

# PROGRAMACIÓN ORIENTADA A OBJETOS **CON JAVA 17**



[www.consultec-ti.com](http://www.consultec-ti.com)

# Agenda del día

# 1



Prueba de diagnóstico en programación con Java



Visión general de la plataforma Java



Introducción al lenguaje de programación Java



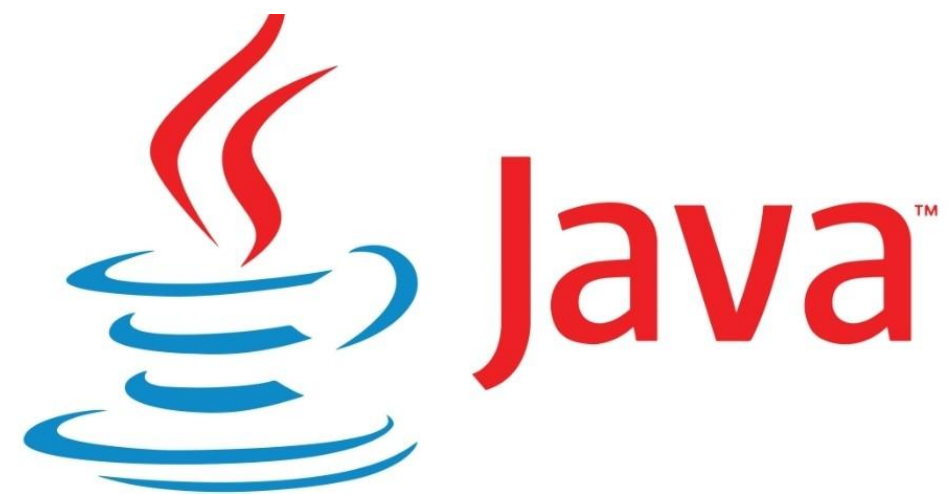
Fundamentos de programación en Java



Programación orientada a objetos.

# Prueba de diagnóstico

# Visión general e **Introducción a JAVA**



Es el lenguaje de programación y la plataforma de desarrollo número uno a nivel empresarial. Reduce costes, acorta los plazos de desarrollo, impulsa la innovación y mejora los servicios de las aplicaciones.

James Gosling creó Java en 1995 mientras trabajaba en Sun Microsystems, el cual fue comprado por Oracle quien sigue su mantenimiento y por tanto evoluciona constantemente.

Hay millones de desarrolladores que ejecutan más de 51.000 millones de máquinas virtuales Java en todo el mundo, por lo que Java sigue siendo la plataforma de desarrollo preferida de empresas y desarrolladores.





# Plataformas JAVA

1

## JAVA SE Standar Edition

En principio cuando se emplea lenguaje de programación Java, se inicia con las APIs de Java SE. La API de Java SE proporciona la funcionalidad básica del lenguaje de programación Java. Define desde los tipos y objetos básicos del lenguaje de programación Java hasta las clases de alto nivel que se utilizan para redes, seguridad, acceso a bases de datos, desarrollo de interfaces gráficas de usuario (GUI) y análisis sintáctico de XML.

Contiene todas las librerías y APIs que cualquier programador Java debe aprender (java.lang, java.io, java.math, java.net, java.util, etc).


2

## Java EE Enterprise Edition

La plataforma Java EE está construida sobre la plataforma Java SE. La plataforma Java EE proporciona una API y un entorno de ejecución para desarrollar y ejecutar aplicaciones de red a gran escala, multinivel, escalables, fiables y seguras.

Añade bibliotecas que proporcionan funcionalidad para desplegar software Java de varios niveles, distribuido y tolerante a fallos, basado principalmente en componentes modulares que se ejecutan en un servidor de aplicaciones.

# Índice TIOBE para **Marzo del 2023**

Mar 2023	Mar 2022	Change	Programming Language		Ratings	Change
1	1			Python	14.83%	+0.57%
2	2			C	14.73%	+1.67%
3	3			Java	13.56%	+2.37%
4	4			C++	13.29%	+4.64%
5	5			C#	7.17%	+1.25%
6	6			Visual Basic	4.75%	-1.01%
7	7			JavaScript	2.17%	+0.09%
8	10	^		SQL	1.95%	+0.11%
9	8	v		PHP	1.61%	-0.30%
10	13	^		Go	1.24%	+0.26%

# Ventajas de Java

1

**Es un lenguaje de programación multiplataforma**

Puede utilizarse para desarrollar aplicaciones que se ejecutan en diferentes plataformas o sistemas operativos, como Windows, Mac, Linux, iOS, Android, etc.

2

**Ofrece una amplia gama de librerías y herramientas**

Enfoque con el principio de programación DRY (Don't Repeat Yourself), además estas bibliotecas permiten a los programadores reutilizar código ya escrito y probado en lugar de tener que escribir todo el código desde cero, lo que ahorra tiempo y reduce errores.

3

**Cuenta con un sistema de seguridad incorporado**

Conjunto de mecanismos que ayudan a proteger las aplicaciones Java de posibles amenazas de seguridad, como virus, malware, ataques de hackers y otros tipos de intrusos. Estos mecanismos proporcionan una capa adicional de protección a los programas Java y a sus usuarios.

4

**Es un lenguaje de programación orientado a objetos**

Permite a los programadores reutilizar el código existente. Por ejemplo, una clase definida para un objeto puede ser utilizada en otro programa Java, lo que ahorra tiempo y esfuerzo en la creación de un nuevo código.

5

**Es un lenguaje de programación de alto nivel**

En lugar de trabajar con bits y bytes, como lo hacen los lenguajes de bajo nivel — como el lenguaje ensamblador—, los lenguajes de alto nivel utilizan estructuras de datos complejas y abstracciones para representar problemas y soluciones.



# Desventajas de Java

1

## Tiene un rendimiento más lento

Tiene un rendimiento más lento en comparación con otros lenguajes de programación, como C y C++, por el proceso de compilación y ejecución, ya que el mismo puede ralentizar el rendimiento del programa en comparación con lenguajes que se compilan directamente en código de máquina.

2

## Puede requerir más memoria que otros lenguajes

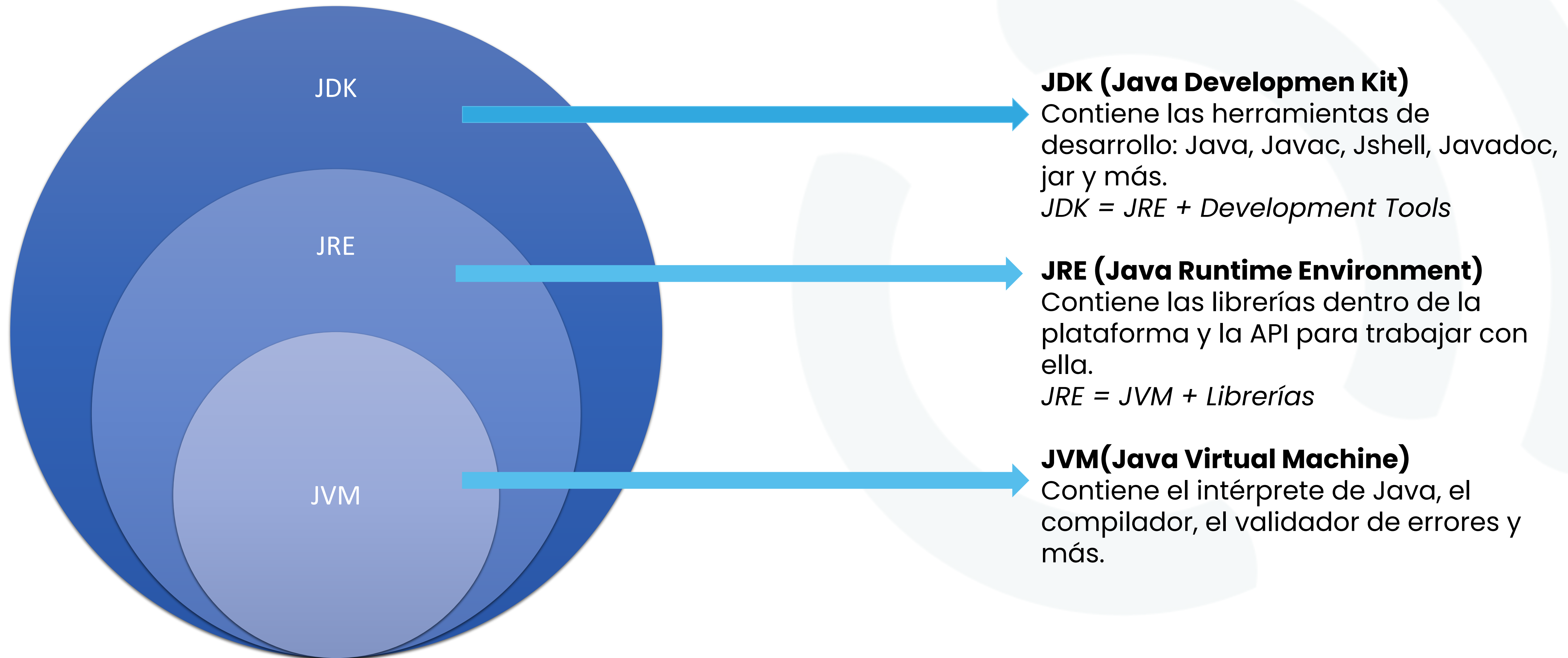
La JVM se encarga de asignar y liberar memoria según sea necesario para el programa en tiempo de ejecución. Aunque esto ofrece muchas ventajas, como la eliminación de problemas comunes de memoria como fugas de memoria y referencias no válidas, también puede aumentar el uso de memoria del programa, ya que la JVM necesita tener una cantidad considerable de memoria disponible para realizar su trabajo.

3

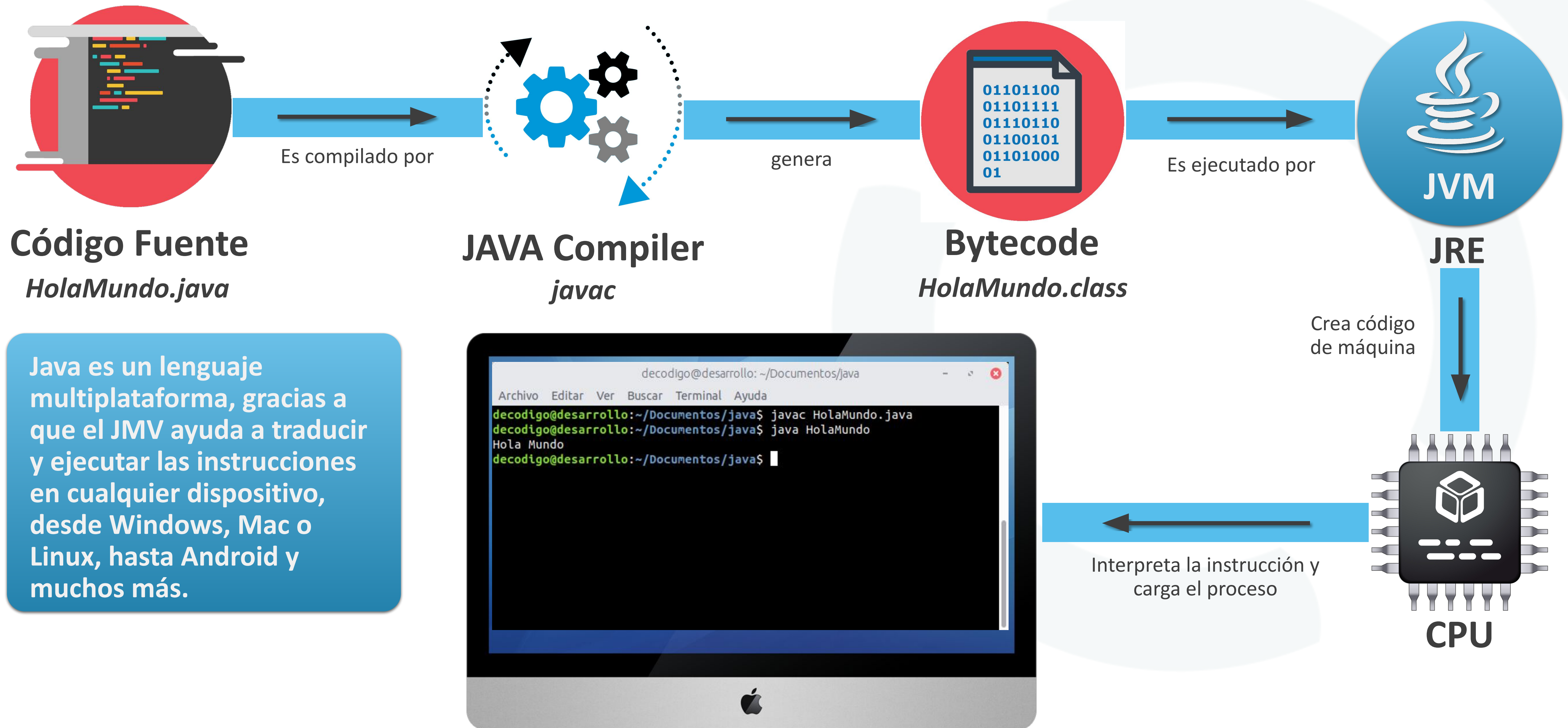
## La programación en Java tiene sobrecarga de código

La sobrecarga de código ocurre cuando se escriben múltiples versiones de un método o una función con diferentes parámetros o tipos de datos de entrada, y cada versión realiza una tarea ligeramente diferente.

# Visión general de la **plataforma Java**



# Proceso de ejecución de un Programa Java





# Herramientas de Desarrollo Java

## Versiones LTS



## Herramientas de gestión y construcción de proyectos



## IDEs de Desarrollo



Receso

**15** min





# Fundamentos de programación en **JAVA**

# Estructura de Código

Importaciones

Nombre de la Clase

Método *Main*

Variables

Notaciones

Métodos

```
import java.util.ArrayList;
import java.util.List;

public class HolaMundo {
    public static void main(String[] args) {
        String usuario1;
        usuario1 = "David";
        List<String> usuario = new ArrayList<String>();
    }

    @Override
    public String toString() {
        return super.toString();
    }
}
```

# Variables

Una variable es un contenedor de datos que almacena algún valor durante la ejecución de un programa, tomando en cuenta que cuando se declara una variable el mismo toma un espacio de la memoria RAM. Por tanto:

- Una variable se compone de un nombre único.
- Su valor puede cambiar a lo largo de la ejecución de un programa (son reutilizables).

```
//Convencion para variables comunes o modificables y
// metodos (Lower Camel Case)
int dias = 2;
String nombreCliente = "valor";
String $nombreCajero = "valor2";
String _nombreCajero = "valor3";

//Convencion Mayuscula para constante
double TAX = 0.12;
double TAX_SEC = 0.30;

//Convencion de clase (Upper Camel case)
class NombreClase {};
```



# Tipos de Variables Primitivas

## Tipos de datos para números enteros (sin decimales):

1. **byte**: Ocupa 1 byte de memoria y su rango es de **-128** hasta **127**.
2. **short**: Ocupa 2 bytes de memoria y su rango es de **-32,768** hasta **32,727**.
3. **int**: Ocupa 4 bytes de memoria y su rango es de **-2,147,483,648** hasta **2,147,483,647**. Puede almacenar hasta 10 dígitos.
4. **long**: Ocupa 8 bytes de memoria y su rango es de **-9,223,372,036,854,775,808** hasta **9,223,372,036,854,775,807**.  
Para diferenciarlo de un tipo de dato **long** debemos terminar el número con la letra **L**.

## Tipos de datos para números flotantes (con decimales)

1. **float**: Ocupan 4 bytes de memoria y su rango es de **1.40129846432481707e-45** hasta **3.40282346638528860e+38**.  
Así como **long**, debemos colocar una letra **F** al final.
2. **double**: Ocupan 8 bytes de memoria y su rango es de **4.94065645841246544e-324d** hasta **1.79769313486231570e+308d**.

# Tipos de Variables Primitivas

## Tipos de datos char y boolean:

1. **char:** Ocupa 2 bytes y sólo puede almacenar 1 dígito, debemos usar comillas simples en vez de comillas dobles.
2. **boolean:** Son un tipo de dato lógico, solo aceptan los valores true y false. También ocupa 2 bytes y almacena únicamente 1 dígito.

Hay un tipo de dato **String** para el manejo de cadenas que no es en sí un tipo de dato primitivo. Con el tipo de dato **String** podemos manejar cadenas de caracteres separadas por dobles comillas.

El elemento **String** es un tipo de dato inmutable. Es decir, que una vez creado, su valor no puede ser cambiado.

El **String** no es un tipo de dato primitivo del lenguaje Java. Pero su uso es igual de importante que el de los tipos de datos revisados aquí. Veremos más en detalle el uso del tipo **String**.



# Tipos de datos estructurados

---

Se denominan así porque en su mayor parte están destinados a **contener múltiples valores de tipos más simples**, primitivos.

## Cadenas de caracteres

Una cadena de texto no deja de ser más que la sucesión de un conjunto de caracteres alfanuméricos, signos de puntuación y espacios en blanco, representadas en Java con la clase **String** y **StringBuffer**.

Cuando declaramos una cadena estamos creando un objeto (Clase String), su declaración no se diferencia de la de una variable de tipo primitivo, ya que la creación de la clase se establece implícitamente.

```
String claseString = "valor";
```

# Vectores o *arrays*

---

Son colecciones de datos de un mismo tipo, sea primitivos, estructurados o definidos por el usuario.

Un vector es una estructura de datos en la que a cada elemento le corresponde una posición identificada por uno o más índices numéricos enteros

## **Tipos definidos por el usuario:**

En Java existen infinidad de clases en la plataforma, **y de terceros**, para realizar casi cualquier operación o tarea que se pueda ocurrir: *leer y escribir archivos, enviar correos electrónicos, ejecutar otras aplicaciones o crear cadenas de texto más especializadas*, entre un millón de cosas más.

Todas esas clases son tipos estructurados también, pero creadas por empresas, comunidades, o crear tus propias clases para hacer todo tipo de tareas o almacenar información. **Serían tipos estructurados definidos por el usuario.**

# Tipos envoltório o *wrapper*

---

Java cuenta con tipos de datos estructurados equivalentes a cada uno de los tipos primitivos que hemos visto. por ejemplo:

*Para representar un entero de 32 bits (int) de los que hemos visto al principio, Java define una clase llamada **Integer** que representa y "envuelve" al mismo dato pero le añade ciertos métodos y propiedades útiles por encima.*

Los tipos "**envoltorio**" buscan facilitar el uso de esta clase de valores allí donde se espera un dato por referencia (un objeto) en lugar de un dato por valor.

Estos tipos equivalentes a los primitivos pero en forma de objetos son: **Byte, Short, Integer, Long, Float, Double, Boolean y Character (8 igualmente).**

# Localización de las Variables

Java tiene tres localizaciones de variables: de instancia, de clase y locales.

## 1. Variables de instancia:

Se utilizan para definir los atributos de un objeto.

## 2. Variables de clase:

Son similares a las variables de instancia, con la excepción de que sus valores son los mismos para todas las instancias de la clase.

## 3. Variables locales:

Se declaran y se utilizan dentro de las definiciones de los métodos.

A diferencia de otros lenguajes, Java no tiene variables globales, es decir, variables que son vistas en cualquier parte del programa.

# Conversión de tipos de datos

El casteo (casting) es un procedimiento para transformar una variable primitiva de un tipo a otro, o transformar un objeto de una clase a otra clase siempre y cuando haya una relación de herencia entre ambas.

Existen distintos tipos de casteo (casting) de acuerdo a si se utilizan tipos de datos o clases:

1. *Casteo Implícito (Widening Casting):*
2. *Casteo Explícito (Narrowing Casting)*

Existen 2 maneras de interpretar el casteo:

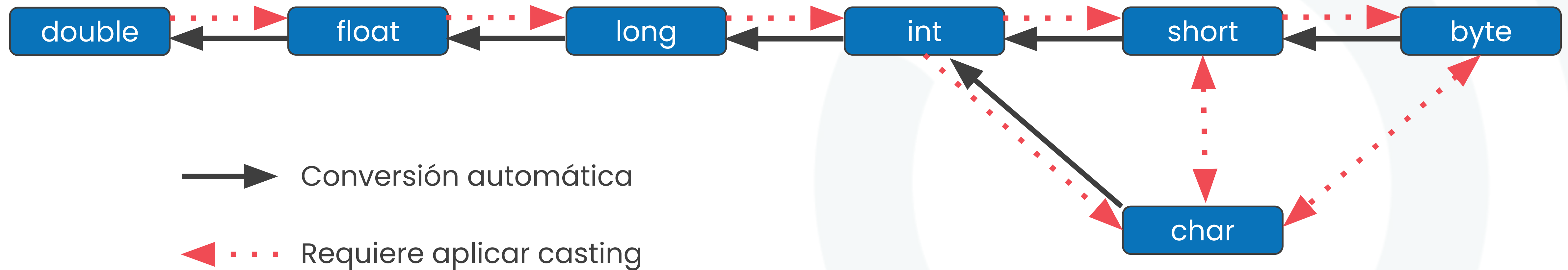
3. *Estimación.*
4. *Exactitud.*

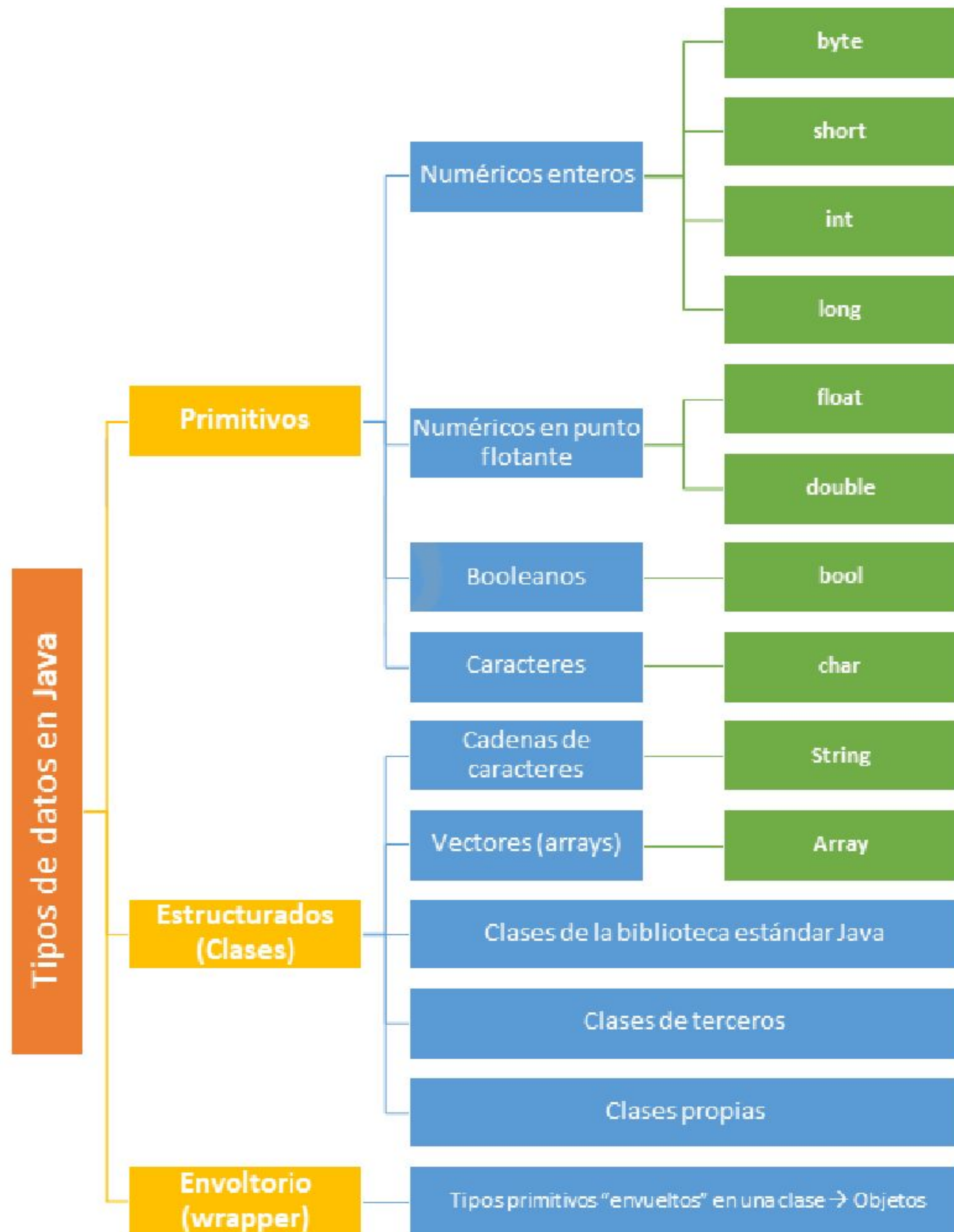
```
int numeroEntero = 100;  
long numeroLargo = numeroEntero;  
  
byte b = 50;  
//casting de tipo int a byte  
//mostraría error si no detallamos (byte)  
b = (byte)(b * 2);  
System.out.println(b);
```



# Conversión de tipos de datos

Cuando se asigna valor de un tipo de datos a otro, los dos tipos de datos pueden no ser compatibles entre sí y se debe hacer un casting. Por ejemplo, podemos asignar un valor **int** a una variable **long**.





El **JVM** trata de distintas maneras de interpretar los tipos de datos en "ceros y unos" en función de ciertas configuraciones que establecen el espacio utilizado de memoria, así como la representación aplicada para codificar y decodificar esa información.

# Mejoras desde Java SDK 10

A partir de Java 10 solo debemos escribir la palabra reservada **var** y Java definirá el tipo de dato de nuestras variables automáticamente, a esto se le llama ***inferencia de tipos de datos***:

```
var _integer = 1234567890; // INT
var _long = 12345678901L; // LONG

var _double = 123.456; // DOUBLE
var _float = 123.456F; //FLOAT

var _string = "Cadena de Texto"; //STRING

var list = new ArrayList<String>(); // LIST
var stream = list.stream(); //COLLECTION
```

Tomar en cuenta que NULL es un valor inferible e igual que un valor vacío.



# Expresiones y Operadores

---

## Expresión:

Una expresión es una combinación de variables, operadores y llamadas de métodos construida de acuerdo a la sintaxis del lenguaje que devuelve un valor.

El tipo de dato del valor regresado por una expresión depende de los elementos usados en la expresión.

## Operadores:

Los operadores son símbolos especiales que por lo común se utilizan en expresiones.

Ambos contenidos se expandirán con mayor detalle más adelante.

# Operadores

## Operadores aritméticos

Operador	Significado	Ejemplo
+	suma	$a + b$
-	resta	$a - b$
*	multiplicación	$a * b$
/	división	$a / b$
%	módulo	$a \% b$

## Operadores asignación

Operador	Significado	Ejemplo
=	asignación	$a = b$
+=	suma	$a += b \ (a = a + b)$
-=	resta	$a -= b \ (a = a - b)$
*=	multiplicación	$a *= b \ (a = a * b)$
/=	división	$a / b \ (a = a / b)$
%=	módulo	$a \% b \ (a = a \% b)$



# Operadores

## Operadores relacionales

Operador	Significado	Ejemplo
==	igualdad	a == b
!=	distinto	a != b
<	menor que	a < b
>	mayor que	a > b
<=	menor o igual que	a <= b
>=	mayor o igual que	a >= b

## Operadores especiales

Operador	Significado	Ejemplo
++	incremento	a++ (postincremento) ++a (preincremento)
--	decremento	a-- (postdecremento) --a (predecremento)
(tipo)expr	cast	a = (int) b
+	concatenación de cadenas	a = "cad1" + "cad2"
.	acceso a variables y métodos	a = obj.var1
()	agrupación de expresiones	a = (a + b) * c

# Precedencia de los operadores

Todos los operadores binarios que tienen la misma prioridad (excepto los operadores de asignación) son evaluados de izquierda a derecha. **Los operadores de asignación son evaluados de derecha a izquierda.**

Operador	Notas
. [] ()	Los corchetes se utilizan para los arreglos
++ -- ! ~	! es el NOT lógico y ~ es el complemento de bits
new (tipo)expr	new se utiliza para crear instancias de clases
* / %	Multiplicativos
+ -	Aditivos
<< >> >>>	Corrimiento de bits
< > <= >=	Relacionales
== !=	Igualdad

Operador	Notas
&	AND (entre bits)
^	OR exclusivo (entre bits)
	OR inclusivo (entre bits)
&&	AND lógico
	OR lógico
? :	Condicional
= += -= *= /= %= &= ^=  = <<= >>= >>>=	Asignación

# Operador Ternario

Un operador ternario (?) es una estructura empleada para reemplazar el uso del condicional *if*, haciendo más corta su escritura y comparación de valores, además permite retornar un valor dependiendo de la condición planteada.

Ejemplos:

```
int varInt1 = 10;  
int varInt2 = 15;
```

```
varInt1 > varInt2 ? "El numero mayor es: " + varInt1 : "El numero mayor es: " + varInt2
```

```
varInt1 > varInt2 ? varInt1 + 10 : varInt2 + 20
```

Finalmente podemos darnos cuenta la facilidad con la que podemos emplear los operadores ternarios en Java; y que su uso brinda una forma diferente de emplear condicionales.

Almuerzo

**45** min



# Operaciones y **Estructuras de Control**



# Sentencias

---

## Sentencia de Expresión y declaración de variables:

Los siguientes tipos de expresiones pueden ser hechas dentro de una sentencia terminando la expresión con punto y coma (;):

1. Expresiones de declaración.
2. Expresiones de asignación.
3. Cualquier uso de los operadores ++ y --.
4. Llamada de métodos.
5. Expresiones de creación de objetos.

Esta clase de sentencias son llamadas sentencias de expresión.

# Sentencias

---

## Sentencias de control de flujo:

Estas determinan el orden en el cual serán ejecutadas otro grupo de sentencias. Las sentencias ***if*** y ***for*** son ejemplos de sentencias de control de flujo.

## Bloque de sentencias (scope):

Un bloque es un grupo de cero o más sentencias encerradas entre llaves ( ***{*** y ***}*** ). Se puede poner un bloque de sentencias en cualquier lugar en donde se pueda poner una sentencia individual.

```
boolean condicion = true
```

```
if ( condicion ){  
    condicion = false;  
    int i = 1;  
    i++;  
}
```

# Sentencias de control de flujo

## La sentencia *if*:

La sentencia *if* permite llevar a cabo la ejecución condicional de sentencias.

```
if ( Expresion ){  
    sentencias;  
}
```

Se ejecutan las sentencias si al evaluar la expresión se obtiene un valor booleano *true*.

```
if ( Expresion ){  
    sentenciasA;  
} else {  
    sentenciasB;  
}
```

*Sentencia if anidada*, si al evaluar la expresión se obtiene un valor booleano *true* se ejecutarán las **sentenciasA**, en caso contrario se ejecutarán las **sentenciasB**.

## La sentencia *switch*:

Cuando se requiere comparar una variable con una serie de valores diferentes, puede utilizarse la sentencia *switch*, en la que se indican los posibles valores que puede tomar la variable y las sentencias que se tienen que ejecutar si es que la variable coincide con alguno de dichos valores.

```
switch( variable ){  
    case valor1:  
        sentencias;  
        break;  
    case valor2:  
        sentencias;  
        break;  
    ...  
  
    case valorN:  
        sentencias;  
        break;  
    default:  
        sentencias;  
}
```



## El ciclo *for*:

El ciclo *for* repite una sentencia, o un bloque de sentencias, mientras una condición se cumpla. Se utiliza la mayoría de las veces cuando se desea repetir una sentencia un determinado número de veces.

La forma general de la sentencia *for* es la siguiente:

```
for (inicialización ; condición ; incremento){  
    sentencias;  
}
```

1. En su forma más simple, la inicialización es una sentencia de asignación que se utiliza para establecer una variable que controle el ciclo.
2. La condición es una expresión que comprueba la variable que controla el ciclo y determina cuándo salir del ciclo.
3. El incremento define la manera en cómo cambia la variable que controla el ciclo.

## Los ciclos *while* y *do-while*:

Al igual que los ciclos *for* repiten la ejecución de un bloque de sentencias mientras se cumpla una condición específica.

### 1. La sentencia *while*

Se aplica la condición “*mientras que*” para iterar en el ciclo. La sentencia *while* es la siguiente:

```
while (condición){  
    sentencias;  
}
```

### 2. La sentencia *do-while*

Se aplica la condición “*hasta que*”, donde la sentencia indica el fin del ciclo. Y al contrario de los ciclos *for* y *while* que comprueban una condición en lo alto del ciclo, el ciclo *do-while* la examina en la parte más baja del mismo. Esta característica provoca que un ciclo *do-while* siempre se ejecute por lo menos una vez.

```
do {  
    sentencias;  
} while (condición)
```

# Condicionadores de Salida de Ciclo

---

## **break:**

Esta sentencia tiene dos usos. El primer uso es terminar un case en la sentencia *switch*. El segundo es forzar la terminación inmediata de un ciclo, saltando la prueba condicional normal del ciclo.

## **continue:**

Es similar a la sentencia **break**. Sin embargo, en vez de forzar la terminación del ciclo, *continue* fuerza la siguiente iteración y salta cualquier código entre medias.

## **return:**

Se utiliza para provocar la salida del método actual; es decir, **return** provocará que el programa vuelva al código que llamó al método.

La sentencia return puede regresar o no, un valor. Para devolver un valor, se pone el valor después de la palabra clave **return**.

```
return valor;
```

# Modificadores de acceso

Los modificadores de acceso definen el alcance de uso o visibilidad de una clase, método o variable a través de otros sectores del código.

- 1. **public**: nos indica que es accesible desde cualquier clase [interface].
- 2. **private**: sólo es accesible desde la clase actual.
- 3. **protected**: accesible desde la clase actual, sus descendientes o el paquete del que forma parte.
- 4. **sin ninguna palabra**: accesible desde cualquier clase del paquete.

MODIFICADOR	CLASE	PACKAGE	SUBCLASE	OTROS
public	SÍ	SÍ	SÍ	SÍ
protected	SÍ	SÍ	SÍ	No
No especificado	SÍ	SÍ	No	No
private	SÍ	No	No	No



# Funciones

---

Una función o **método** en Java es un fragmento de código, unas cuantas líneas de programación, que resuelven un problema único y muy puntual, que se puede reutilizar cuantas veces sea necesario a lo largo de un proyecto o programa de mayor tamaño.

## Estructura de una función en Java:

1. **Acceso:** El acceso puede ser *public* o *private*, es decir, podrá ser invocado solo dentro de la clase donde se encuentra o también en todo el programa.
2. **Modificador:** El modificador será *final* y/o *static*.
3. **Tipo:** El tipo se refiere al tipo de dato que va a retornar o devolver al programa esa función. El nombre debe asignarse haciendo referencia a la tarea que hace esa función, esto facilita la lectura y comprensión del código.
4. **Argumentos:** En cuanto a los argumentos deben tener su tipo y estar separados por comas, son los datos que necesita la función para ejecutarse.
5. **Return:** La palabra `return` marca el final de las instrucciones de una función.

# Funciones

## Estructura de una función en Java:

```
[acceso][modificador] tipo nombre(tipo nombre argumento1, tipo nombre argumento2) {  
    instrucciones  
    return;  
}
```

## Ejemplo:

```
// construimos una función de dominio público, estática y de tipo entero  
// esta función tiene 2 parámetros de tipo entero  
public static int sumar(int num1, int num2) {  
    // en la variable interna 'sumar' guardamos el resultado  
    int sumar = num1 + num2;  
    //return devuelve el resultado de esta función  
    return sumar;  
}
```

Receso

**15** min



# Programación **Orientada a Objetos**



# Programación Orientada a Objetos

---

La programación Orientada a Objetos nace de los problemas creados por la programación estructurada y nos ayuda a resolver ciertos problemas como:

1. Código muy largo: A medida que un sistema va creciendo y se hace más robusto el código generado se vuelve muy extenso haciéndose difícil de leer, depurar, mantener.
2. Si algo falla, todo se rompe: Ya que con la programación estructurada el código se ejecuta secuencialmente al momento de que una de esas líneas fallará todo lo demás deja de funcionar.
3. Difícil de mantener.

# Programación Orientada a Objetos

---

## Paradigma :

Es una teoría que suministra la base y modelo para resolver problemas.

## Paradigma de Programación Orientada a Objetos:

En este modelo de paradigma se construyen modelos de objetos que representan elementos (objetos) del problema a resolver, que tienen características y funciones.

Permite separar los diferentes componentes de un programa, simplificando así su creación, depuración y posteriores mejoras

# Programación Orientada a Objetos

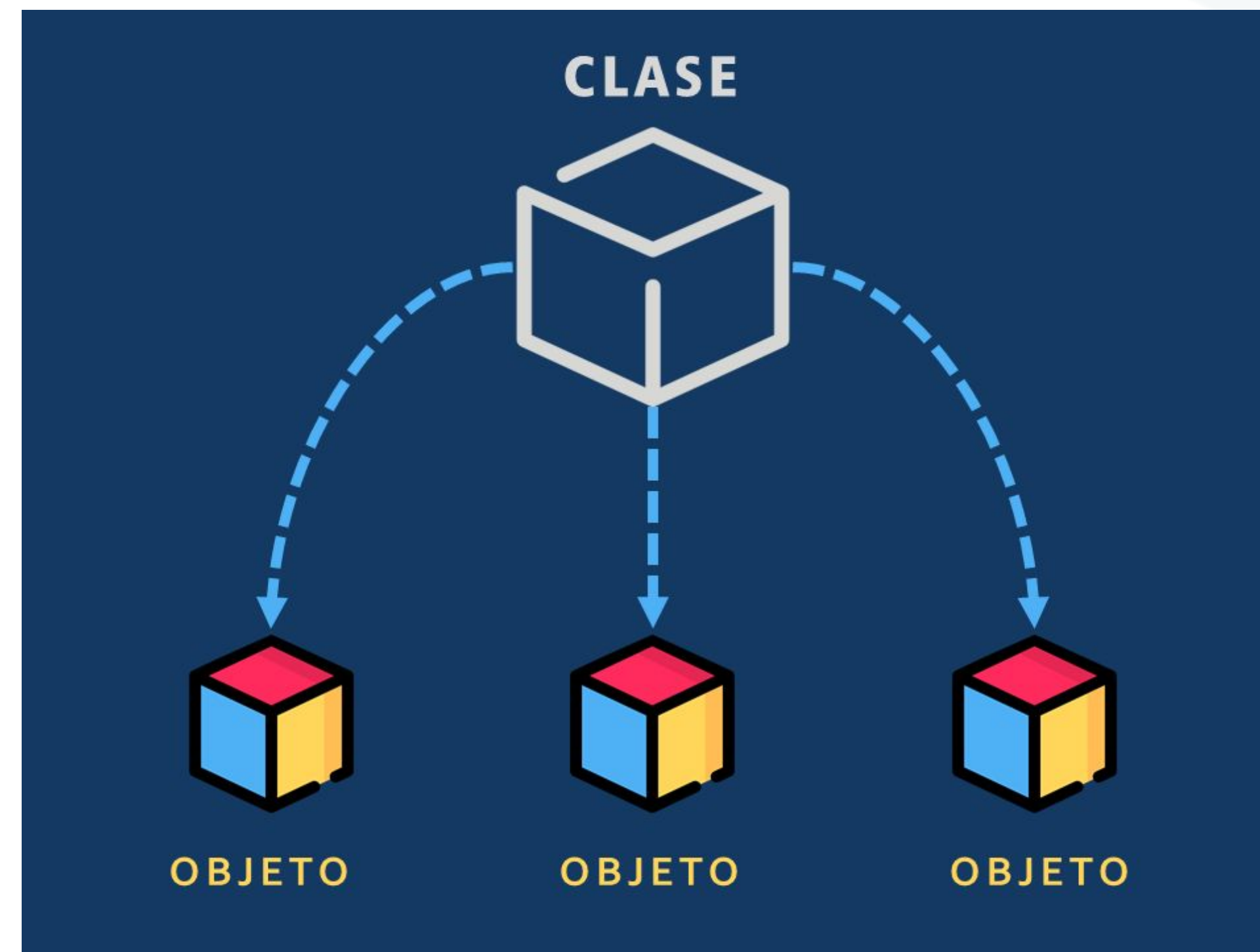
La programación orientada a objetos se sirve de diferentes conceptos, como: *Abstracción de datos, Encapsulación, Eventos, Modularidad, Herencia y Polimorfismo*. El cual podemos definir de la siguiente manera.

4 elementos:

1. **Clases**
2. **Propiedades**
3. **Métodos**
4. **Objetos**

Y 4 Pilares:

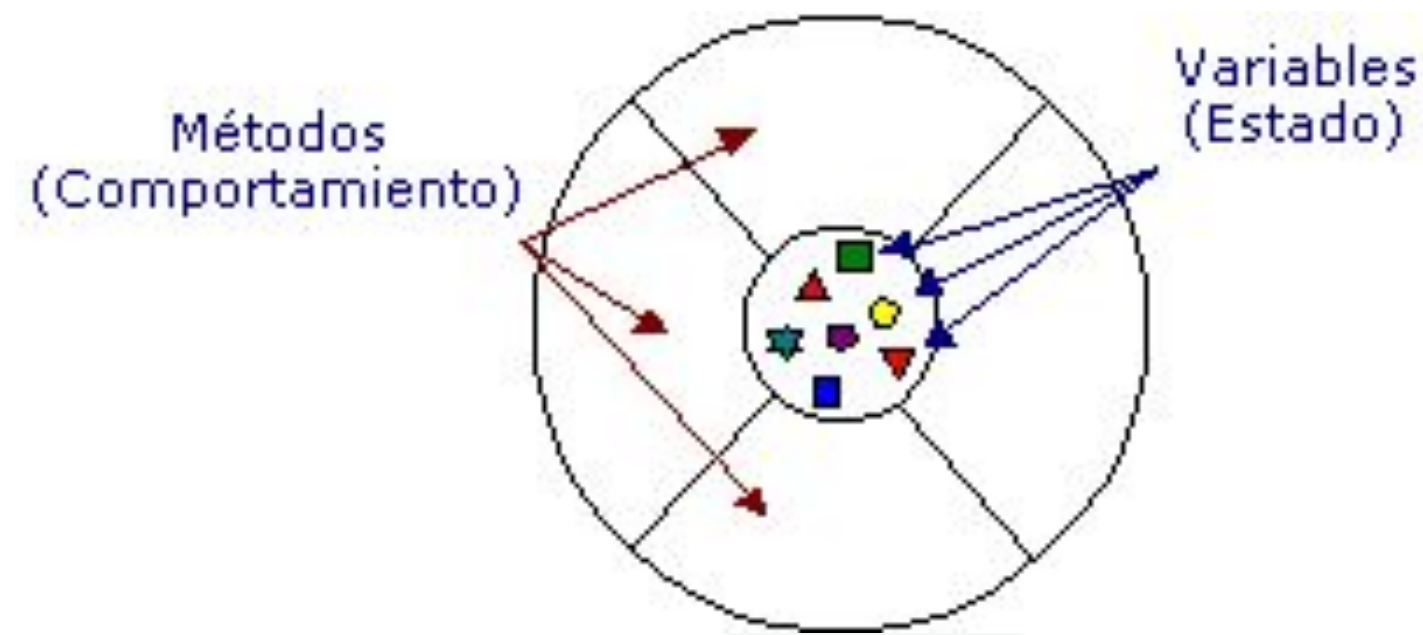
1. **Encapsulamiento**
2. **Abstracción**
3. **Herencia**
4. **Polimorfismo**



# Clases y objetos

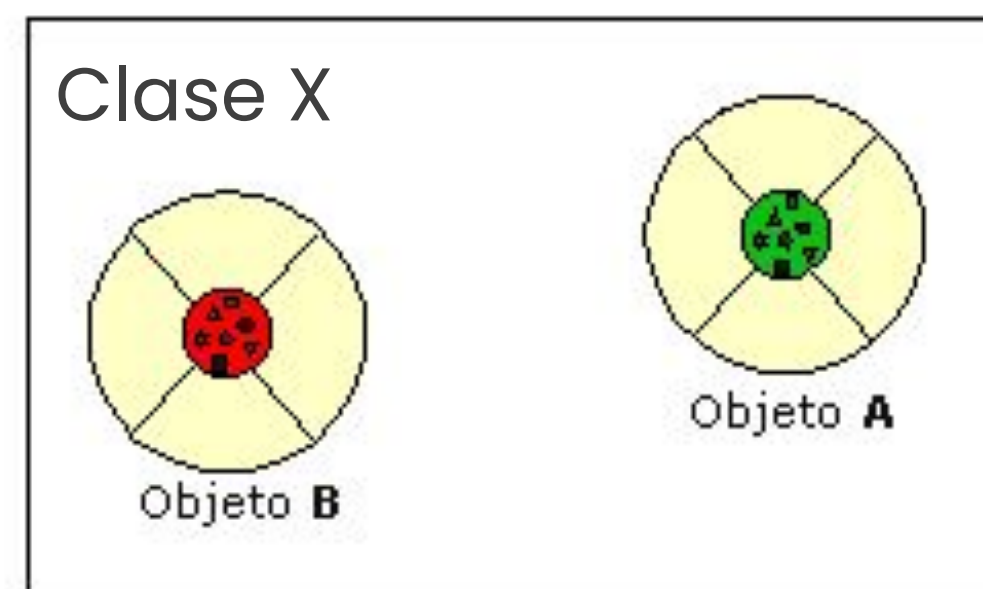
## Objeto:

Un objeto es una encapsulación genérica de datos y de los procedimientos para manipularlos.



## Clase:

Una clase está formada por los métodos y las variables que definen las características comunes a todos los objetos de esa clase. Precisamente la clave de la OOP está en abstraer los métodos y los datos comunes a un conjunto de objetos y almacenarlos en una clase.





# Objetos

---

## Estados o propiedades (atributos):

Son las características de nuestros objetos. Estas propiedades siempre serán sustantivos y pueden tener diferentes valores que harán referencia a nombres, tamaños, formas y estados.

Por ejemplo: el color de tus ojos es verde o azul (color es el atributo, verde y azul son posibles valores para este atributo).

## Los Comportamientos o métodos:

Son todas las operaciones de nuestros objetos que solemos llamar usando verbos o sustantivos y verbos. Por ejemplo: los métodos del objeto ojo pueden ser, abrirPestana(), cerrarPestana(), llorar(), entre otras.

# Programación **orientada a objetos**



Es el proceso de seleccionar conjuntos de datos importantes para un Objeto en su software y omitir los insignificantes.

Permite que se puedan definir nuevas clases basadas de unas ya existentes a fin de reutilizar el código.

Es el proceso de almacenar en una misma sección los elementos de una abstracción que constituyen su estructura y su comportamiento.

Es la capacidad que tienen los objetos de una clase en ofrecer respuesta distinta e independiente en función de los parámetros utilizados durante su invocación

# Abstracción y Clases

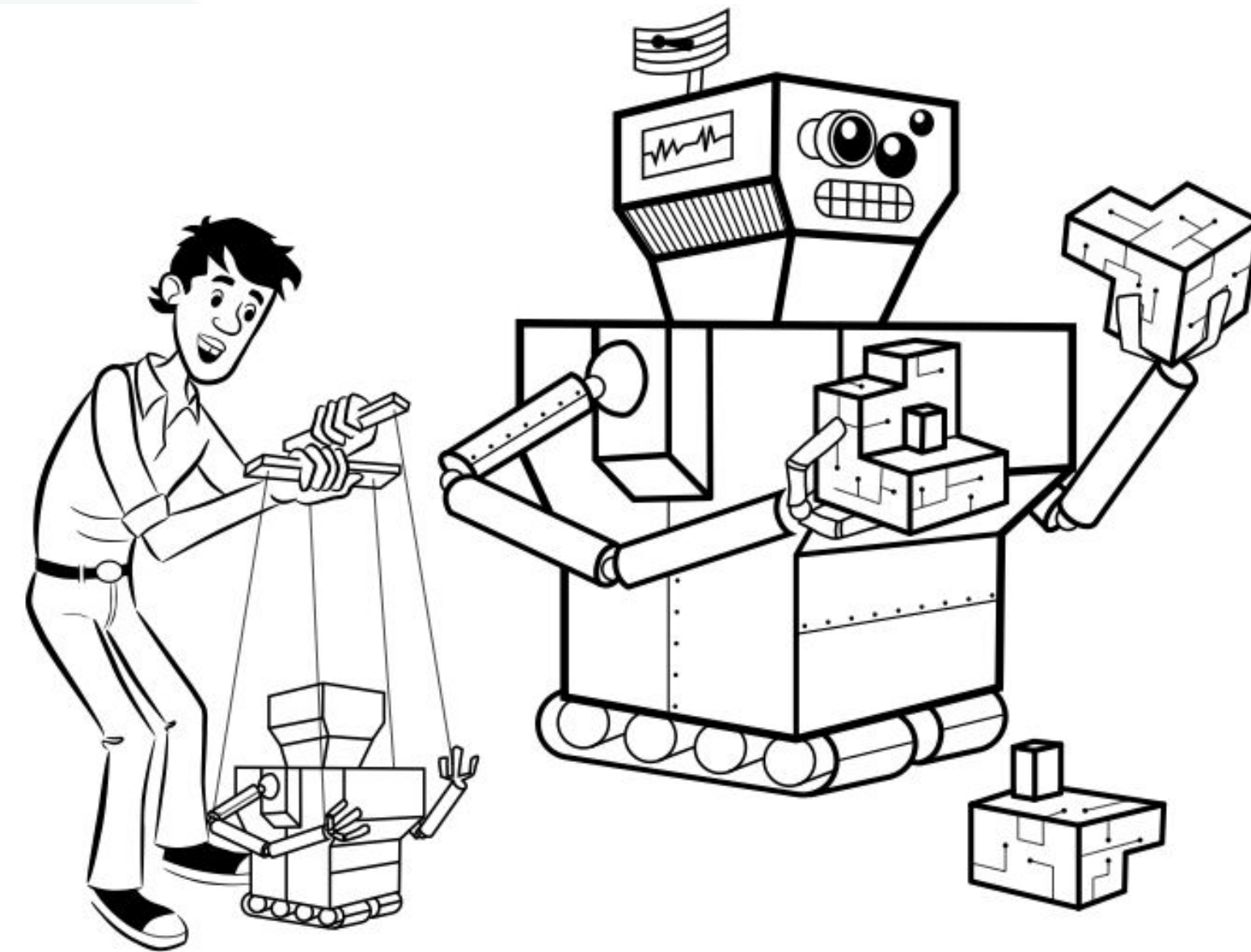
## Abstracción:

Se trata de analizar objetos de forma independiente, sus propiedades, características y comportamientos, para abstraer su composición y generar un modelo, lo que traducimos a código como clases, por tanto, cada clase debe tener identidad (con un nombre de clase único), estado (con sus atributos) y comportamiento (con sus métodos y operaciones), por ejemplo:

**Nombre de la clase:** *Persona*.

**Atributos:** *edad, altura, ancho, peso, nombre.*

**Operaciones:** *caminar(), comer(), respirar().*



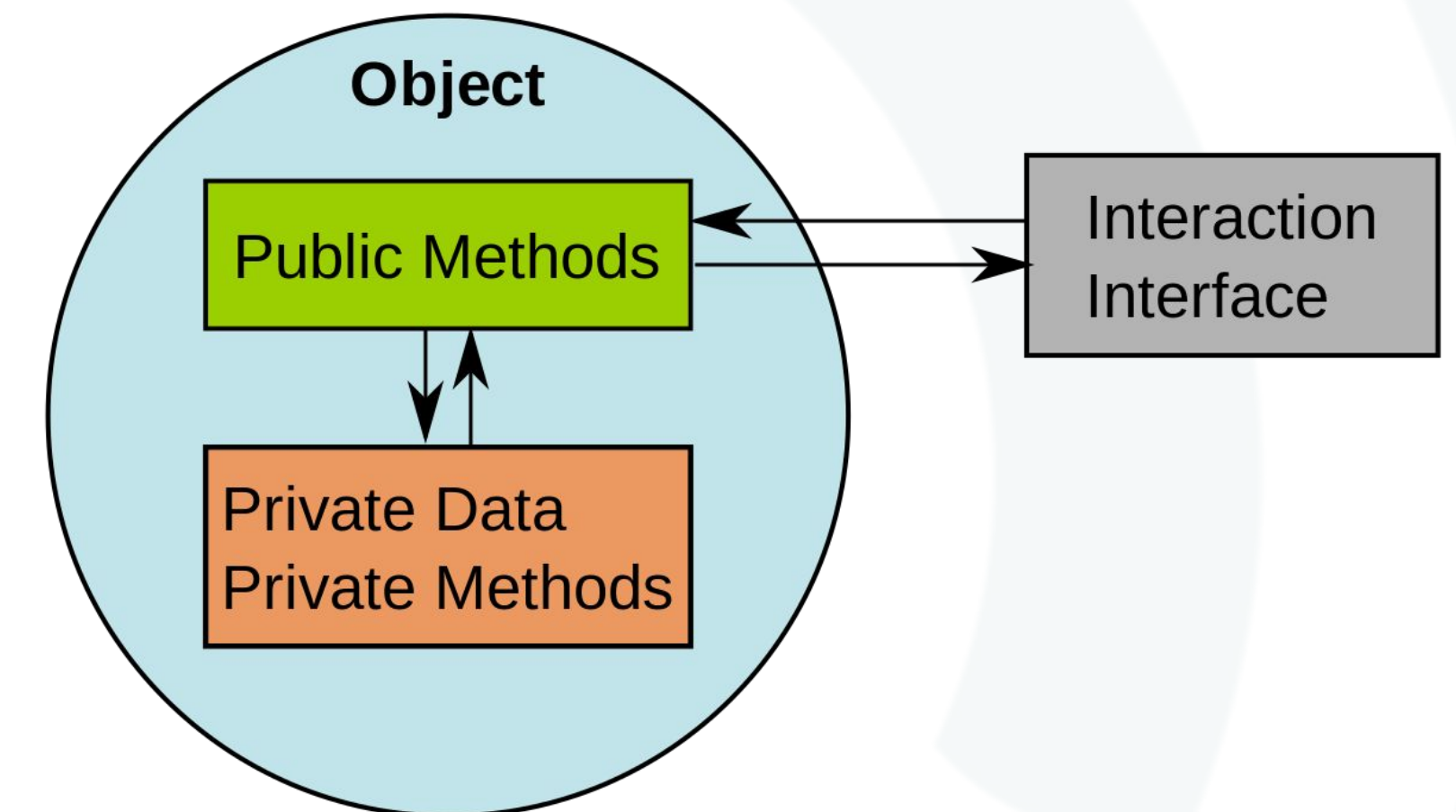


# Encapsulación

El empaquetamiento o agrupación de las variables de un objeto externo con la protección de sus métodos se le llama encapsulamiento. Los métodos rodean y esconden el núcleo del objeto de otros objetos en el programa.

Los **Modificadores de Acceso** nos ayudan a limitar desde dónde podemos leer o modificar atributos especiales de nuestras clases. Podemos definir qué variables se pueden leer/editar por fuera de las clases donde fueron creadas.

El encapsulamiento consiste en agrupar en una Clase las **características (atributos)** con un acceso privado y los **comportamientos (métodos)** con un acceso público.

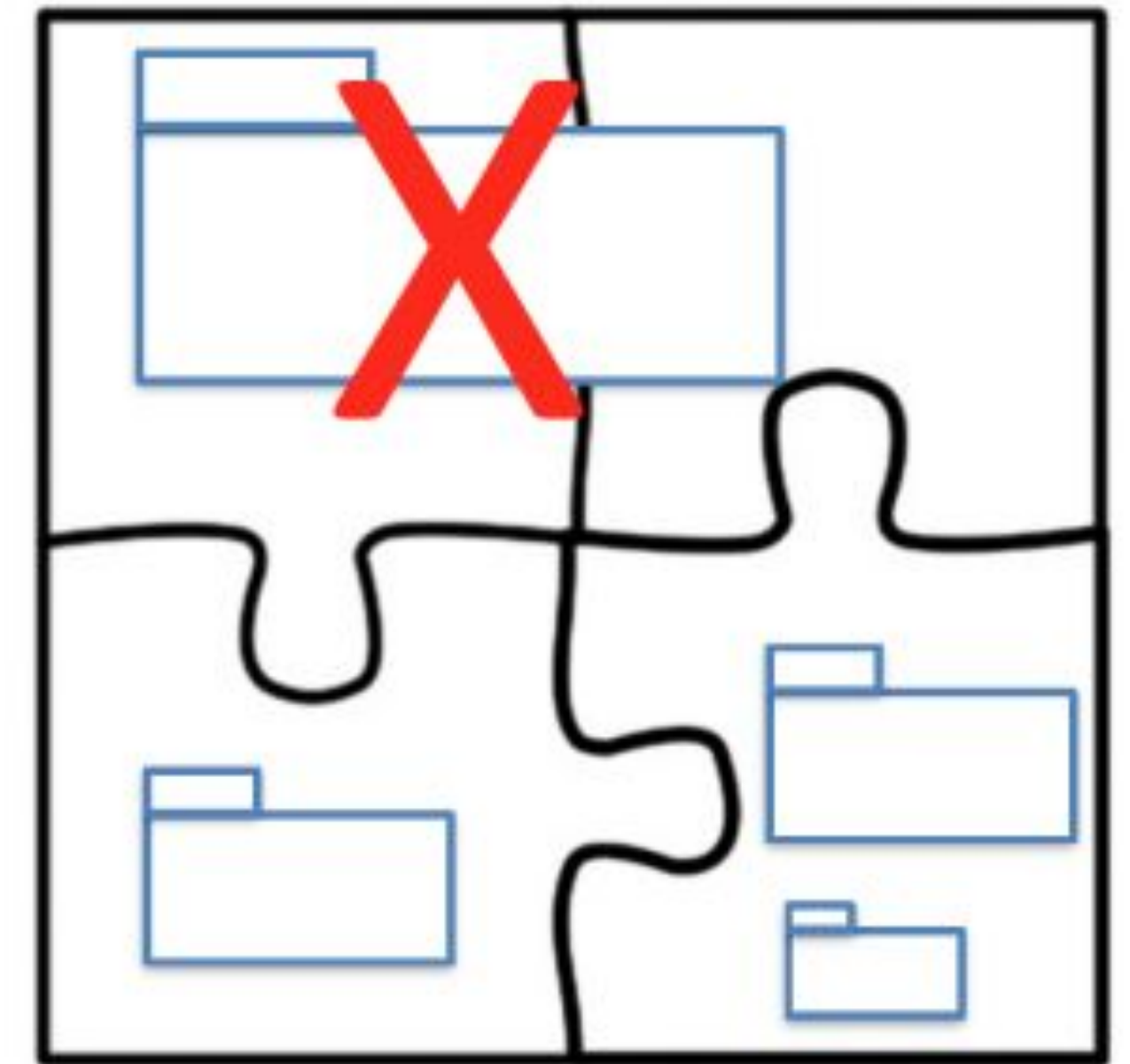




# Encapsulación

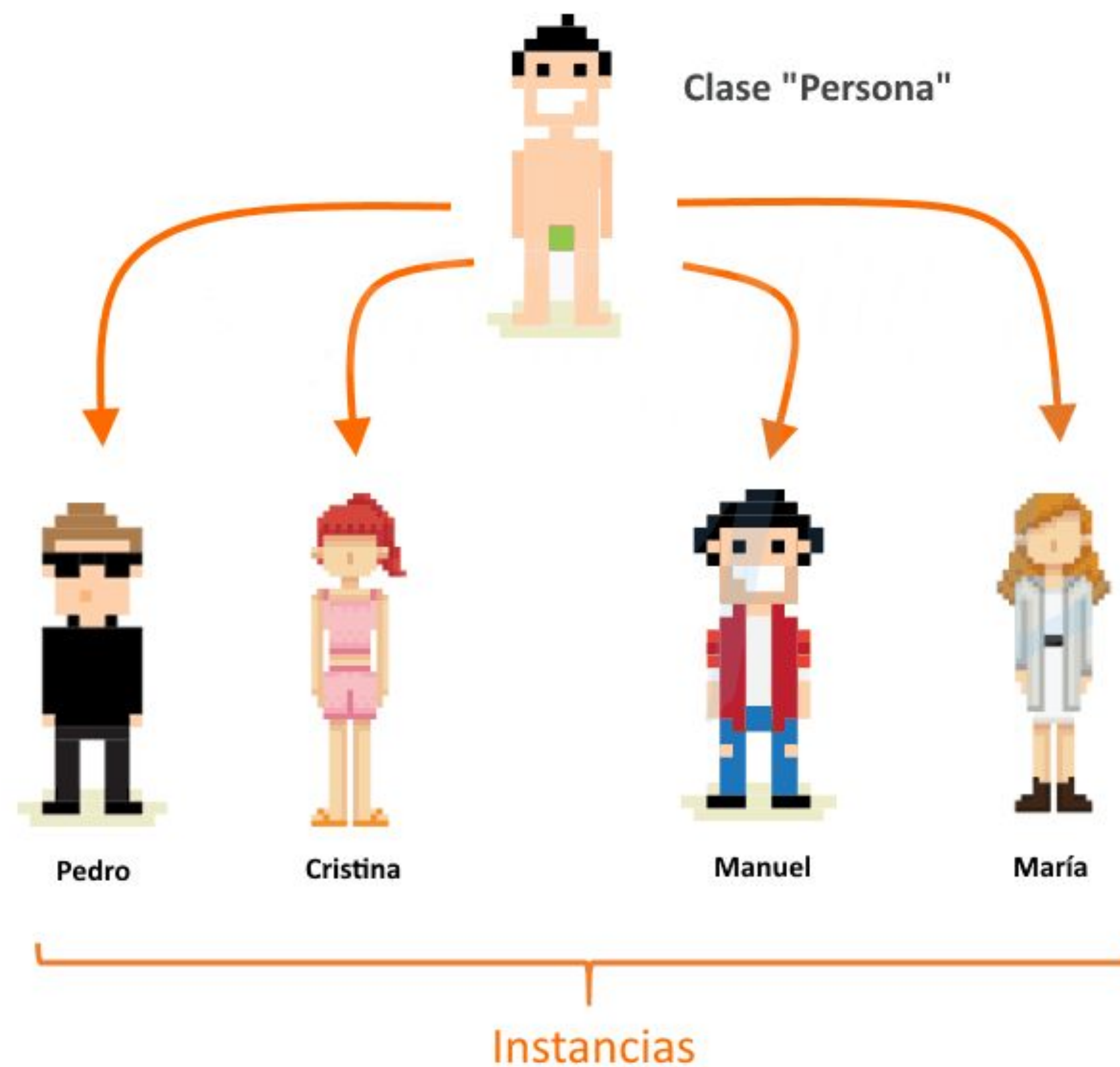
El encapsulamiento de variables y métodos en un componente de software ordenado es, todavía, una simple idea poderosa que provee dos principales beneficios a los desarrolladores de software:

1. **Modularidad**, esto es, el código fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos. Así mismo, un objeto puede ser transferido alrededor del sistema sin alterar su estado y conducta.
2. **Ocultamiento de la información**, es decir, un objeto tiene una "interfaz pública" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier tiempo sin afectar a los otros objetos que dependan de ello.



# Herencia

Es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.



Por tanto una herencia es:

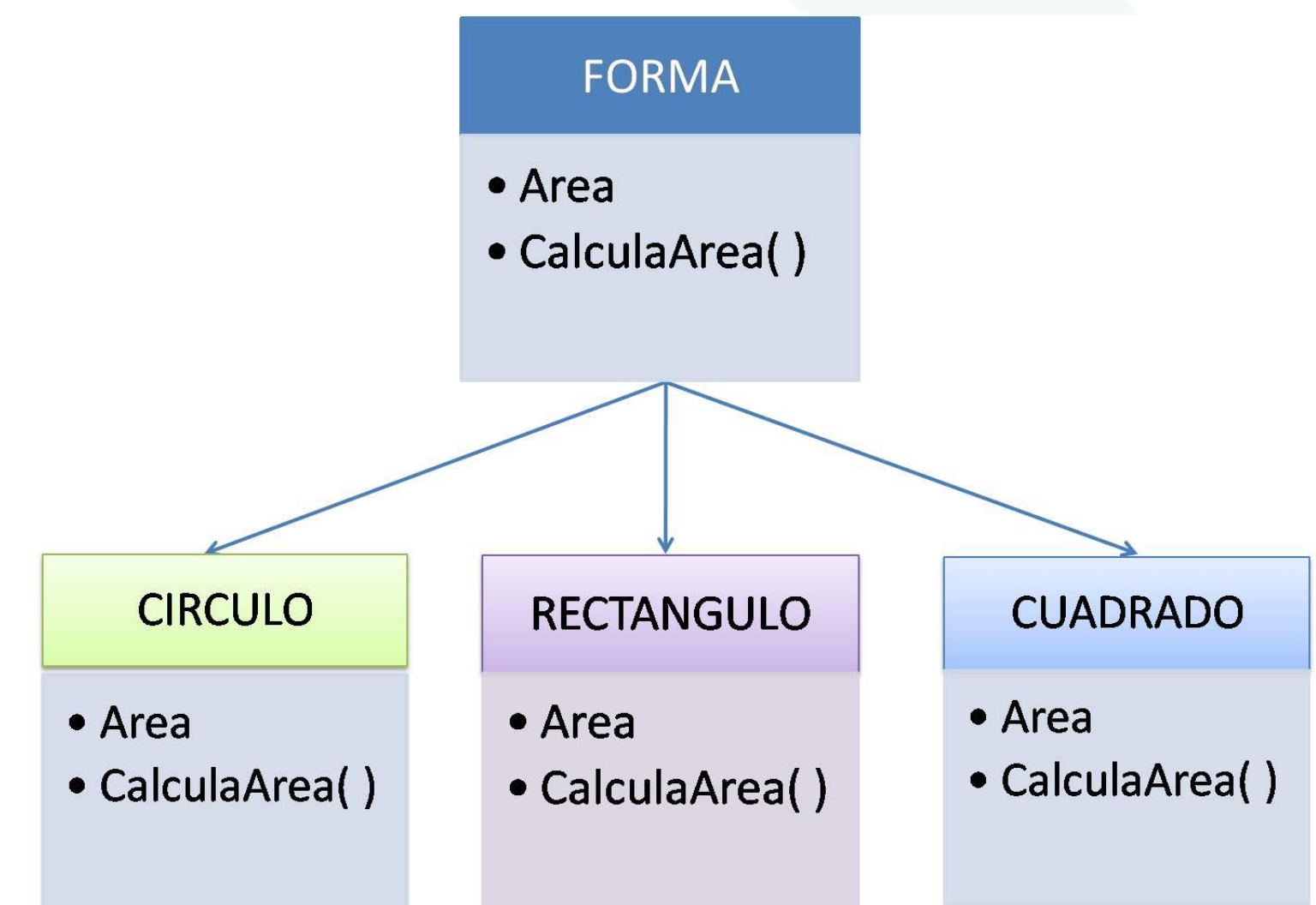
1. Es la relación entre una clase general y otra clase más específica.
2. Es un mecanismo que nos permite crear clases derivadas a partir de clases base.
3. Nos permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

# Polimorfismo

Consiste en sobrescribir algunos métodos de la clase de la cual heredan nuestras subclases para asignar comportamientos diferentes.

No podemos sobrescribir los métodos marcados como **final** o **static**.

La sobreescritura de constructores consiste en usar los miembros heredados de una **superclase** pero con argumentos diferentes.





# Clases Abstractas

---

Este tipo de clases nos permiten crear "*métodos generales*", que recrean un comportamiento común, pero sin especificar cómo lo hacen, por tanto, declara la existencia de métodos pero no la implementación de dichos métodos (o sea, las llaves { } y las sentencias entre ellas), se considera una clase abstracta.

A nivel de código tiene por particularidad que algunos de sus métodos no tienen "cuerpo de declaración".

Una clase abstracta puede contener métodos no-abstractos pero al menos uno de los métodos debe ser declarado abstracto.

```
abstract class Drawing
{
    abstract void miMetodo(int var1, int var2);
    String miOtroMetodo( ){ ... }
}
```

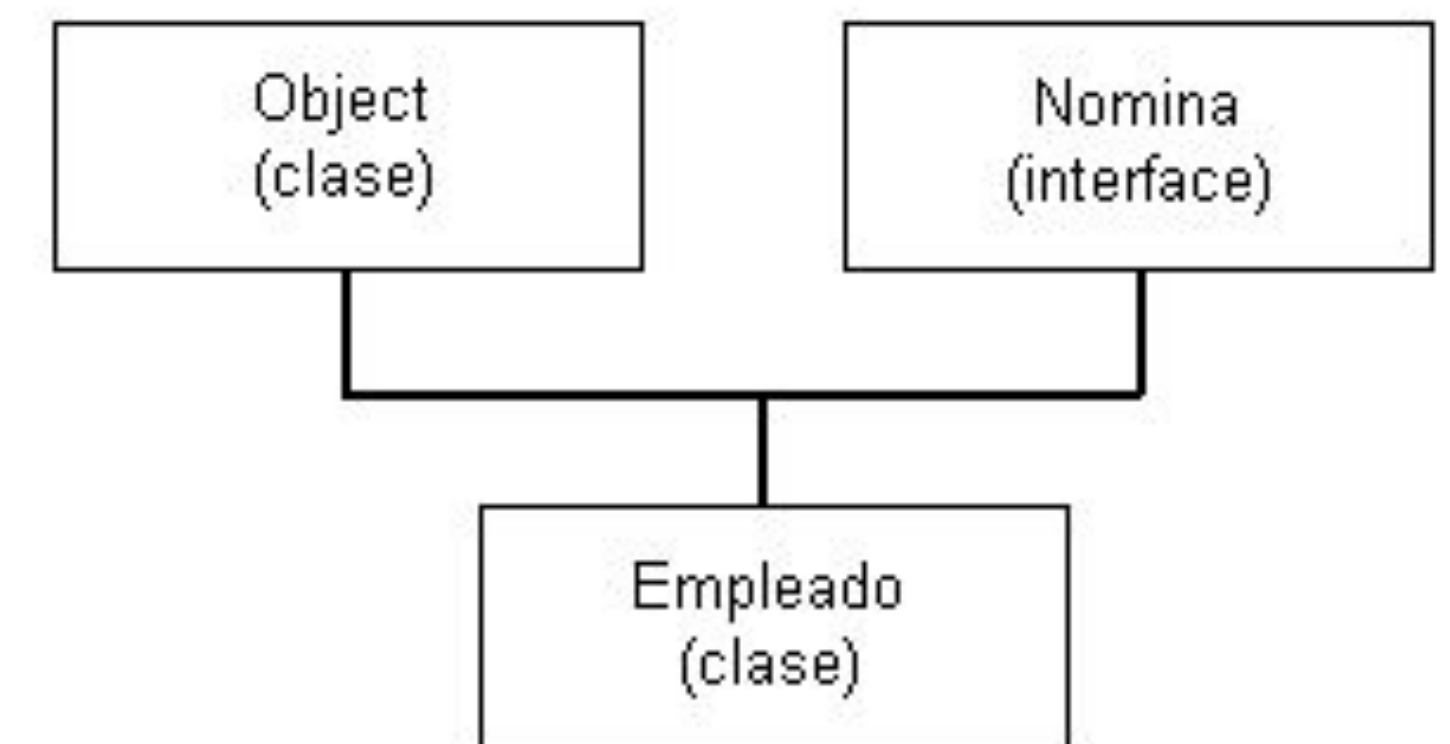
# Interfaz

Es un conjunto de métodos abstractos y de constantes cuya funcionalidad es la de determinar el funcionamiento de una clase, es decir, funciona como un molde o como una plantilla.

Las interfaces establecen la forma de las clases que la implementan, así como sus nombres de métodos, listas de argumentos y listas de retorno, pero NO sus bloques de código, eso es responsabilidad de cada clase.

Si la interfaz va a tener atributos, éstos deben llevar las palabras reservadas ***static final*** y con un valor inicial ya que funcionan como constantes por lo que, por convención, su nombre va en ***mayúsculas***.

```
interface Nomina {  
    public static final String EMPRESA = "Patito, S. A.";  
    public void detalleDeEmpleado(Nomina obj);  
}
```







# Gracias

¡Nos vemos pronto!