.

Este condicional es útil a la hora de definir por ejemplo un menú de usuario en aplicaciones que se ejecutan por consola.

La estructura en el lenguaje de programación JAVA es:

```
1  switch (variable){
2      case const1:
3          sentencia;
4          ...
5          break;
6
7      case const2:
8          sentencia;
9          ...
10         break;
11
12         ...
13
14     default:
15         sentencia;
16 }
```

```
int i = 2;

switch(i) {
    case 0:
        System.out.println("i es cero.");
        break;
    case 1:
        System.out.println("i es uno.");
        break;
    case 2:
        System.out.println("i es dos.");
        break;
    case 3:
        System.out.println("i es tres.");
        break;
    default:
        System.out.println("i es mayor a tres.");
}
```

## Bubles

### WHILE

Este es sin menor duda la forma más básica de un bucle, lo único que hace es que ejecuta un bloque de código una y otra vez mientras una condición sea verdad.

```
1 | while (condicion){  
2 |     sentencias;  
3 | }
```

La verdad es que lo es, es por eso que es el bucle más poderoso, pero la verdad es que mucha gente lo acaba usando es un estilo más o menos así:

```
int i = 0;           //Este es nuestro contador  
while (i < tope){    //Lo vamos a seguir haciendo tope veces  
    sentencias;      //Ejecutamos algo  
    i++;             //Aumentamos el contador en uno  
}
```

Pero el problema es que como te das cuenta resulta poner mucho código para repetir cosas, por eso se inventó un nuevo bucle.

# FOR

Este es lo mismo que el bucle while, simplemente con otra sintaxis, es más esta es la sintaxis:

```
1 | for (inicialización; condición; incremento){  
2 |     sentencia;  
3 | }
```

Así el último código que vimos con los whiles, se vería así con for:

```
1 | for (i=0; i < tope; i++){  
2 |     sentencia;  
3 | }
```

```
for(int j = 0; j < 10; j++ ) {  
    System.out.println("El valor de i es " + j );  
}
```

## DO WHILE

Al contrario que los bucles for y while que comprueban la condición en lo alto de la misma, el bucle do/while comprueba la condición en la parte baja del mismo, lo cual provoca que el bucle se ejecute como mínimo una vez. La sintaxis del bucle do/while es:

```
1  do{  
2      sentencia;  
3  }  
4  while(condicion);
```

```
i = 0;  
do {  
    System.out.println("El valor de i es " + i++ );  
} while ( i < 10);
```

## Break y Continue

Las sentencias de control break y continue permiten modificar y controlar la ejecución de los bucles anteriormente descritos.

- La sentencia **break** provoca la salida del bucle en el cual se encuentra y la ejecución de la sentencia que se encuentra a continuación del bucle.
- La sentencia **continue** provoca que el programa vaya directamente a comprobar la condición del bucle en los bucles while y do/while, o bien, que ejecute el incremento y después compruebe la condición en el caso del bucle for.

```
i = 0;
while ( i < 10) {
    System.out.println("El valor de i es " + i++ );
    if (i > 5 ) {
        continue;
    }
    if (i > 8 ) {
        break;
    }
}
```

## Funciones del Sistema

Para tener más métodos disponibles, es muy útil ver la API de Java

### Imprimir por Pantalla

Se puede usar Print para imprimir o Println para añadir además una nueva línea al final.

```
System.out.print("Salida por pantalla");  
System.out.println("Salida por pantalla con nueva linea");
```

# Scanner



Es una clase y permite obtener información desde el teclado y lo mete dentro de un atributo.

```
1 //Primero se importa:
2 import java.util.Scanner;
3
4 //Crear un objeto de la Clase Scanner:
5 Scanner NombreDelObjeto = new Scanner(system.in);
6
7 //Obtener datos con Scanner:
8 Atributo1 = NombreDelObjeto.nextDato();
```



```
//Crear un objeto de la Clase Scanner:  
Scanner teclado = new Scanner(System.in);  
  
//Obtener datos con Scanner:  
String texto = teclado.nextLine();  
  
//Muestro el Texto  
System.out.println("Texto ingresado: " + texto);  
  
int numero = teclado.nextInt();  
  
//Muestro el Numero  
System.out.println("Numero ingresado: " + numero);  
  
teclado.close();
```

# Random



Permite generar un número aleatorio.

```
//Crear un objeto de la Clase Random:  
Random random = new Random(System.nanoTime());  
  
//Obtener un Número Random entre 0 y X (x=100):  
int numeroAleatorio = random.nextInt(100);  
  
System.out.println("El numero es: " + numeroAleatorio);
```

# Arrays

## Arrays o Vectores

Un vector sería algo muy similar a una lista, además que siempre se usan junto con bucles. En Java, un array unidimensional se declara como:

```
tipo[] vector = {valores};
```

En Java, el primer elemento de un array es el que posee el índice 0, por lo tanto, un array de 20 elementos posee sus elementos numerados de 0 a 19.

```
int[] vector = {0,0,0,0};  
vector[0] = 8;  
|  
for(int i = 0; i < vector.length; i++) {  
    System.out.println("La posicion " + i + "tiene el valor " + vector[i]);  
}
```

En Java los vectores son instancias de una clase, por lo tanto tienen diversos **métodos** muy útiles:

```
1 | //Longitud del Vector  
2 | nombreVector.length()
```

Los arrays en Java se pueden utilizar de la siguiente manera:

```
int[] vector;  
  
vector = new int[50];  
  
for(int i = 0; i < vector.length; i++) {  
    vector[i] = 0;  
}  
  
for(int i = 0; i < vector.length; i++) {  
    vector[i] = i;  
}  
  
for(int i = 0; i < vector.length; i++) {  
    System.out.println("El valor de la posicion " + i + " es: " + vector[i]);  
}
```

# String

Un string en Java es mucho más fácil de manejar que en C, incluso se puede:

- Comparar directamente
- Añadir más frases con algo tan simple como **String + “bla”**

## Sintaxis:

```
1 | String nombreDelString;
```

## Métodos

En Java los Strings son instancias de una clase, por lo tanto tienen diversos **métodos** muy útiles:

```
1 | //LONGITUD
2 | nombreDelString.length()
3 |
4 | //REGRESAR CHAR EN X POSICION
5 | nombreDelString.CharAt(x)
```

## Matrices

Es de la forma más sencilla un array de arrays

```
1  //DECLARACIÓN
2  tipo[][]nombreMatriz = new tipo[tamañoEnX][tamañoEnY];
3
4  //INICIALIZACION
5  tipo[][]nombreMatriz = {
6      {1,2,3}
7      {4,5,6}
8      ...
9      {7,8,9}
10 };
```

## Métodos

En Java las Matrices son instancias de una clase, por lo tanto tienen diversos **métodos** muy útiles:

```
1  //TAMAÑO EN X
2  nombreMatriz.length
3
4  //TAMAÑO EN Y
5  nombreMatriz[0].length
```

## Try Catch

La sentencia try-catch en Java se utiliza para manejar excepciones, que son eventos que ocurren durante la ejecución de un programa y que pueden interrumpir su flujo normal, como intentar acceder a un archivo que no existe, dividir por cero, o trabajar con un índice fuera de los límites de un array.

```
try {  
    // Código que podría causar una excepción  
} catch (TipoDeExcepcion e) {  
    // Código para manejar la excepción  
}
```

### Desglose

**try:** Contiene el bloque de código que quieres "probar". Es aquí donde pones el código que podría generar una excepción.

**catch:** Este bloque captura y maneja la excepción que haya ocurrido en el bloque try. Puedes especificar diferentes bloques catch para diferentes tipos de excepciones.

**TipoDeExcepcion:** Es la clase de la excepción que estás intentando capturar. Por ejemplo, `IOException`, `NullPointerException`, `ArithmeticException`, etc.

**e:** Es una variable que almacena la instancia de la excepción capturada. Puedes usarla para obtener más información sobre la excepción, como su mensaje de error.

## Ejemplo

Aquí tienes un ejemplo simple en el que se intenta dividir dos números. Si el divisor es cero, se lanza una `ArithmeticException` que se captura y maneja en el bloque `catch`.

```
public class EjemploTryCatch {  
    public static void main(String[] args) {  
        try {  
            int a = 10;  
            int b = 0;  
            int resultado = a / b; // Esto causará una excepción  
        } catch (ArithmeticException e) {  
            System.out.println("Error: No se puede dividir por cero.");  
        }  
    }  
}
```



## Bloque finally (Opcional)

Puedes agregar un bloque finally después de try-catch. El bloque finally se ejecuta siempre, independientemente de si ocurrió una excepción o no. Es útil para liberar recursos, como cerrar un archivo o una conexión de base de datos.

```
try {  
    // Código que podría causar una excepción  
} catch (Exception e) {  
    // Manejo de la excepción  
} finally {  
    // Código que se ejecuta siempre  
}
```

## Ejemplo con finally

```
public class EjemploTryCatchFinally {  
    public static void main(String[] args) {  
        try {  
            int[] numeros = {1, 2, 3};  
            System.out.println(numeros[5]); // Esto causará una excepción  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Error: Índice fuera de los límites del array.");  
        } finally {  
            System.out.println("Este bloque se ejecuta siempre.");  
        }  
    }  
}
```

En Java, puedes manejar diferentes tipos de excepciones usando múltiples bloques catch, cada uno para un tipo específico de excepción. Esto te permite manejar distintos tipos de errores de manera específica. Después de estos bloques específicos, puedes usar un bloque catch general para capturar cualquier otra excepción que no haya sido capturada por los bloques anteriores.

```
try {  
    // Código que podría lanzar varias excepciones  
} catch (TipoDeExcepcion1 e1) {  
    // Manejo de la primera excepción  
} catch (TipoDeExcepcion2 e2) {  
    // Manejo de la segunda excepción  
} catch (Exception e) {  
    // Manejo general de cualquier otra excepción  
}
```

```

public class EjemploMultiplesCatch {
    public static void main(String[] args) {
        try {
            int[] numeros = {1, 2, 3};
            System.out.println(numeros[5]); // Esto causará ArrayIndexOutOfBoundsException
            int resultado = 10 / 0;          // Esto causará ArithmeticException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Índice fuera de los límites del array.");
        } catch (ArithmeticException e) {
            System.out.println("Error: No se puede dividir por cero.");
        } catch (Exception e) {
            System.out.println("Error general: " + e.getMessage());
        }
    }
}

```

## Explicación

**ArrayIndexOutOfBoundsException:** Este bloque captura el error específico de acceder a un índice que está fuera de los límites del array.

**ArithmeticException:** Este bloque captura el error específico de intentar dividir por cero.

**Exception:** Este bloque es general y captura cualquier otra excepción que no haya sido manejada por los bloques anteriores. Exception es la clase base de todas las excepciones en Java, por lo que este bloque puede capturar cualquier tipo de excepción.

Ejemplo:

```
public static double dividir(int numero1, int numero2) throws Exception {  
    if (numero2 == 0) {  
        throw new Exception("No se puede dividir por 0");  
    }  
    return Integer.valueOf(numero1).doubleValue() /  
        Integer.valueOf(numero2).doubleValue();  
}
```

Como primer punto, el método debe avisar las excepciones que explícitamente lanza, con la palabra clave throws. En este caso “throws Exception”.

Luego, cuando se detecta la condición para lanzar una excepción, con la palabra reservada throw se lanza la excepción, la del tipo que se desea.

En este caso throw new Exception(“....”);

## Nota importante

El orden de los bloques catch es crucial. Debes colocar primero los bloques catch para las excepciones más específicas y luego el bloque catch general (Exception). Si pones el bloque catch general antes de los específicos, las excepciones específicas nunca serán capturadas porque el bloque general las atraparé primero, lo que resulta en un error de compilación.

```
try {  
    // Código que podría lanzar varias excepciones  
} catch (Exception e) {  
    // Este bloque general atrapa todas las excepciones  
} catch (ArrayIndexOutOfBoundsException e) {  
    // Error de compilación: código inalcanzable  
}
```

Este orden es incorrecto porque el bloque catch (Exception e) atraparía cualquier excepción, lo que haría que el bloque catch (ArrayIndexOutOfBoundsException e) sea inalcanzable y, por tanto, provocaría un error de compilación.

## Enums constantes

En Java son un tipo especial de clase que representa un grupo fijo de constantes, es decir, valores que no cambian. Se utilizan para representar un conjunto finito de opciones, como días de la semana, colores, direcciones, etc.

### Definición básica

La sintaxis para definir un enum es muy sencilla:

```
public enum Dia {  
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO  
}
```

## Uso básico de enum

Una vez que has definido un enum, puedes utilizarlo de la siguiente manera:

```
public class EjemploEnum {  
    public static void main(String[] args) {  
        Dia dia = Dia.LUNES;  
  
        // Uso en un switch  
        switch(dia) {  
            case LUNES:  
                System.out.println("Es lunes, comienzo de la semana.");  
                break;  
            case VIERNES:  
                System.out.println("Es viernes, casi fin de semana.");  
                break;  
            case DOMINGO:  
                System.out.println("Es domingo, día de descanso.");  
                break;  
            default:  
                System.out.println("Es un día entre semana.");  
                break;  
        }  
    }  
}
```

## Comparación de enum

Puedes comparar valores de enum utilizando el operador == porque los enum en Java son singletons, es decir, una instancia única por valor.

```
if (dia == Dia.LUNES) {  
    System.out.println("Hoy es lunes.");  
}
```

## Ventajas de usar enum

Legibilidad: Mejoran la legibilidad del código al reemplazar valores numéricos o cadenas mágicas con constantes descriptivas.

Mantenimiento: Facilitan el mantenimiento al centralizar un conjunto de constantes en un solo lugar.

Tipo seguro: Previenen errores al no permitir valores fuera del conjunto definido.



## Registros

En Java, no existe una estructura de datos exactamente igual a los structs de C. Sin embargo, puedes lograr una funcionalidad similar usando clases. En Java, una clase puede agrupar diferentes tipos de datos bajo un solo nombre, al igual que un struct en C, pero con muchas más características y capacidades. Este modo incorrecto de utilizar la clase solo se acepta estas 2 semanas hasta ingresar en el tema TDA.

```
class Persona {  
    String nombre;  
    int edad;  
    float altura;  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Persona persona = new Persona();  
        persona.nombre = "Juan";  
        persona.edad = 30;  
        persona.altura = 1.75f;  
  
        System.out.println("Nombre: " + persona.nombre);  
        System.out.println("Edad: " + persona.edad);  
        System.out.println("Altura: " + persona.altura);  
    }  
}
```

# Paquetes

En Java, un paquete (package) es un mecanismo que organiza y agrupa clases e interfaces relacionadas en un espacio de nombres específico. Los paquetes ayudan a evitar conflictos de nombres entre clases, mejoran la modularidad y facilitan la organización del código en grandes proyectos.

## Conceptos Clave sobre los Paquetes

**Agrupación Lógica:** Los paquetes permiten agrupar clases relacionadas de forma lógica. Por ejemplo, todas las clases relacionadas con la entrada y salida (I/O) están en el paquete `java.io`. Control de

**Acceso:** Los paquetes también proporcionan un nivel de control de acceso. Por ejemplo, las clases o miembros de clases con acceso "package-private" (es decir, sin un modificador de acceso explícito) solo son accesibles dentro del mismo paquete.

**Estructura de Directorios:** Cada paquete en Java está directamente relacionado con una estructura de directorios en el sistema de archivos. Por ejemplo, un paquete `com.miempresa.miapp` correspondería a un directorio `com/miempresa/miapp/`.

**Evitar Colisiones de Nombres:** Al organizar las clases en paquetes, puedes tener clases con el mismo nombre en diferentes paquetes sin que entren en conflicto. Por ejemplo, puedes tener `com.miempresa.producto.Cliente` y `com.otroempresa.crm.Cliente`.

## Definición de un Paquete

Para definir un paquete en una clase, debes usar la palabra clave `package` al principio del archivo fuente:

```
package com.miempresa.miapp;  
  
public class MiClase {  
    // Código de la clase  
}
```

## Estructura del Proyecto

Supongamos que tienes un paquete llamado `com.miempresa.miapp`. Este paquete contiene dos clases: `MiClase` y `OtraClase`. La estructura del proyecto sería algo como:

```
src/  
├── com/  
│   └── miempresa/  
│       └── miapp/  
│           ├── MiClase.java  
│           └── OtraClase.java
```

## Uso de Clases desde un Paquete

Para utilizar una clase de un paquete en otro archivo, puedes hacer lo siguiente:

### 1. Importar la clase específica:

```
import com.miempresa.miapp.MiClase;

public class Principal {
    public static void main(String[] args) {
        MiClase obj = new MiClase();
        // Uso de MiClase
    }
}
```

### 2. Importar todas las clases del paquete:

```
import com.miempresa.miapp.*;

public class Principal {
    public static void main(String[] args) {
        MiClase obj1 = new MiClase();
        OtraClase obj2 = new OtraClase();
        // Uso de las clases
    }
}
```

### 3. Uso sin importar (usando el nombre completo):

```
public class Principal {  
    public static void main(String[] args) {  
        com.miempresa.miapp.MiClase obj = new com.miempresa.miapp.MiClase();  
        // Uso de MiClase  
    }  
}
```

#### Paquetes Especiales en Java

java.lang: Este paquete es el paquete base de Java y se importa automáticamente. Contiene clases fundamentales como String, Math, Integer, System, etc.

java.util: Contiene clases de utilidades como colecciones (ArrayList, HashMap), fechas (Date, Calendar), etc.

java.io: Contiene clases para realizar operaciones de entrada y salida, como File, InputStream, OutputStream, etc.

## Funciones o Métodos

En Java, las funciones (o métodos, como se les llama en el contexto de la programación orientada a objetos) se declaran dentro de las clases. Un método en Java define un bloque de código que realiza una tarea específica y puede ser invocado desde otras partes del programa.

### Sintaxis de Declaración de un Método

```
modificador_de_acceso tipo_de_retorno nombre_del_metodo(parametros) {  
    // Cuerpo del método  
}
```

### Desglose de la Sintaxis

**Modificador de acceso:** Define la visibilidad del método (si puede ser llamado desde fuera de su clase). Por ahora utilizaremos public.

**Tipo de retorno:** Especifica el tipo de dato que el método devuelve. Si el método no devuelve ningún valor, se usa la palabra clave void.

**Nombre del método:** Debe ser un identificador válido y descriptivo de la tarea que realiza el método.

**Parámetros:** Una lista opcional de parámetros (variables) que el método recibe como entrada. Los parámetros se declaran con su tipo y nombre. Si no hay parámetros, se dejan los paréntesis vacíos.

**Cuerpo del método:** El bloque de código encerrado entre llaves {} que define las acciones que realiza el método.

## Ejemplo Básico de Métodos en Java

Por ahora, durante estas 2 semanas, utilizaremos esta forma de llamar a los métodos, hasta ver el tema TDA

```
public class Principal {  
  
    // Método que suma dos números  
    public static int sumar(int a, int b) {  
        return a + b;  
    }  
  
    // Método que imprime un saludo  
    public static void saludar() {  
        System.out.println("¡Hola, bienvenido a la calculadora!");  
    }  
  
    public static void main(String[] args) {  
  
        // Llamada al método saludar  
        saludar();  
  
        // Llamada al método sumar  
        int resultado = sumar(5, 3);  
        System.out.println("La suma es: " + resultado);  
    }  
}
```

Tipo Fecha, alternativa Moderna: `java.time` (Java 8 y Posterior)

Debido a las limitaciones, se recomienda utilizar las clases del paquete `java.time`, que forman parte de la API de fecha y hora de Java 8 en adelante. Estas clases son más poderosas, seguras y fáciles de usar.

Algunas de las clases más utilizadas en `java.time` son:

`LocalDate`: Representa una fecha (solo año, mes, y día) sin hora.

`LocalTime`: Representa una hora (solo hora, minutos, segundos, y nanosegundos) sin fecha.

`LocalDateTime`: Combina una fecha y una hora.

`ZonedDateTime`: Representa una fecha y hora con una zona horaria específica.

`Instant`: Representa un instante en la línea de tiempo (similar a `Date` en milisegundos desde la época Unix).



## Ejemplo:

```
import java.time.LocalDate;
import java.time.LocalDateTime;

public class EjemploJavaTime {
    public static void main(String[] args) {
        // Fecha actual
        LocalDate fechaActual = LocalDate.now();
        System.out.println("Fecha actual: " + fechaActual);

        // Fecha y hora actuales
        LocalDateTime fechaHoraActual = LocalDateTime.now();
        System.out.println("Fecha y hora actuales: " + fechaHoraActual);

        // Crear una fecha específica
        LocalDate fechaEspecifica = LocalDate.of(2021, 1, 1);
        System.out.println("Fecha específica: " + fechaEspecifica);
    }
}
```

## Conversion de String a Date

```
LocalDateTime fechaHoraActual = LocalDateTime.now();  
DateTimeFormatter formatoHora = DateTimeFormatter.ofPattern("HH-mm-ss");  
  
System.out.println("La hora actual es: " + fechaHoraActual.format(formatoHora));
```

Primero se crea el patron a formatear, y luego se formatea una hora en particular para transformarla en texto.

Les dejo la referencia para crear patrones:

<https://dev.java/learn/date-time/parsing-formatting/>

<https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/time/format/DateTimeFormatter.html>

## Leer un archivo de Texto:

Para leer un archivo .txt línea por línea en Java, puedes utilizar la clase `BufferedReader`, que proporciona un método conveniente para leer texto de un archivo. Aquí te dejo un ejemplo sencillo:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class LeerArchivo {
    public static void main(String[] args) {
        String rutaArchivo = "ruta/del/archivo.txt"; // Cambia esta ruta por la ruta

        try (BufferedReader br = new BufferedReader(new FileReader(rutaArchivo))) {
            String linea;
            while ((linea = br.readLine()) != null) {
                // Procesa cada línea leída
                System.out.println(linea);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explicación del código:

`FileReader`: Se usa para leer el archivo especificado.

`BufferedReader`: Envuelve al `FileReader` y proporciona el método `readLine()` para leer el archivo línea por línea.

`try-with-resources`: Se utiliza para asegurar que el archivo se cierre automáticamente después de que se complete la lectura.

`readLine()`: Este método devuelve `null` cuando llega al final del archivo, lo que indica que se debe salir del bucle.

Este código imprimirá cada línea del archivo en la consola. Puedes modificar la sección donde se imprime la línea (`System.out.println(linea)`) para procesar cada línea de acuerdo a tus necesidades.

## Escribir un archivo de Texto

Para escribir en un archivo línea por línea en Java, puedes utilizar la clase `BufferedWriter`, que proporciona métodos eficientes para escribir texto en un archivo. Aquí te dejo un ejemplo:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class EscribirArchivo {
    public static void main(String[] args) {
        String rutaArchivo = "ruta/del/archivo.txt"; // Cambia esta ruta por la ruta

        try (BufferedWriter bw = new BufferedWriter(new FileWriter(rutaArchivo))) {
            // Escribe líneas en el archivo
            bw.write("Primera línea");
            bw.newLine(); // Salto de línea
            bw.write("Segunda línea");
            bw.newLine();
            bw.write("Tercera línea");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## Explicación del código:

**FileWriter:** Se utiliza para abrir el archivo en modo de escritura. Si el archivo no existe, se crea uno nuevo. Si ya existe, se sobrescribirá a menos que utilices el constructor con el parámetro `true` (es decir, `new FileWriter(rutaArchivo, true)`) para agregar contenido en lugar de sobrescribir.

**BufferedWriter:** Envuelve al `FileWriter` y proporciona el método `write()` para escribir texto en el archivo.

**`newLine()`:** Este método inserta un salto de línea en el archivo, lo que equivale a `\n`.

**`try-with-resources`:** Asegura que el `BufferedWriter` se cierre automáticamente después de que se complete la escritura, liberando recursos y asegurando que todos los datos se escriban correctamente en el archivo.

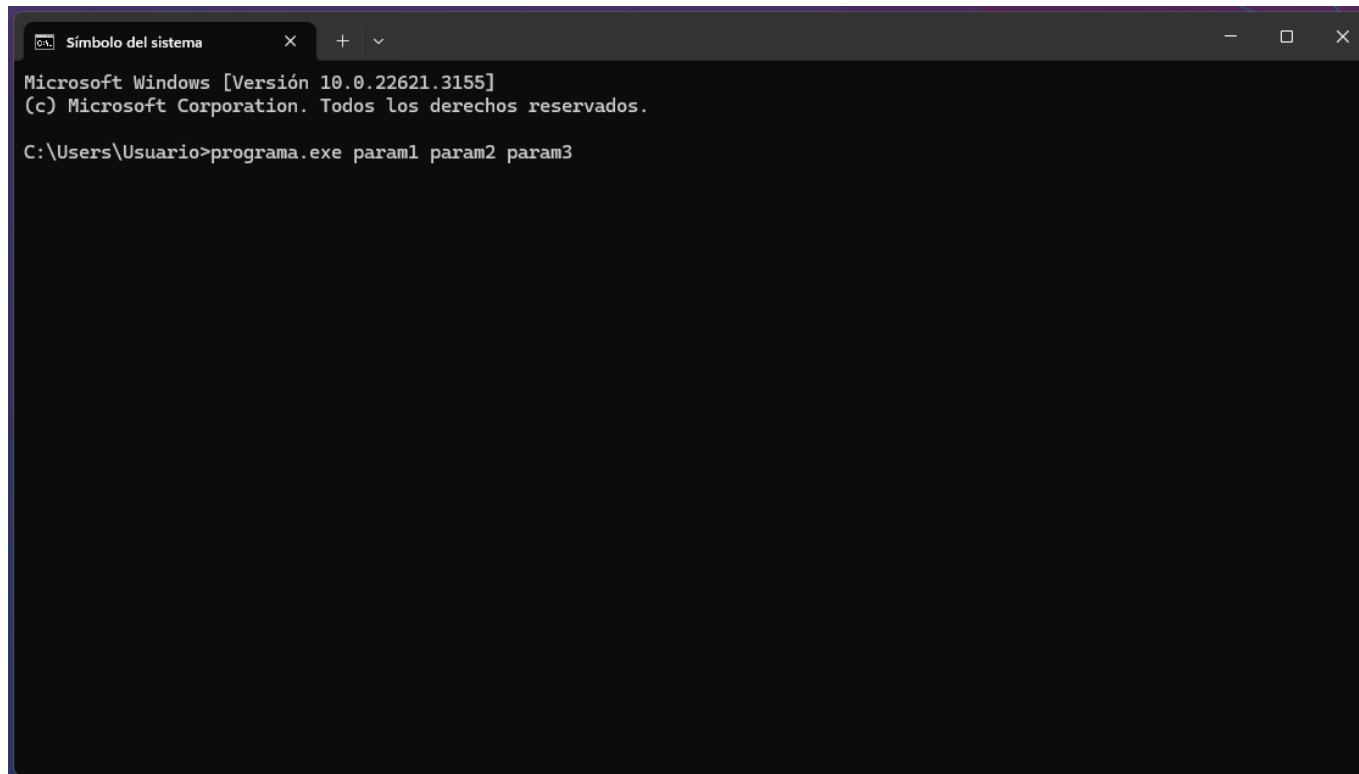
Este código escribirá tres líneas en el archivo. Puedes modificar el código para escribir el contenido que desees en el archivo.

# Programas

- ▶ Archivos .java (para código) y archivos .class para los compilados
- ▶ Punto de entrada al programa principal
- ▶ Metodo main

```
public static void main(String[] args) {  
  
}
```

# Programas para consola y ejecución



A screenshot of a Windows Command Prompt window. The title bar at the top reads 'Símbolo del sistema' and includes standard window controls (minimize, maximize, close). The main area of the window is black with white text. The text displayed is as follows:

```
Microsoft Windows [Versión 10.0.22621.3155]  
(c) Microsoft Corporation. Todos los derechos reservados.  
  
C:\Users\Usuario>programa.exe param1 param2 param3
```



# Conversiones

```
public class Conversiones {  
    public static void main(String[] args) {  
        // int a String  
        int numero = 1234;  
        String cadena1 = String.valueOf(numero);  
        String cadena2 = Integer.toString(numero);  
        String cadena3 = numero + "";  
  
        System.out.println("int a String:");  
        System.out.println("String.valueOf: " + cadena1);  
        System.out.println("Integer.toString: " + cadena2);  
        System.out.println("Concatenación: " + cadena3);  
  
        // String a int  
        String cadenaNumero = "5678";  
        int numero1 = Integer.parseInt(cadenaNumero);  
        int numero2 = Integer.valueOf(cadenaNumero); // Desempaquetado automático  
  
        System.out.println("\nString a int:");  
        System.out.println("Integer.parseInt: " + numero1);  
        System.out.println("Integer.valueOf: " + numero2);  
    }  
}
```

# Compilación: Contenido

El proceso de compilación en Java es el procedimiento mediante el cual el código fuente escrito en el lenguaje Java se convierte en un formato ejecutable por la Máquina Virtual de Java (JVM). Este proceso consta de varios pasos clave que transforman el código fuente en bytecode, que es el formato que la JVM puede interpretar y ejecutar.

## Pasos en el Proceso de Compilación de Java

### 1) Escritura del Código Fuente:

Los desarrolladores escriben código Java en archivos con la extensión .java. Estos archivos contienen definiciones de clases, interfaces y métodos escritos en el lenguaje de programación Java.

### 2) Compilación:

El compilador de Java (javac) se utiliza para convertir los archivos .java en bytecode. Durante la compilación, el compilador verifica la sintaxis y el cumplimiento de las reglas del lenguaje Java. Si hay errores en el código fuente, el compilador los reportará y la compilación fallará. Si la compilación es exitosa, el compilador genera archivos .class que contienen el bytecode. Ejemplo de comando de compilación:

```
javac MiClase.java
```

Este comando generará un archivo MiClase.class si MiClase.java no tiene errores.

### 3) Verificación del Bytecode:

Una vez que el bytecode se ha generado, pasa por un proceso de verificación antes de ser ejecutado por la JVM. La verificación del bytecode es un mecanismo de seguridad que asegura que el código no contiene instrucciones ilegales o peligrosas que podrían comprometer la seguridad del entorno de ejecución.

### 4) Carga de Clases:

La JVM carga las clases necesarias para ejecutar el programa. La clase principal (la que contiene el método main) se carga primero. Durante este proceso, el ClassLoader de la JVM se encarga de encontrar y cargar las clases requeridas desde diferentes fuentes como el sistema de archivos o archivos JAR.

### 5) Linking:

En esta fase, la JVM combina las clases cargadas y resuelve las referencias a otros componentes (métodos, variables, etc.). Se divide en tres subprocesos:

**Verificación:** Asegura que las clases y métodos cumplen con las reglas de la JVM.

**Preparación:** Asigna memoria para variables estáticas y configura sus valores iniciales por defecto.

**Resolución:** Resuelve las referencias simbólicas (nombres de métodos, variables, etc.) en direcciones de memoria reales.

## 6) Ejecución:

Después de la carga y linking, la JVM ejecuta el bytecode. La JVM interpreta el bytecode o lo compila a código nativo utilizando el Just-In-Time (JIT) compiler para mejorar el rendimiento. El método `main(String[] args)` es el punto de entrada para la ejecución del programa.

## 7) Garbage Collector:

Durante la ejecución, la JVM administra la memoria automáticamente mediante el proceso de recolección de basura (Garbage Collector). La recolección de basura libera la memoria ocupada por objetos que ya no se utilizan, lo que ayuda a evitar fugas de memoria.

# Fin