



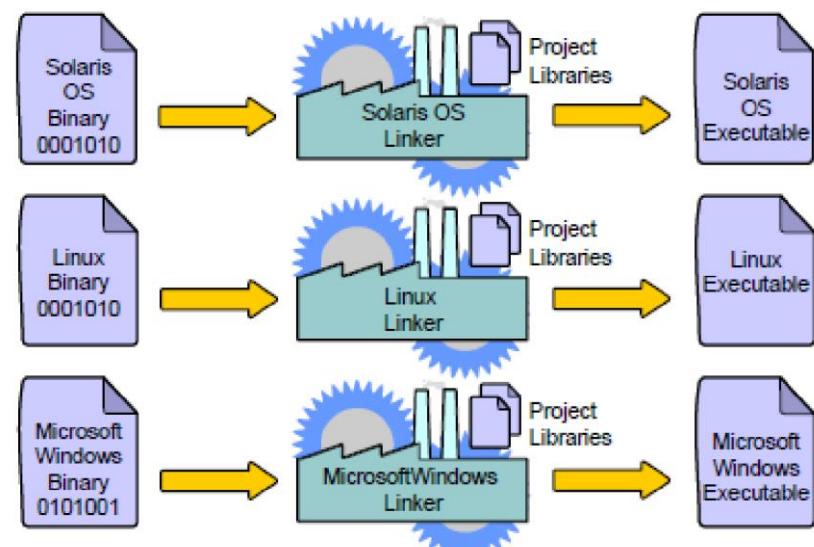
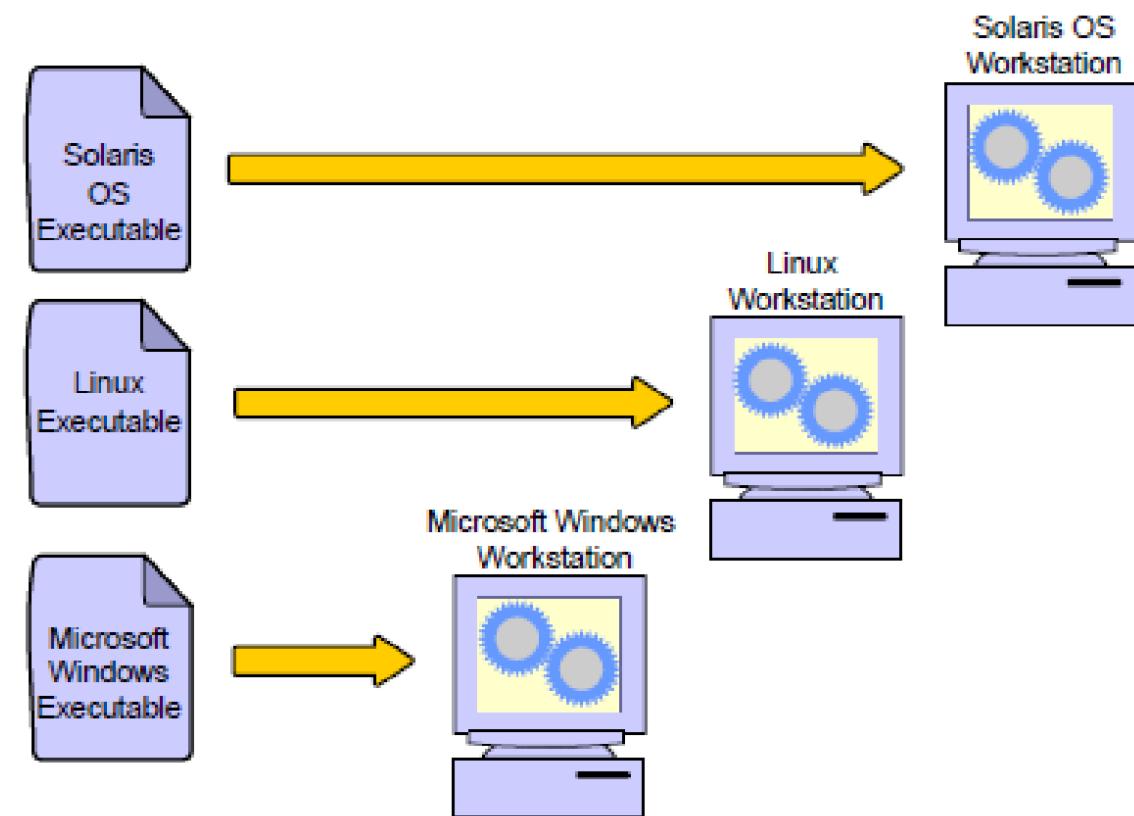
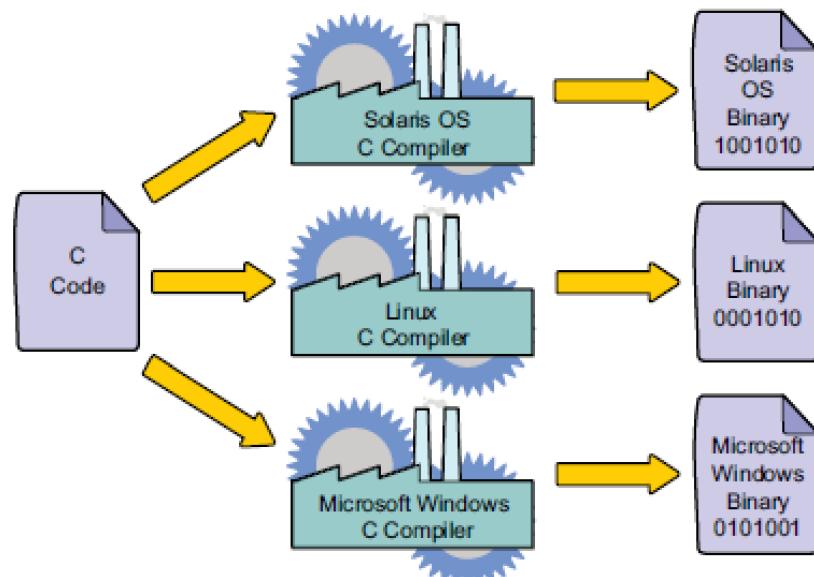
Módulo 1: Iniciación a Java

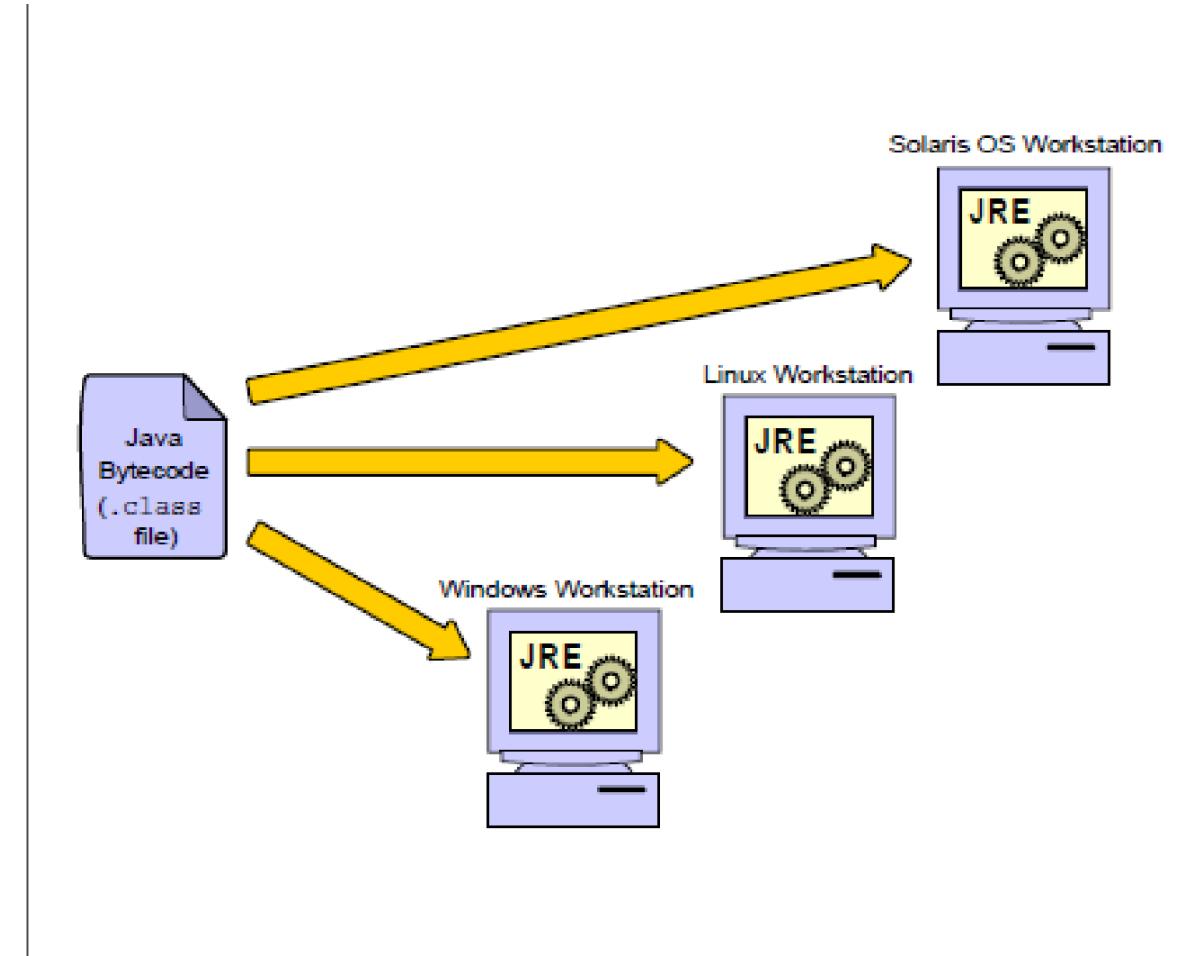
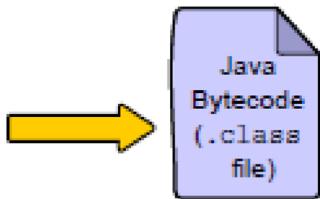
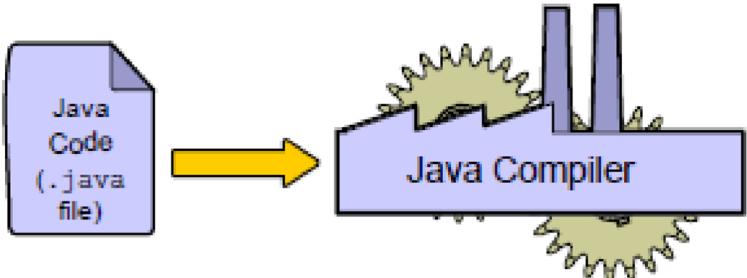
- **Unidad 1:** Introducción al lenguaje Java
- **Unidad 2:** Tipos de datos y operadores
- **Unidad 3:** Sentencias de control
- **Unidad 4:** Vectores y cadenas de texto
- **Unidad 5:** Introducción a la Programación Orientada a Objetos: clases, objetos y métodos
- **Unidad 6:** Herencia
- **Unidad 7:** Uso de interfaces
- **Unidad 8:** Excepciones
- **Unidad 9:** Módulos

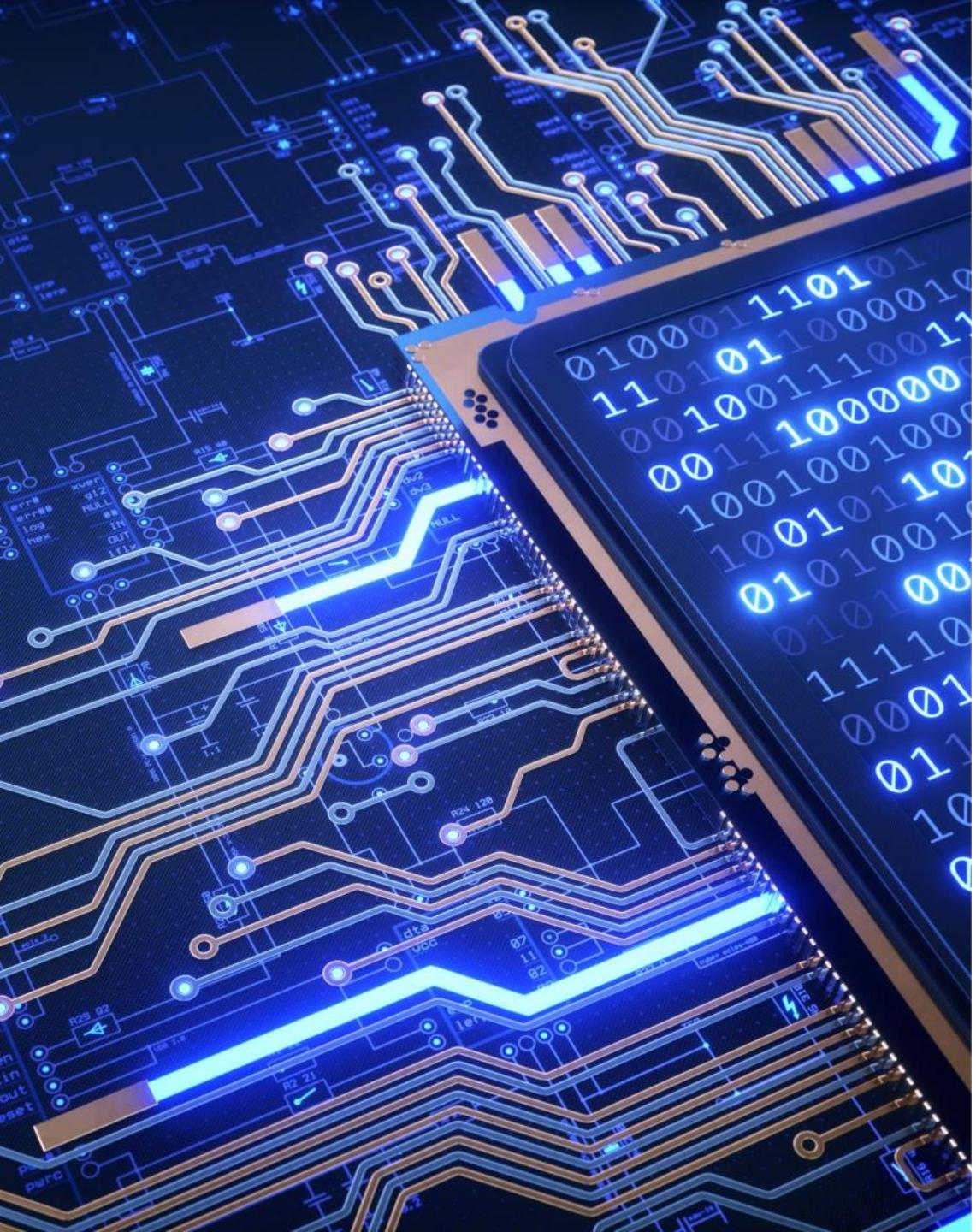
Herramientas

- JDK 11
- IntelliJ IDEA CE







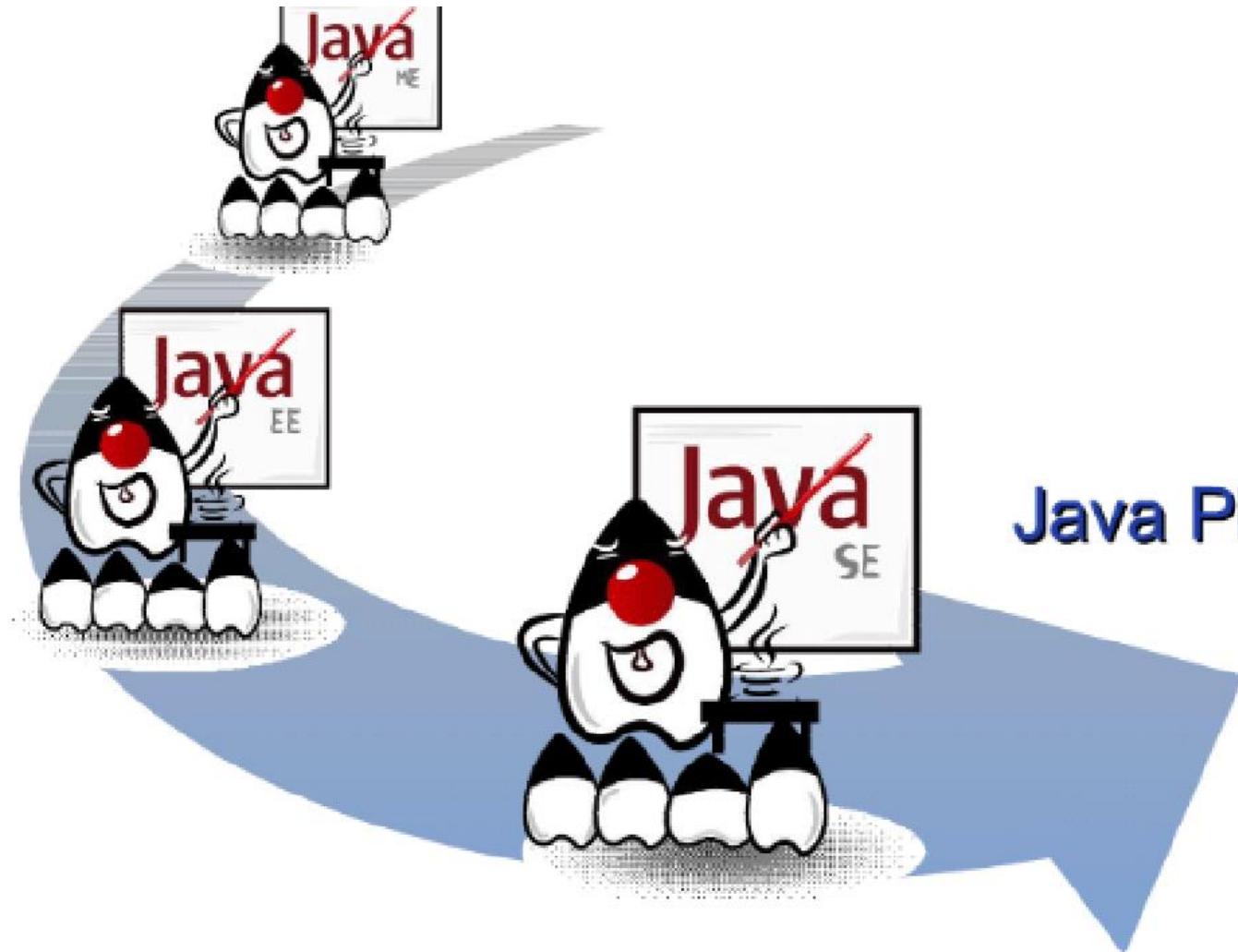


JRE Y JDK

• **JRE (Java Runtime Environment);** Seria necesario unicamente para ejecutar una aplicacion. Esta seria la version que se deben descargar los clientes, usuarios finales de nuestra aplicacion. Incluye unicamente la JVM y un conjunto de librerias para poder ejecutar.

• **JDK (Java Development Kit);** Estos son los recursos que necesitamos los desarrolladores ya que incluye lo siguiente:

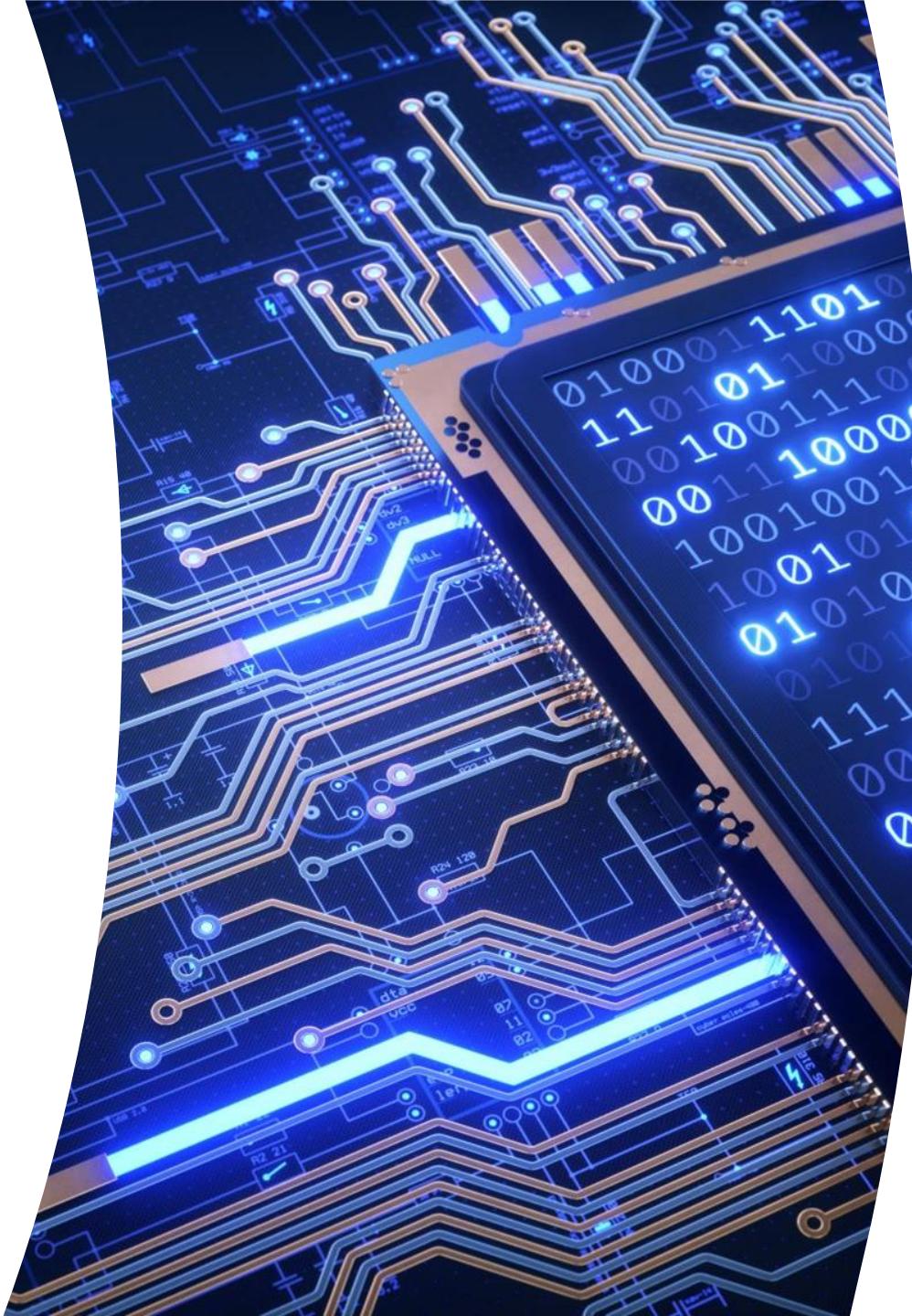
- JRE
- Compilador de java
- Documentacion del API (todas las librerias de Java)
- Otras utilidades por ejemplo para generar archivos .jar, crear documentacion,



Java Platforms



CARACTERÍSTICAS DE JAVA



- Orientado a objetos; como se mencionaba al inicio del capitulo, en Java todo se considera como un objeto. La OO facilita la reutilizacion de codigo como se vera mas adelante.
- Distribuido; Java permite desarrollar aplicaciones distribuidas, esto significa que parte de la aplicacion puede alojarse en un servidor de Madrid y otra parte puede residir en otro servidor de Barcelona por ejemplo.
- Simple; Aunque os parezca mentira en este momento, Java es un lenguaje muy simple de programar. Ademas, contamos con muchas librerias ya desarrolladas, listas para utilizarse y lo mas importante una buena documentacion de uso.
- Multihilo; El lenguaje Java funciona mediante hilos no a traves de procesos como otros lenguajes. Esto hace que sea mas rapida y mas segura su ejecucion.
- Seguro; En el curso iremos viendo las diversas formas de implementar seguridad en aplicaciones Java.
- Independiente de la plataforma; como ya comentamos anteriormente.

```
package com.jorge.holamundo;

/*
 * Primer programa escrito en
Java
*/
public class HolaMundo{
    public static
void main(String
          args[])
    {
        System.out.printl
n("Hola Mundo!");
    }
}
```

Algo de código para empezar

- Los **comentarios** se escriben entre los caracteres /* y */ y pueden ocupar **varias líneas**
- Si un **comentario** sólo ocupa **una línea** se puede escribir justo después de los caracteres //
- La instrucción System.out.println("Hola") escribe el **texto indicado por pantalla**
- Cada **sentencia** de código se **termina** siempre con un ;
- La **definiciones de las clases y métodos** no se consideran sentencias y no terminan con el carácter ; ya que definen nuevos bloques de código
- Cada **bloque de código** se inicia con el carácter { y **termina con }**

Palabras reservadas del lenguaje Java



abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	assert
continue	goto	package	synchronized	enum

Consideraciones

- Normalmente una **aplicación se corresponde con un Proyecto** (en IntelliJ IDEA en nuestro caso)
- Estructura basada en **paquetes** (carpetas) y debemos **especificar uno como mínimo**
- La **unidad mínima de un proyecto** es la **clase** (normalmente pública) y normalmente cada una se escribe en **un fichero de código**
- La **unidad mínima de ejecución** es el **método** que normalmente estará compuesto de un conjunto de instrucciones relacionadas
- Como mínimo tendrá que haber **una clase pública** con el método `public static void main(String args[])`, que será el **punto de arranque** cuando se ejecute el proyecto
- El compilador **nunca procesa los comentarios**
- **Nunca** debemos **editar/modificar** los ficheros del proyecto que **no sean código Java**



Project ▾



Tutorial.java

- >HelloWorld C:\Users\Stepanenko\Desktop\JavaCurso\Ja
 > .idea
- > out
- >HelloWorld
- src
 - >HelloWorld
 - HelloWorld.iml
- > External Libraries
- Scratches and Consoles

```
1 package HelloWorld;  
2  
3 public class Tutorial {  
4     public static void main (String arg []){  
5         System.out.println("Hola mundo");  
6     }  
7 }  
8 }
```

Run: Tutorial

▶ "C:\Program Files\Java\jdk-18.0.1.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA

↑ Hola mundo

🔧



Proceso de creación de una aplicación

- Al compilar, **cada clase de código genera un fichero .class** y sólo una de ellas será la que inicie la aplicación
- No es raro que un proyecto pueda contener **más de 100 clases**, a lo que habría que sumar **otros recursos** (texto, imágenes, . . .)
- Lo normal es que el código compilado se empaquete como **.jar** o **.war**
- Junto con las clases compiladas se empaqueta el **manifiesto de la aplicación**, donde se especifica, entre otras cosas, cuál es la clase que contiene el método main (punto de arranque de la aplicación)
- Los IDE incorporan **menús para facilitar la creación de empaquetados** y manifiestos del proyecto
- Dependiendo de cómo esté configurado el Sistema Operativo **podemos lanzar la aplicación simplemente haciendo doble-click** sobre el empaquetado (HolaMundo.jar, por ejemplo)



DEMO

Tipo	Tamaño	Valor mínimo	Valor máximo
byte	1 byte	-128	127
short	2 bytes	-32768	32767
int	4 bytes	-2147483648	2147483647
long	8 bytes	-9223372036854775808	9223372036854775807
float	4 bytes	-3.402823e38	3.402823e38
double	8 bytes	-1.79769313486232e308	1.79769313486232e308
char	2 bytes		
boolean	1 byte	false	true

Tipos de datos primitivos

Símbolo	Significado	Ejemplo
=	Asignación	$x = 4$
+	Suma	$x + 3$
-	Resta	$x - 3$
/	División	$x / 3$
*	Multiplicación	$x * 4 / 2$
%	Módulo (resto entero)	$20 \% 3$

Operadores de asignación y aritméticos

Símbolo	Significado	Ejemplo
<code>++</code>	Preincremento	<code>++x</code>
<code>++</code>	Posincremento	<code>x++</code>
<code>--</code>	Preditcremento	<code>--x</code>
<code>--</code>	Posdidecremento	<code>x--</code>

Operadores pre/pos incremento

Símbolo	Significado	Ejemplo
<code>+=</code>	Suma y asignación	<code>x += 3</code>
<code>-=</code>	Resta y asignación	<code>x -= 4</code>
<code>*=</code>	Multiplicación y asignación	<code>x *= 10</code>
<code>/=</code>	División y asignación	<code>x /= 10</code>
<code>%=</code>	Módulo y asignación	<code>x %= 2</code>

Operadores combinados

Símbolo	Significado	Ejemplo
<code>==</code>	Igual que	<code>x == 3</code>
<code>!=</code>	Distinto que	<code>x != 3</code>
<code><</code>	Menor que	<code>x < 3</code>
<code><=</code>	Menor o igual que	<code>x <= 4</code>
<code>></code>	Mayor que	<code>x > 3</code>
<code>>=</code>	Mayor o igual que	<code>x >= 6</code>

Operadores de comparación

Símbolo	Significado	Ejemplo	Resultado
!	Negación	<code>!(10 == 10)</code>	false
	Or	<code>(10 == 10) (10 == 3)</code>	true
&&	And	<code>(10 == 10) && (10 == 3)</code>	false

Operadores lógicos

Declaración de variables

- Las variables en Java se deben declarar antes de poderse usar
- En la declaración se debe especificar el tipo y nombre.
Opcionalmente se les puede asignar un valor inicial
- **int cantidad;**
boolean
terminado; **int**
cantidad = 10;
- **boolean** terminado = false;

Uso de valores literales

- Los valores literales son valores fijos que podemos asignar a las variables o bien utilizar para representar directamente por pantalla. En general hay que tener en cuenta lo siguiente:
 - Los números se escriben directamente (cantidad = 3)
 - Los caracteres (char) se escriben siempre entre comillas simples (caracter = 'c')
 - Por defecto un valor numérico literal entero se interpreta como int (cantidad = 3). **Si queremos que se interprete como un long tendremos que añadir el carácter l** (long x = 3l)
 - Por defecto un valor numérico literal decimal se interpreta como double (peso = 10.3). **Si queremos que se interprete como un float tendremos que añadir el carácter f** (float peso = 10.3f)
 - Las cadenas de texto de longitud variable se escriben entre comillas dobles (String nombre = "Tokio School") y el tipo de dato utilizado sería un String (no es un tipo de dato primitivo pero Java nos deja tratarlo como tal)

Declaración de constantes

- Las constantes son variables cuyo **valor no puede ser modificado**. Se declaran siempre con un valor por defecto que debe permanecer inalterado.
- Se declaran con la palabra reservada **final** antes del tipo de dato, y siempre se escriben en mayúsculas.
- Son útiles para representar **valores fijos** a lo largo del programa, y siempre será mejor que escribirlos literalmente en diferentes ubicaciones del código.
- `final int CANTIDAD = 10;`

Pueden formar parte de expresiones junto con otras variables:

- `float precioFinal = precioUnidad * CANTIDAD;`
- Pero nunca podrán ser modificadas:
- ~~`CANTIDAD += 10;`~~

Utilizar cadenas de texto

Consideraciones:

- El tipo String es realmente una **clase**, así que las cadenas de texto son realmente **objetos Java**
- Al tratarse de un objeto podemos realizar **operaciones** sobre la cadena (invocar a sus **métodos**).
- Java nos permite utilizarlo **como si fuera un tipo primitivo** (es un tipo de dato muy utilizado)
- Como es un objeto, su **valor por defecto** es null
- **Cualquier valor** puede ser representado como una cadena, por lo que normalmente es el tipo destino para el input de usuario
- El operador suma (+) aplicado a cadenas de texto **concatena** las cadenas operando para formar una cadena más larga
 - `String nombreApellidos = nombre + " " + apellidos;`

Conversión de tipos

- **Conversión implícita:** La hace automáticamente el compilador siempre que sea posible y no suponga posible pérdida de información

```
float numeroDecimal = 10;
```

- **Conversión explícita:** La hace explícitamente el programador asumiendo cualquier posible pérdida de información

```
float numeroDecimal = 3.4f;  
int cantidad = (int) numeroDecimal;
```

- **Parseo de valores:** No se convierte el tipo (no son compatibles) sino que se extrae el valor para almacenarlo en una variable de otro tipo. Conviene tener en cuenta que si la cadena está vacía se producirá una Excepción (finalización forzada del programa por un fallo)

The blackboard contains several handwritten mathematical equations and diagrams related to calculus, specifically derivatives. At the top left, there is a graph with a curve labeled $y = g(x)$. To its right, there are two lines labeled "Secant Lines". Below these, there is a diagram showing a point x on a vertical axis and a point $x+h$ on a horizontal axis, with a line segment connecting them. To the right of this diagram, there are two derivative definitions:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$
$$= \lim_{h \rightarrow 0} \frac{(x+h)^2 - x^2}{h}$$
$$= \lim_{h \rightarrow 0} \frac{x^2 + 2xh + h^2 - x^2}{h}$$
$$= \lim_{h \rightarrow 0} \frac{2xh + h^2}{h}$$
$$= \lim_{h \rightarrow 0} 2x + h$$

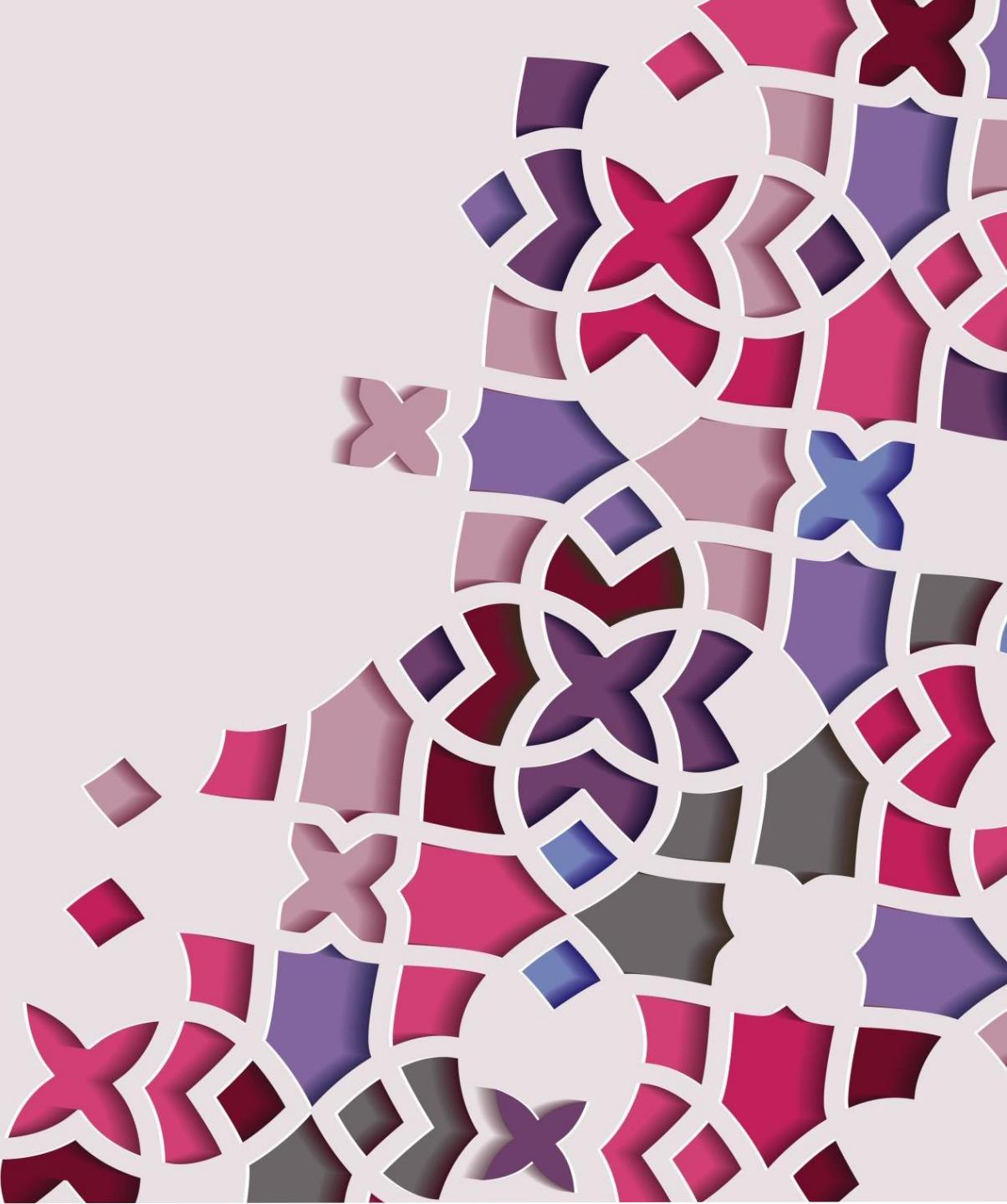
Below these, there are more equations:

$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$
$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h}$$



DEMO

EJERCICIO



Operador condicional

Es un operador ternario que permite asignar un valor u otro a una variable en función a que se cumpla o no una condición. Es una forma reducida de utilizar una sentencia condicional if .. else

La sintaxis del operador es:

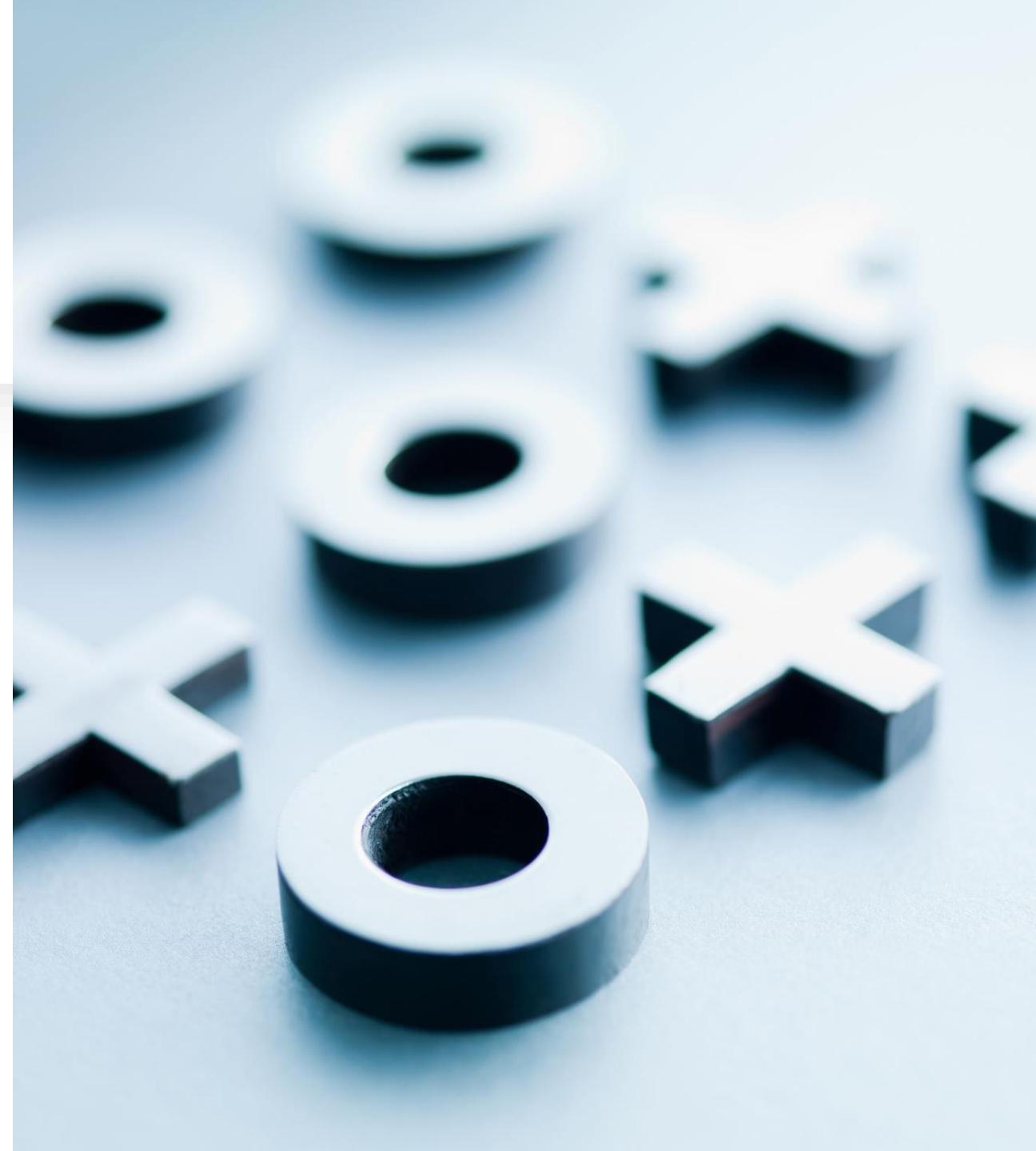
- resultado = condicion ? valorSiVerdadero: valorSiFalso

El siguiente ejemplo realiza la asignación del valor dependiendo de si un número es par o impar en función del resultado de la operación módulo con el número 2

- int numero = 10;
- String cadena = 10 % 2 == 0 ? "Es número par" : "Es número impar";

En el siguiente, se parsea la cadena de forma segura:

- String valor = "30";
- int cantidad = valor.equals("") ? 0 : Integer.parseInt(valor);



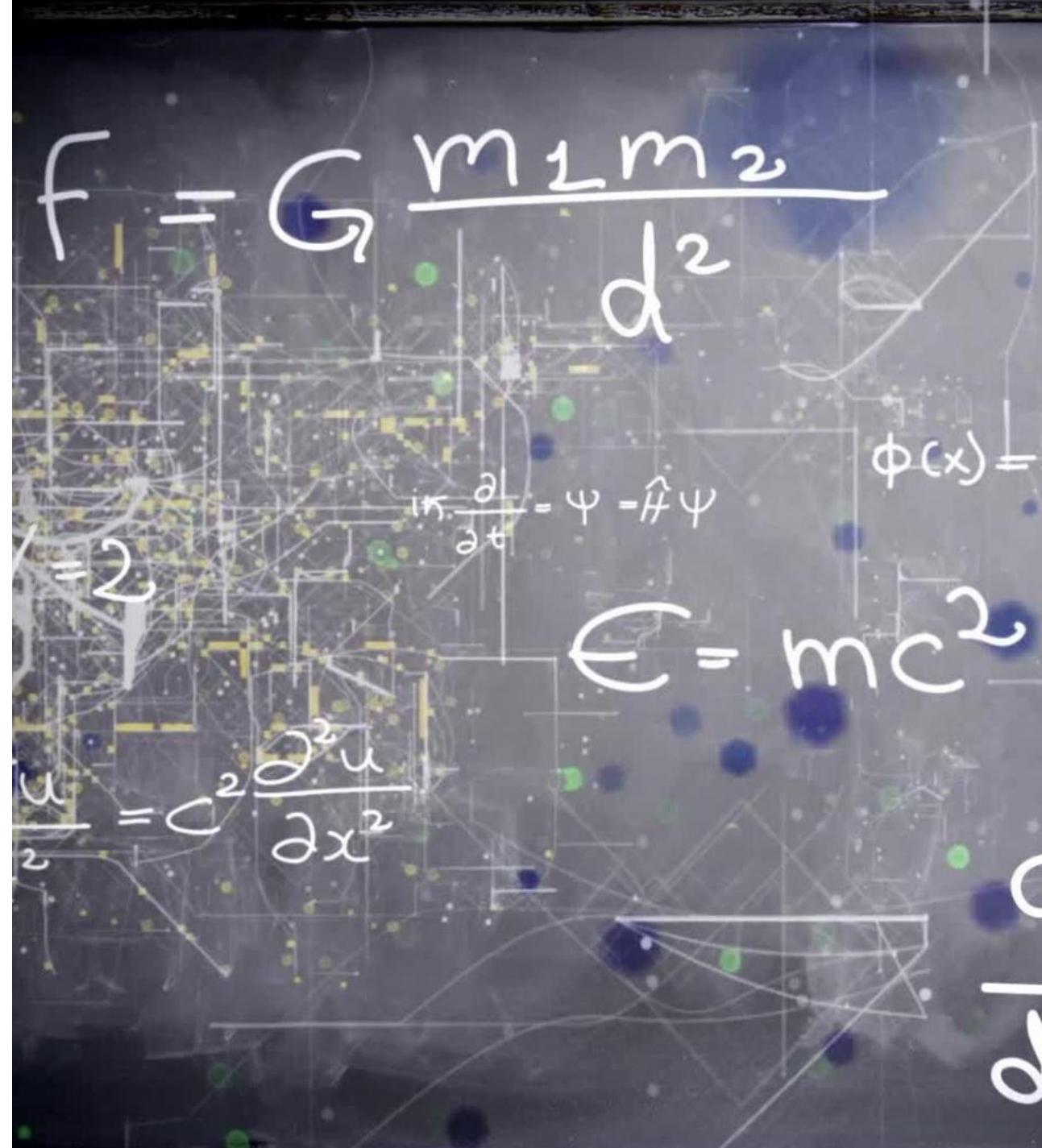
if-else

- El flujo del programa tomará **un camino** u otro **en base a la condición** que antes se cumpla
- Sólo se ejecuta **un camino** (el primero cuya condición se cumpla), incluso aunque varias condiciones puedan cumplirse
- La parte **else if** es **opcional**
- La parte **else** es **opcional**
- Si dentro de la parte **if**, **else if** o **else** sólo se va a escribir una instrucción, se pueden **omitar las llaves**, aunque **no** se recomienda
- Se **recomienda añadir siempre un caso else**, incluso cuando parezca que no es necesario



switch-case

- El flujo del programa tomará **un camino u otro en base a la condición** que antes se cumpla
- Existe la posibilidad de que se ejecuten **varios casos**, siempre y cuando no se rompa la ejecución del primero con la instrucción break;
- El caso **default** es **opcional**
- Se recomienda utilizar para aquellos casos en los que haya más de dos posibilidades.
En caso contrario se recomienda utilizar una sentencia **if-else**
- Se **recomienda añadir siempre un caso default**, incluso cuando no parece que se necesite.



EJERCICIO



while

- Ejecuta el conjunto de instrucciones **mientras se cumpla la condición**
- **La condición debe cumplirse para que el bucle se ejecute al menos una vez**

do-while

- Ejecuta el conjunto de instrucciones **mientras se cumpla la condición**
- Siempre **se ejecuta, al menos, una vez**, se cumpla o no la condición de permanencia

for

- Se ejecuta el conjunto de instrucciones mientras se cumpla la condición (segunda expresión de la definición del for)
- Ideal en los **casos en que sabemos el número de veces** que algo debe ejecutarse

break

- La instrucción **break** permite **salir repentinamente del bucle**
- Siempre **debe ir dentro de un bloque condicional if** puesto que de otra manera se ejecutaría siempre y el bucle no tendría sentido
- Es una forma de **terminar un bucle bajo condiciones excepcionales** que quizás complicarían la definición de la condición de permanencia si se definieran allí.

continue

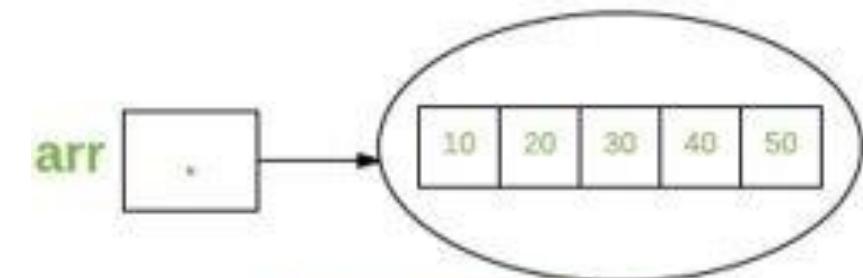
- La instrucción **continue** permite **terminar la iteración actual** de un bucle y fuerza que se continúe con la siguiente
- Siempre **debe ir dentro de un bloque condicional if** puesto que de otra manera se ejecutaría siempre y el bucle no tendría sentido
- Es una **forma de terminar una iteración concreta bucle bajo condiciones excepcionales** que quizás complicaría la lógica dentro de dicho bucle.

EJERCICIO

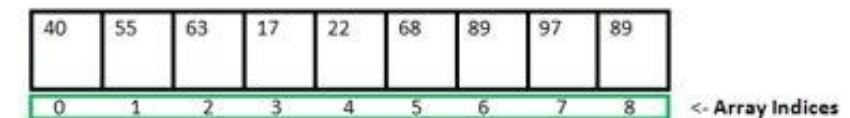


Vector / Array

- Un vector o array es una estructura de memoria que permite almacenar un **conjunto agrupado de valores de mismo tipo**.
- Para definirlo se especifica el **tipo de dato** de los valores que contendrá y la **capacidad** de éste, que será el número de valores que podrá almacenar.
- Cada elemento del array es **accesible** a través del **nombre** del array y la **posición** que ocupa dentro de éste
- La **primera posición** de una array siempre será la posición **0**



Ejemplo de array (Fuente: geeksforgeeks.org)



Array Length = 9
First Index = 0
Last Index = 8

Estructura de un array (Fuente: geeksforgeeks.org)

Declarar un vector

- Vector de **enteros**. Sin tamaño. Sin instanciar
 - `int[] notas;`
- Vector de **enteros**. Tamaño 2, **inicializado** (valores por defecto = 0)
 - `int[] notas = new int[2];`
- Vector de **cadenas**. Tamaño 2, **inicializado** con valores
 - `String[] palabras = new String[]{"una", "dos"};`



Cómo acceder a los valores de un vector

- Es posible acceder a todos los valores de un vector utilizando un bucle for para acceder posición a posición (mayor control)

```
for (int i = 0; i < notas.length; i++) {  
    System.out.println(notas[i]);  
}
```

- Es posible acceder a cada uno de los elementos de un vector utilizando un bucle for-each (de principio a fin)

```
for (int nota : notas)  
{ System.out.println(nota)  
;  
}
```

Cómo acceder a los valores de un vector

- También es posible acceder a cualquier posición directamente
- **int[]**
notas =
new int[2];
notas[0] =
10;
 - notas[1] = 5;
 - System.out.println("Nota primer examen: " + notas[0]);
 - System.out.println("Nota segundo examen: " + notas[1]);



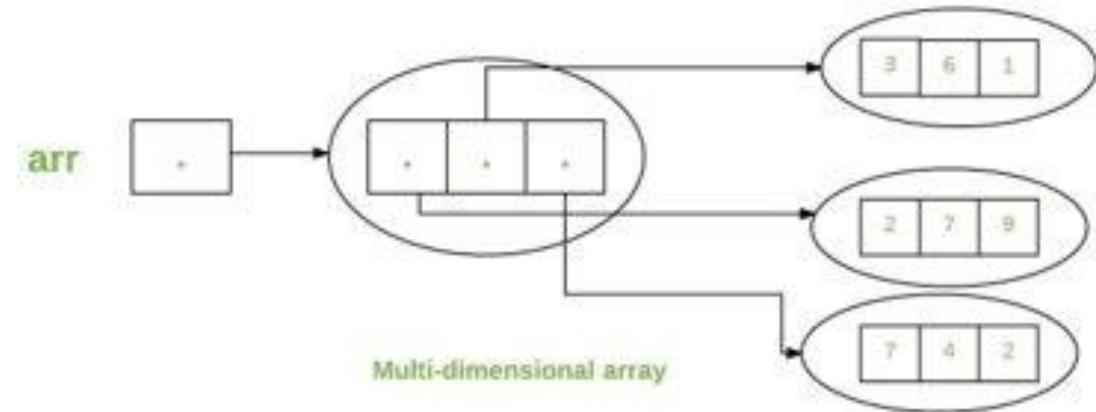
Operaciones con vectores

- En Java existe la clase Arrays para realizar operaciones con vectores:
 - **int binarySearch(vector[], valor):** Busca el valor especificado en el vector y devuelve su posición. En caso de que no lo encuentre devolverá el valor -1
 - **vector[] copyOf(vector[], nuevaLongitud):** Recorta o alarga el vector que se pasa como parámetro para que se ajuste a la nueva longitud. Devuelve el nuevo vector modificado.
 - **fill(vector[], valor):** Rellena el vector que se pasa como parámetro con el valor especificado en todas sus posiciones
 - **sort(vector[]):** Ordena todos los valores del vector
 - **String toString(vector[]):** Devuelve el contenido del vector representado por una cadena



Vectores de dos dimensiones

- A los vectores de dos dimensiones se les conoce también con el nombre de matrices
- Funcionan igual que los vectores unidimensionales pero en este caso hay dos posiciones que indicar: fila y columna
 - **int[][] valores = new int[2][5]**
- También se pueden considerar como "vectores de vectores", puesto que cada posición del vector contiene otro vector





String

- Una cadena de texto es una **secuencia de caracteres** (cualquiera)
- En Java se definen como el tipo **String** y su tamaño es **variable** (y no es necesario especificarlo)
- En Java se permite asignar valor a un String **como si se tratara de un tipo primitivo** pero realmente **es un objeto** (más adelante hablaremos de ello pero conviene conocer este dato ya)
- La cadena de texto es un **objeto inmutable** por lo que todas las operaciones que invoquemos sobre ella nunca la modifican, sino que devuelven su valor modificado (que tendrá que volver a ser almacenado sobre si misma u otra cadena)

Declarar un vector

- Cadena de texto con **valor nulo** (es la forma de especificar que no tiene valor)
 - String nombre = null;
- Cadena de **texto vacía** (es un valor válido)
 - String nombre = “”;
- Cadena de texto con **valor**
 - String nombre = “Santiago Faci”;
- Cadena de texto **compuesta de dígitos** (sigue siendo una cadena de texto)
 - String numero = “123”;

Cuándo utilizar una cadena de texto

- Por supuesto, usaremos una cadena cuando queramos almacenar texto.
- A veces, incluso cuando el valor que se quiere almacenar está compuesto exclusivamente de dígitos, éste debe almacenarse como cadena de texto, puesto que no son valores numéricos realmente ya que no soportan ninguna operación matemática ni se espera que se haga ninguna con ellos.
- Estos son algunos ejemplos
 - DNI
 - Número de la Seguridad Social
 - Cuenta Bancaria
 - Teléfono

Casos de uso con cadenas

- Un usuario introduce su nombre y apellidos en una caja de texto
 - Convertir a mayúscula/minúscula y eliminar los espacios a inicio y final
- Un usuario introduce su número de cuenta
 - Convertir la cadena en un array de caracteres para realizar el cálculo del dígito de control
- Un usuario introduce su número de DNI (con letra)
 - Convertimos el valor a un número para hacer los cálculos y luego lo almacenamos todo junto (con letra) concatenando.
- Un usuario introduce un precio en un formulario
 - Tras las comprobaciones (caja de texto vacía o valor sólo compuesto por dígitos) tendremos que parsearlo al tipo de dato que corresponda (float en este caso)

for-each

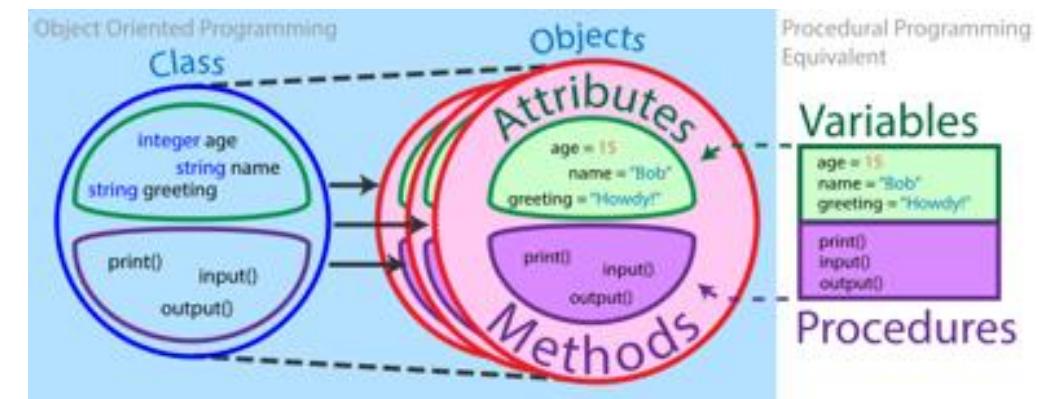
- Se ejecuta el conjunto de instrucciones **tantas veces como elementos haya en la colección que se recorre**. Además, en cada iteración se sirve cada elemento de la lista en la variable que se define en el for-each
- Ideal para **recorrer colecciones de datos** y cuando se necesita recorrerla **de inicio a fin**

EJERCICIO



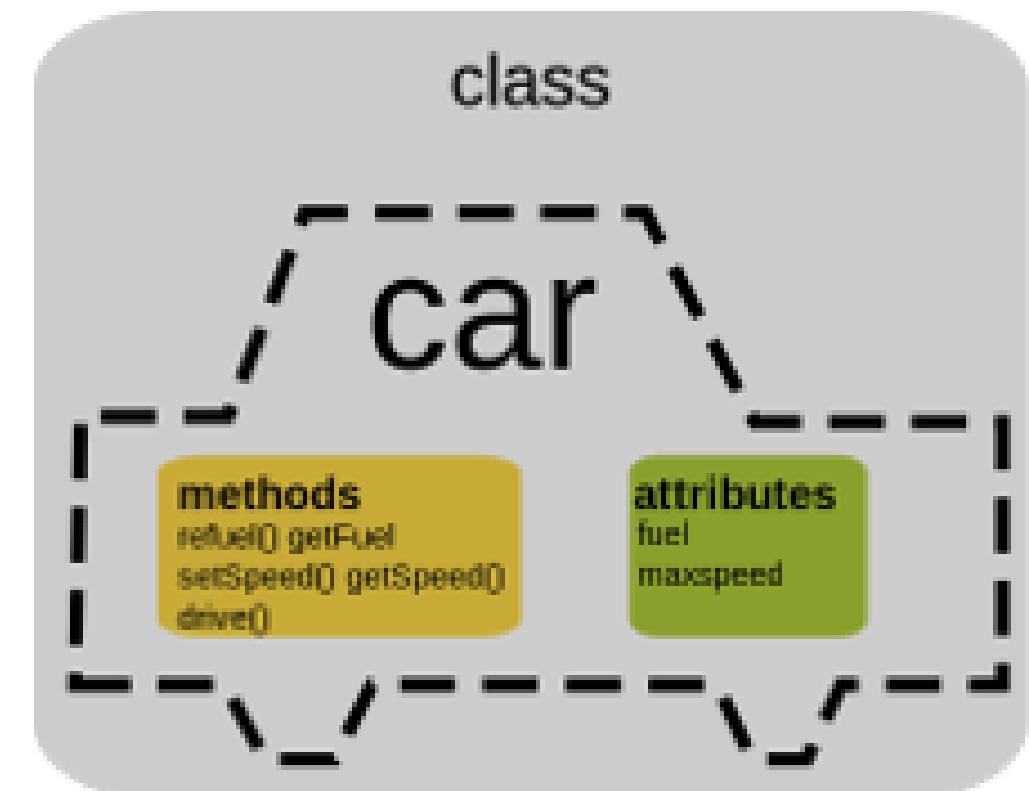
Introducción a la POO I

- Paradigma de programación
- Clase
 - Plantilla que permite definir conceptos o ideas que forma parte de un programa
- Objeto
 - Cada uno de los elementos que existen en un programa cuyo tipo viene definido por una clase
 - determinada
- Método
 - Cada una de las operaciones que un objeto puede realizar



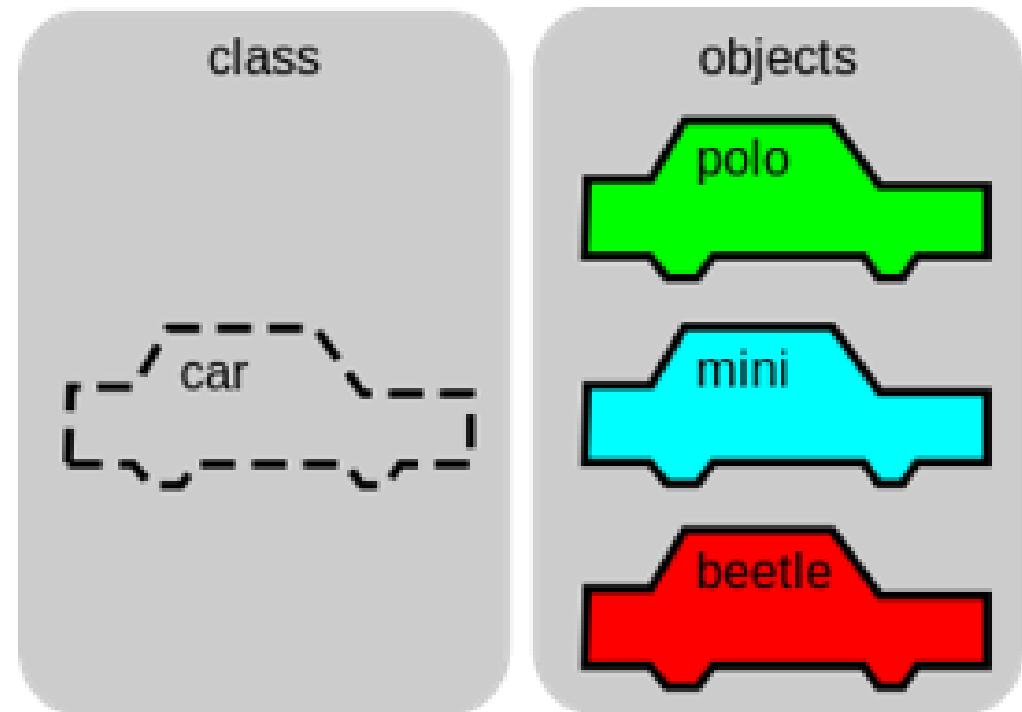
Clases

- **Atributos**
 - Definen las características que tendrá cada uno de los objetos de la clase.
 - Pueden ser de tipos primitivos o de otras clases (y de la propia clase)
- **Métodos**
 - Operaciones que puede realizar cualquier objeto de la clase



Objetos

- Son **instancias** de una clase
- **No** hay **límite** en el número de objetos que puede haber de una clase determinada
- Todos los objetos de una misma clase tienen las mismas características (**atributos**) y operaciones (**métodos**) pero **diferentes valores** para los primeros.



Declarar una clase

```
public class Coche {  
    private String marca;  
    private String modelo;  
    Coche(String marca,  
          String modelo) {  
        this.marca = marca;  
        this.modelo = modelo;  
    }  
    public String getMarca() {  
        return marca;  
    }  
    public String getModelo() {  
        return modelo;  
    }  
}
```

Modificadores

- Los modificadores de accesibilidad pueden aplicarse a clases, atributos o a métodos y permite fijar el nivel de visibilidad de un determinado elemento de una clase con respecto a las demás clases de la aplicación.
- **<ninguno>**: Conocido como package. El elemento será accesible sólo dentro del paquete
- **public**: El elemento será accesible desde cualquier parte del proyecto
- **protected**: El elemento será accesible desde el mismo paquete y las clases derivadas
- **private**: El elemento será accesible sólo dentro de la clase

Atributos

- **public**
String
nombre;
- **int**
cantidad;
- **private**
float precio;

Constructor

- El constructor es un tipo especial de método que se utiliza para inicializar los atributos de una clase en el momento de ser instanciada, cuando se invoca con la palabra reservada **new**.
- Hay que tener en cuenta que es posible definir **tantos constructores como sea necesario** (mismo nombre pero diferentes parámetros)
- **La palabra reservada **this** se utiliza para hacer referencia a elementos de la clase actual*

```
Coche(int modelo, int marca) {  
    this.modelo = modelo;  
    this.marca = marca;  
}
```



Getters y Setters

- Los getters y setters son una convención Java por la que se proporciona métodos **get** a aquellos atributos que son accesibles y métodos **set** a aquellos a los que se les puede modificar su valor.
- **Si el tipo del atributo es un booleano, se modifica la palabra get por is*

```
public String getNombre() { return nombre; }
```

```
public void setCapacidad(int capacidad) {  
    this.capacidad = capacidad; }
```



Métodos

- Los métodos permite definir las operaciones que pueden realizar los objetos de una clase.
- En caso de que el método no devuelva ningún resultado se utilizará la palabra reservada **void** en el lugar del tipo de devolución

```
public void llenarDeposito(int litros){  
    deposito += litros;  
}
```

Definir las clases para nuestro problema

- Para cada elemento que aparezca en el problema que queremos solucionar, diseñaremos una **clase** con tantos **atributos** como característica tenga (no confundir con los posibles valores de esas características) y tantos métodos como operaciones o tareas puedan realizarse sobre ese elemento.
- Por lo general (y por ahora) declararemos todas las **clases como públicas** y cada una se escribirá en un fichero con el mismo nombre que la clase (y la extensión .java)
- Por lo general (y sino se demuestra lo contrario) todos los atributos de la clase serán **privados** (encapsulamiento)
- Diseñar al menos un constructor o bien algunos con las combinaciones de atributos más habituales a la hora de inicializar los objetos de la clase
- Siempre será más conveniente un método que haga cambios sobre un atributo de la clase que modificar directamente dicho atributo



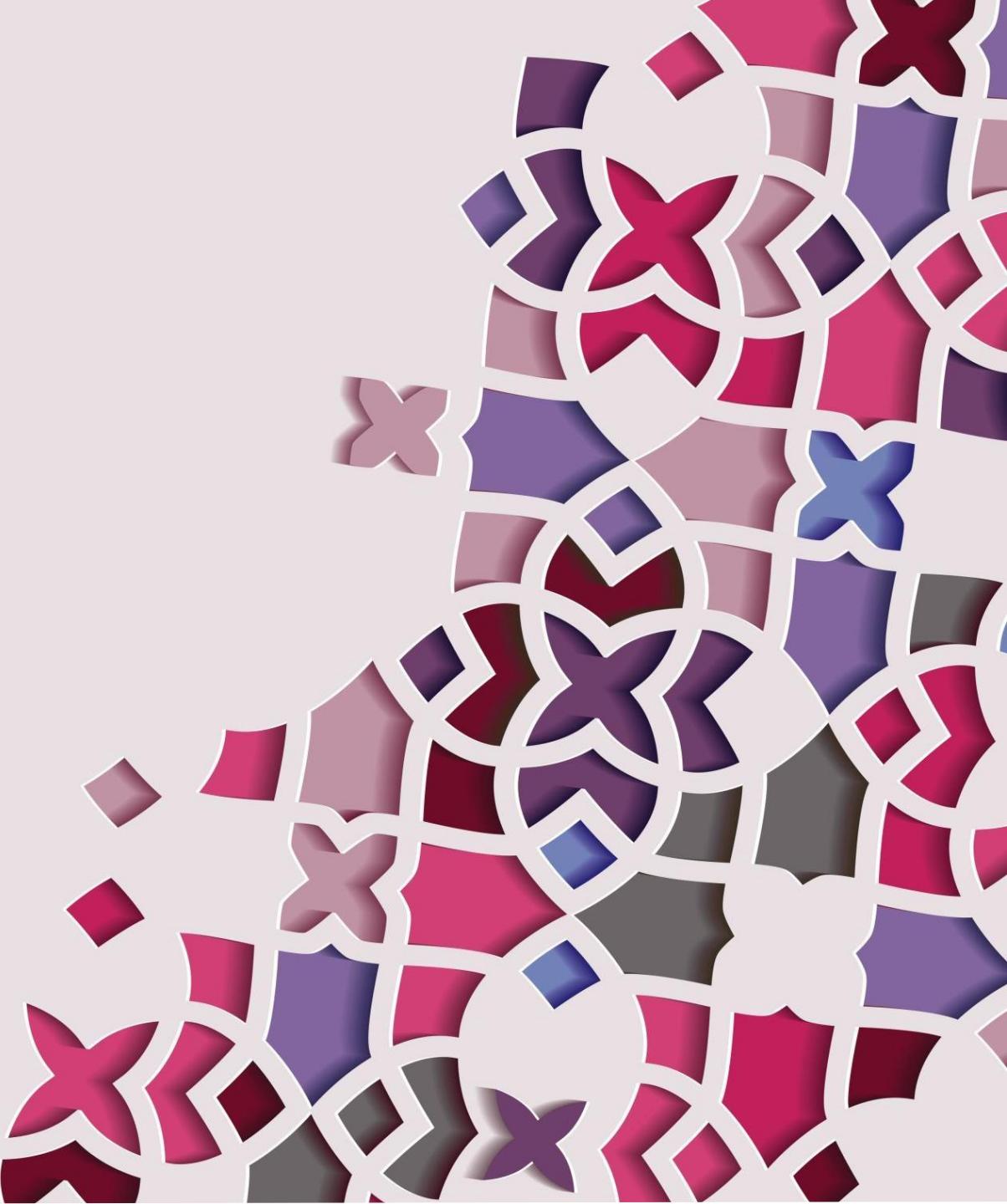
```
public class  
Propietario {  
    private String dni;  
    . . .  
    private Vehiculo  
    vehiculo;  
}
```

```
public class  
Vehiculo {  
    private String  
    matricula;  
    . . .  
    private Propietario  
    propietario;  
}
```

Relaciones entre clases

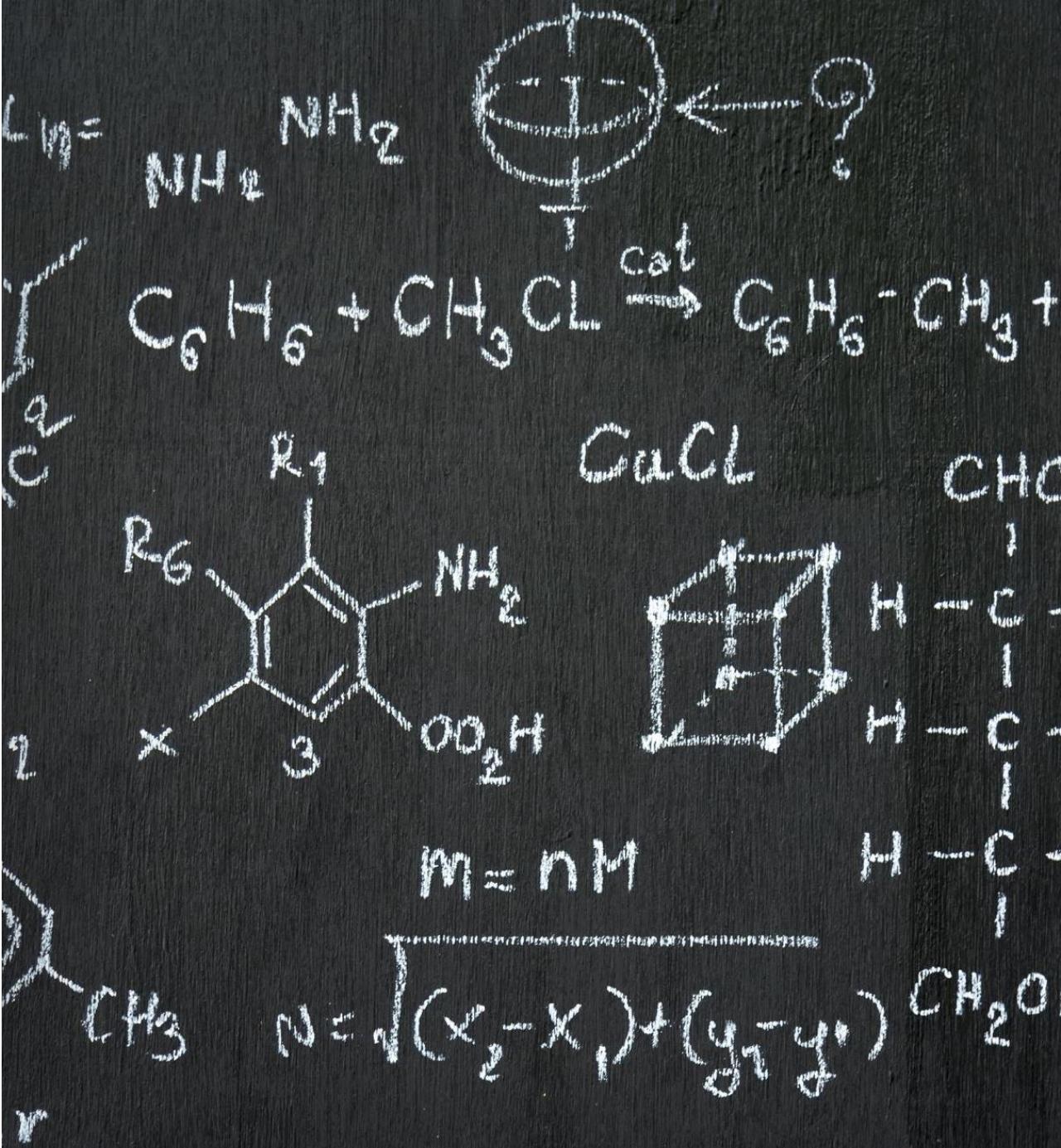
- Si queremos establecer una relación entre dos clases, haremos que cada una de ellas tenga un atributo de la otra
- Las clases pueden hacerse referencia unas a otras. Es decir, en una clase pueden aparecer atributos cuyo tipo es a su vez otra clase
- En una clase también pueden aparecer atributos cuya clase sea la propia clase

EJERCICIO



Clase Math

- Clase con métodos estáticos para realizar todo tipo de operaciones matemáticas
 - **abs(numero)**: Valor absoluto de un número (útil para calcular la diferencia o distancia entre dos valores)
 - **ceil(numero)**: Obtiene el siguiente valor entero por encima del que se pasa como parámetro
 - **floor(numero)**: Obtiene el anterior valor entero por debajo del que se pasa como parámetro
 - **max(numero1, numero2)**: Obtiene el máximo de dos valores dados
 - **min(numero1, numero2)**: Obtiene el mínimo de dos valores dados
 - **random()**: Devuelve un número aleatorio entre 0 y 1
 - **round(numero)**: Redondea el número que se pasa como parámetro (tiene otra opción donde indicar la precisión)
 - **sqrt(numero)**: Calcula la raíz cuadrada del número que se pasa como parámetro



A close-up photograph of a pair of dark-rimmed glasses lying on top of an open book. The book's pages are visible, showing some text and markings. The background is slightly blurred.

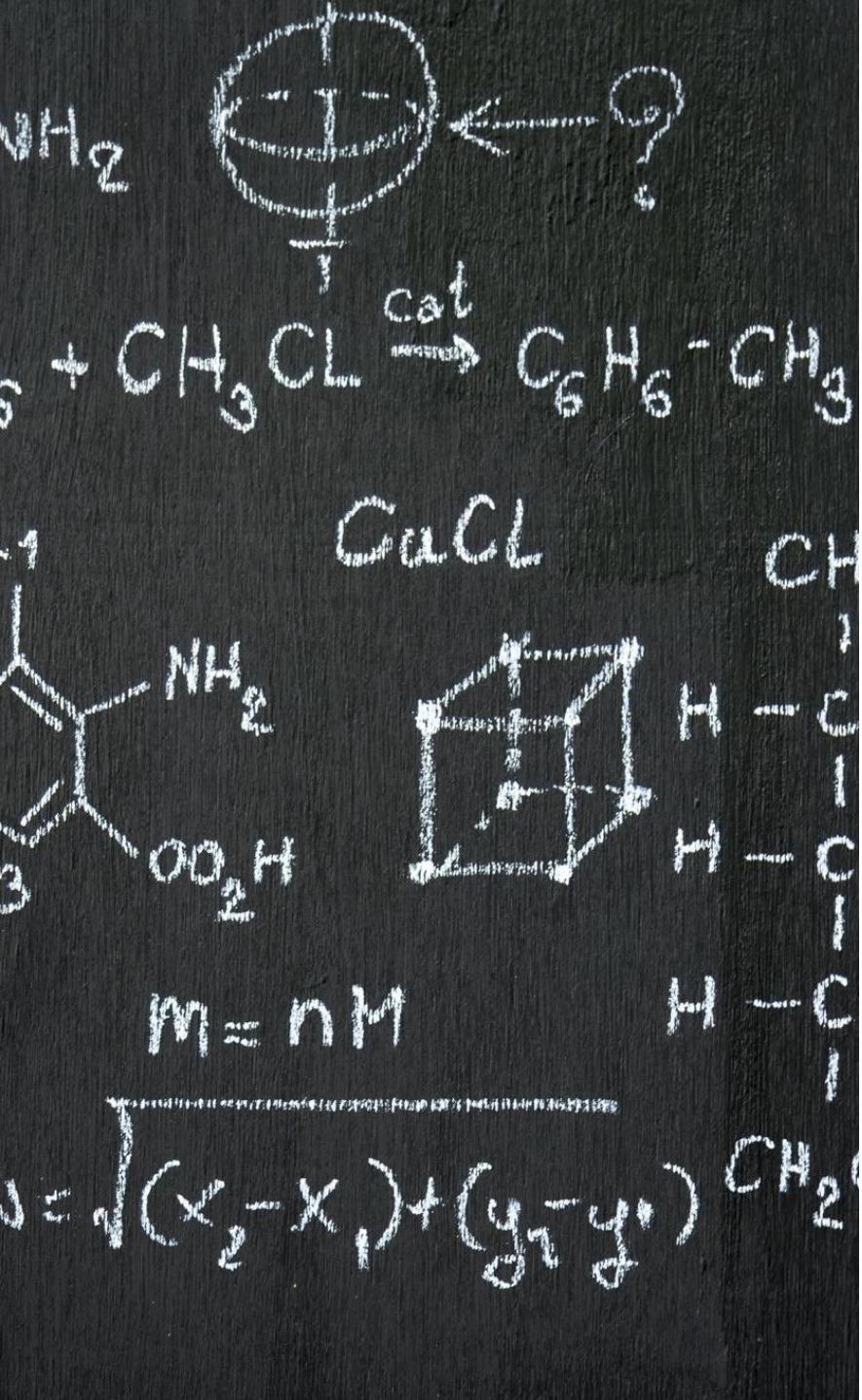
Herencia en Java

- Para heredar de una clase se utiliza la palabra reservada **extends**
- La clase que hereda se conoce como clase **heredada** o clase hija. La clase de la que se hereda se conoce como clase **base** o clase padre.
- Cuando una clase A hereda de una clase B, la clase A adquiere todos los **atributos y comportamiento (atributos y métodos)** de la clase B (aunque quizás no pueda acceder a algunos de ellos por los modificadores de accesibilidad)
- En Java, **sólo se puede heredar de una clase** (sin contar la clase Object de la que heredan todas las clases de forma implícita). En Java se idearon las **interfaces** para suplir en parte esta carencia.
- No hay límite en cuanto al número de clases que pueden heredar de una clase determinada
- No hay límite en cuanto al nivel de profundidad en el árbol de herencias entre clases
- Es posible evitar que se herede de una clase utilizando el modificador **final** a la hora de declararla
- Cuando una clase Coche hereda de una clase Vehiculo, un objeto de clase Coche se puede considerar ahora de tipo Vehículo pero no al revés.
- Cuando una clase hereda de otra está obligada a implementar constructores apropiados para invocar a los de la clase base

Clases abstractas



- Las clases abstractas son clases que se definen con el único propósito de heredar de ellas desde otras clases.
- Se definen con la palabra reservada **abstract** -> **public abstract class** Vehiculo { . . . }
- No es posible instanciar objetos de una clase definida como abstracta
- Las clases abstractas pueden definir métodos sin cuerpo. Eso significa que será obligatorio que quien hereda de esta clase tendrá que definirlo (excepto que también sea abstracta). -> **public abstract void** hacerAlgo();
- El hecho de que existan métodos abstractos hace que diferentes objetos que hereden de una misma clase tendrán mismos métodos pero diferente implementación. Cuando éstos sean manejados a través de variables/parámetros de la clase base, veremos que el mismo método hará cosas diferentes en función del objeto real al que apunte dicha variable -> **Polimorfismo**



Sobrescritura de métodos

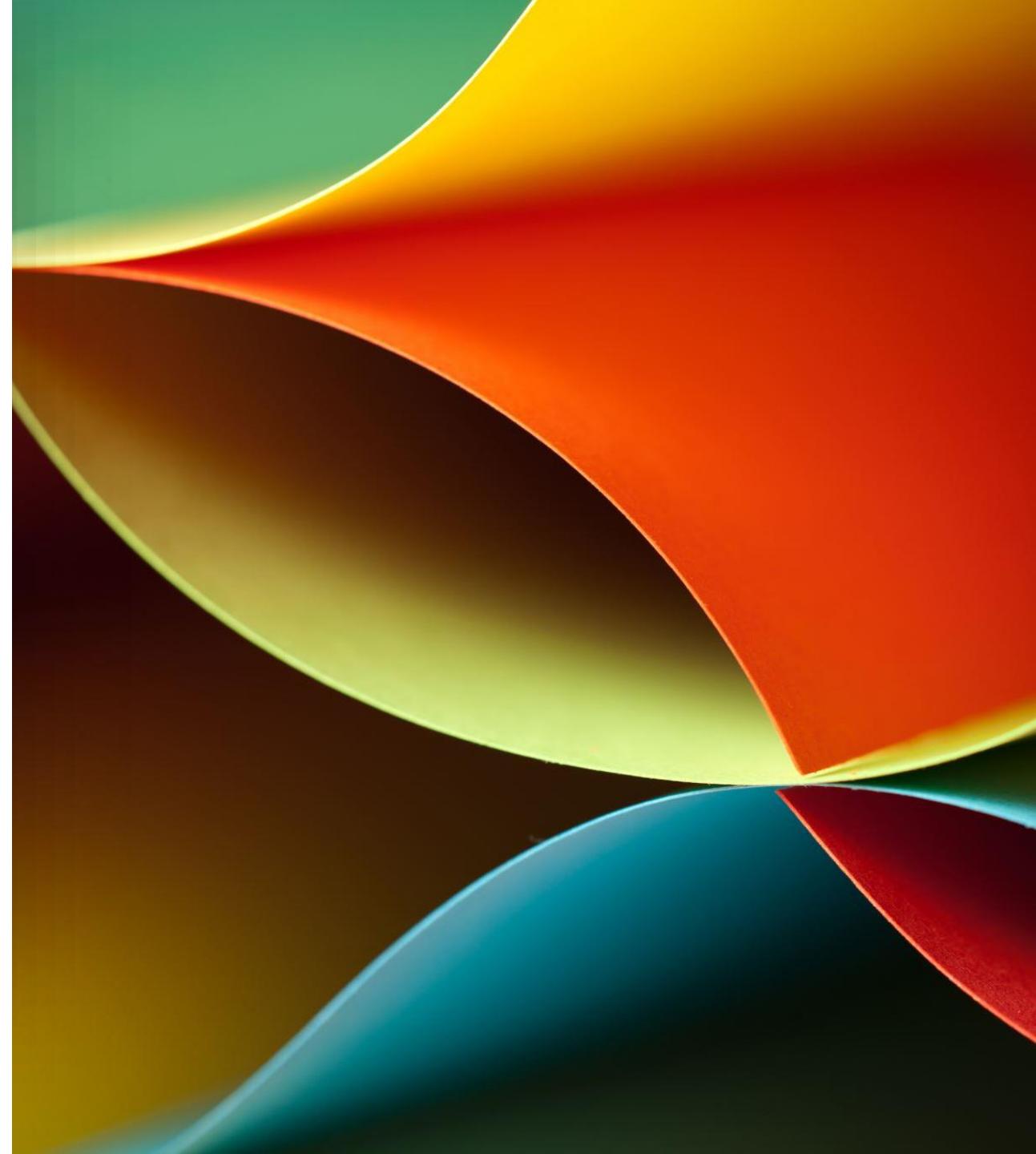
- Cualquier clase que herede de alguna clase base podrá sobrescribir los métodos que herede
 - Para ampliarlo, añadiendo más código (es posible invocar a la versión heredada y añadir código nuevo)
 - Para sustituirlo por otra implementación (simplemente redefiniendo el método de nuevo)
- Cuando un método sobrescribe a otro ya heredado, se debe anotar con **@Override**
- Si el método simplemente sobrecarga al método heredado (diferente firma) no está sobrescribiendo al heredado, es otro método

EJERCICIO



¿Qué es una interface?

- Es una clase que sólo permite la definición de métodos abstractos (sin implementación) y atributos que serán considerados, implicitamente, como public static final
- La idea es definir comportamientos que no puede ser implementados de forma genérica pero se quiere obligar a que existan
- Cuando una clase "hereda" de una interface se dice que la implementa, y se usa la palabra reservada implements. Y nunca diremos que "hereda de", sino que diremos que "implementa"
- Java permite que una clase implemente de varias interfaces, siempre y cuando implemente el código de los métodos definidos en ellas (o se haga abstracta la propia clase)
- Por definición, los métodos de una interface son públicos, por lo que no hace falta indicarlo
- En la práctica, funciona como una clase abstracta donde todos sus métodos son abstractos



EJERCICIO

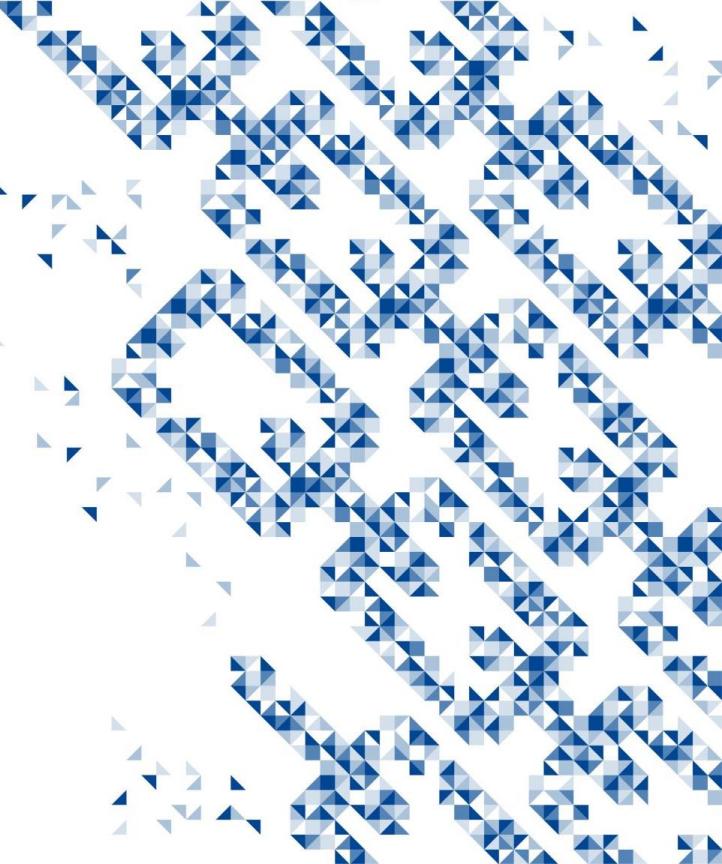


EJERCICIOS DE REPASO

Excepciones en Java



- Son errores que se producen en tiempo de ejecución
- Se producen porque la línea de código que debe ejecutarse no puede hacerlo por alguna razón
- Son errores imposibles de prever durante el desarrollo de la aplicación
- Si se producen la aplicación falla y se cierra sin previo aviso. Puesto que no se pueden prever, simplemente se captura para que la aplicación no falle y para notificar al usuario de alguna manera
- Ejemplos:
 - Durante la descarga de un fichero, falla la conexión a Internet
 - Al escribir a disco, no disponemos de permisos para hacerlo o el disco está lleno
 - Accedemos a atributos o métodos de un objeto cuyo valor no existe (es nulo)
 - Accedemos a una posición no válida de un array
 - Realizamos una operación matemática no válida (división por cero)



Bloque try-catch

- El código que es susceptible de generar una excepción, lo tendremos que incluir dentro de un bloque try-catch:

```
try {  
    <línea que genera la excepción>  
} catch (Exception ex) {  
    // La ejecución sigue aqui cuando se produce  
    // la excepción  
    // En el objeto e hay información sobre la  
    // excepción  
}
```



Instrucción throws

- La instrucción throws se utiliza cuando el método en el que lanza la excepción no es el adecuado para capturarla. Con esta instrucción notifico que dicho método puede lanzar esa excepción y será quien lo haya invocado, quién tendrá que incluir su bloque try-catch para capturarla

```
public void unMetodo() throws IOException
{
    // En algún lugar de este método se produce
    // una excepción

    // de tipo IOException pero no es el momento
    // para capturarla

    ...
}

...
```

Excepciones en Java



- Por ejemplo:
 - Durante la descarga de un fichero, falla la conexión a Internet
 - Al escribir a disco, no disponemos de permisos para hacerlo o el disco está lleno (operaciones entrada/salida)
 - Accedemos a atributos o métodos de un objeto cuyo valor no existe (es nulo)
 - Accedemos a una posición no válida de un array
 - Realizamos una operación matemática no válida (división por cero)

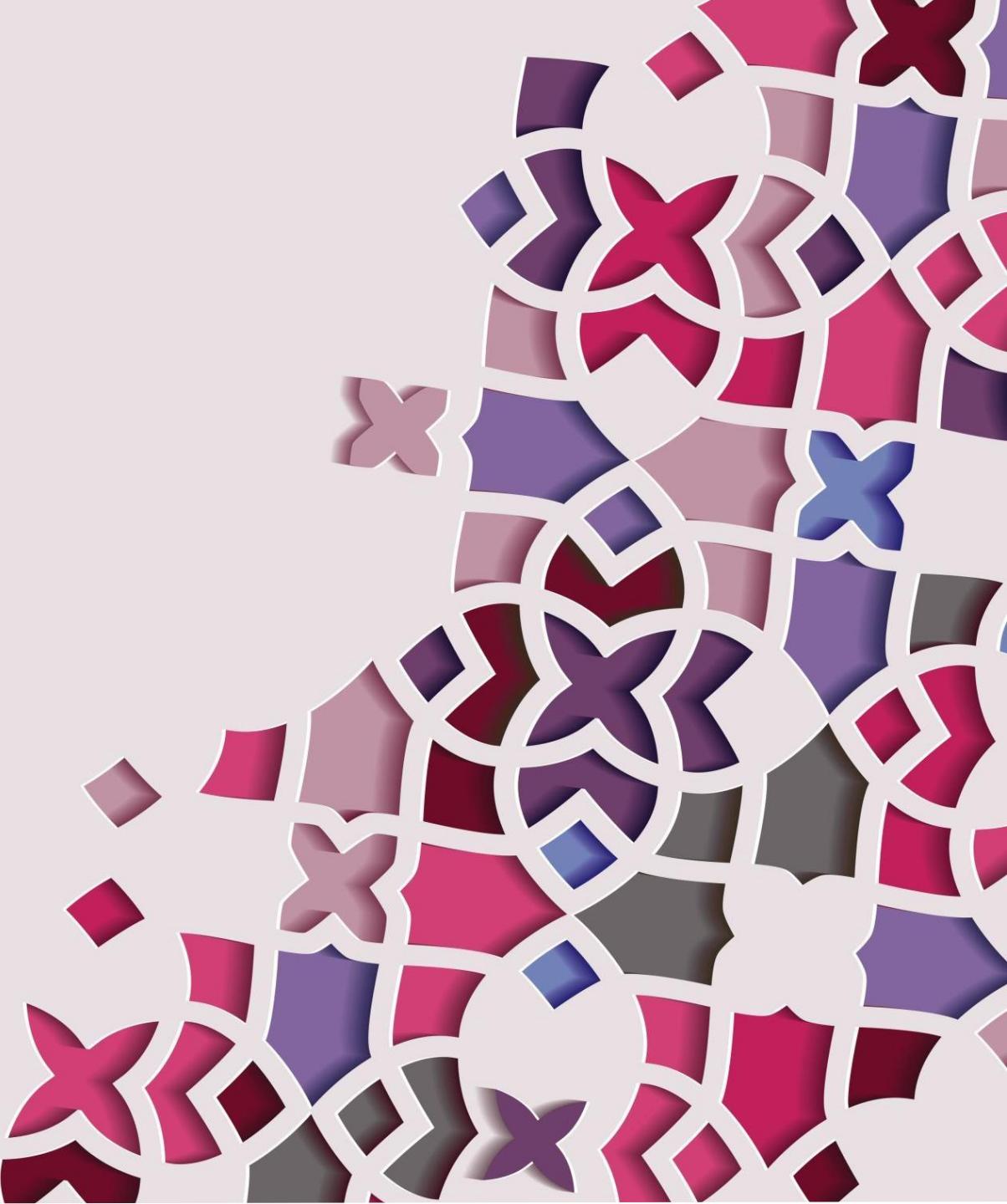
Instrucción throw

- También es posible lanzar explícitamente una excepción con la sentencia throw en cualquier momento:

```
throw new Exception("Error por  
alguna causa");
```

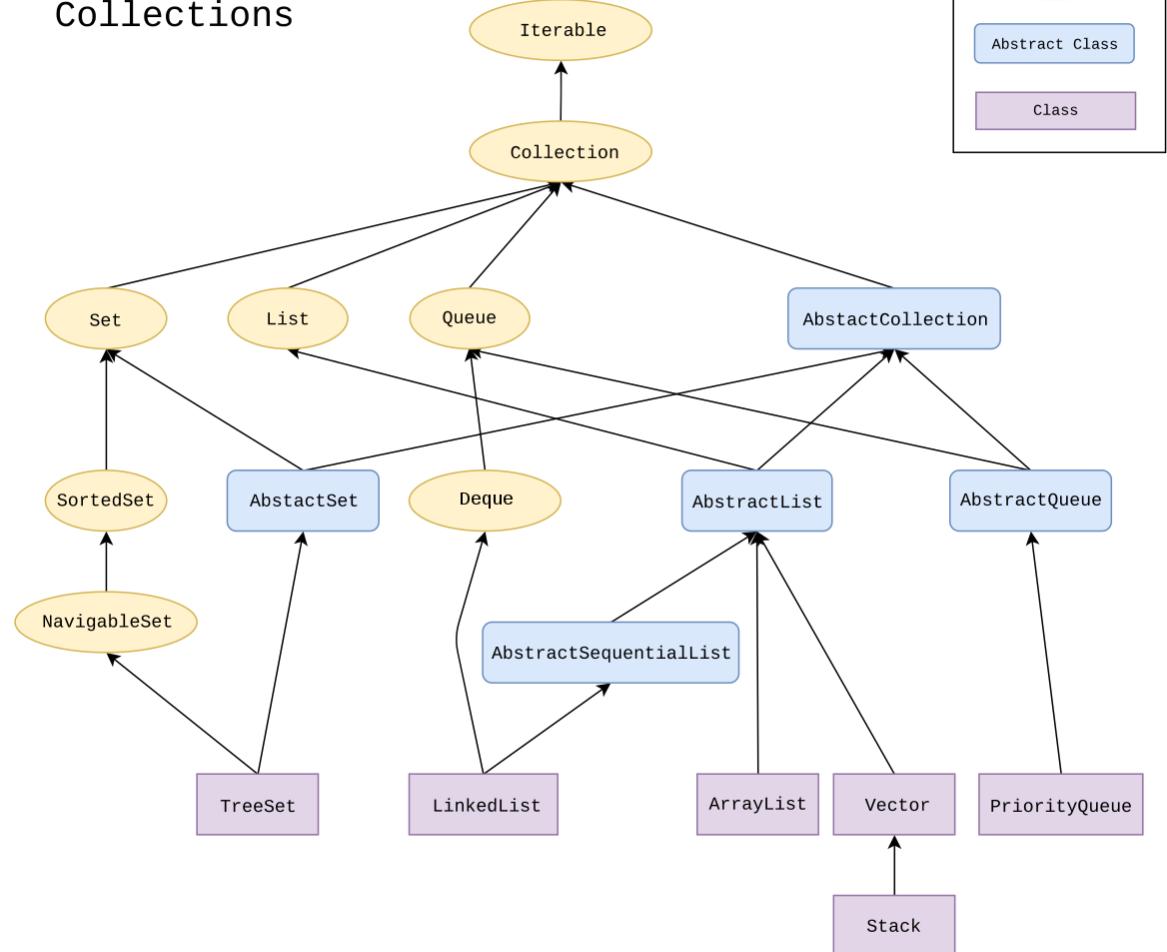
- El método que lanza esta línea de código tendrá que notificar que puede lanzar excepción (instrucción throws) y quién invoque a dicho método deberá dejarla pasar (instrucción throws de nuevo) o bien capturarla (try-catch)

EJERCICIO



LISTAS

Collections

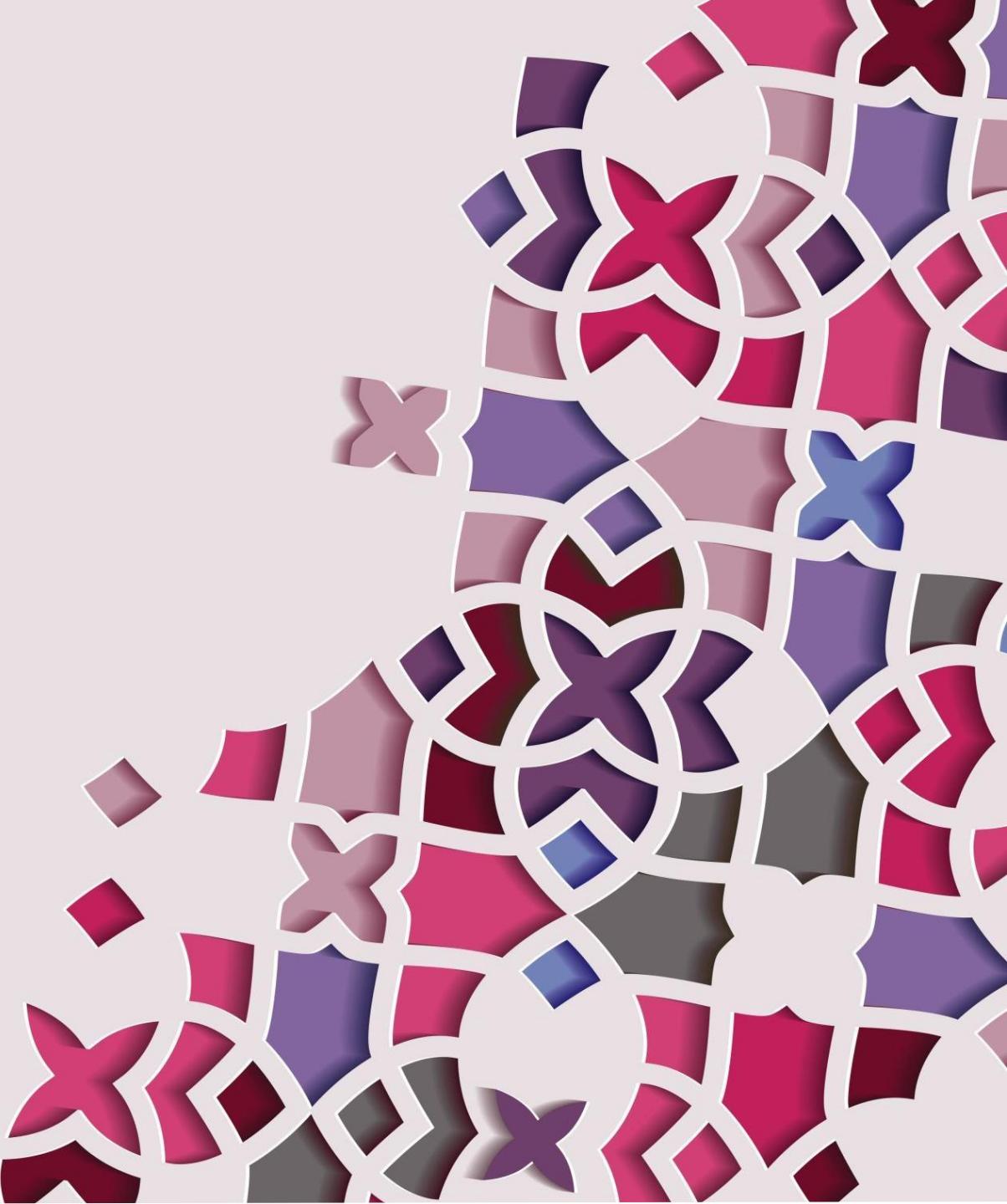


EJERCICIO



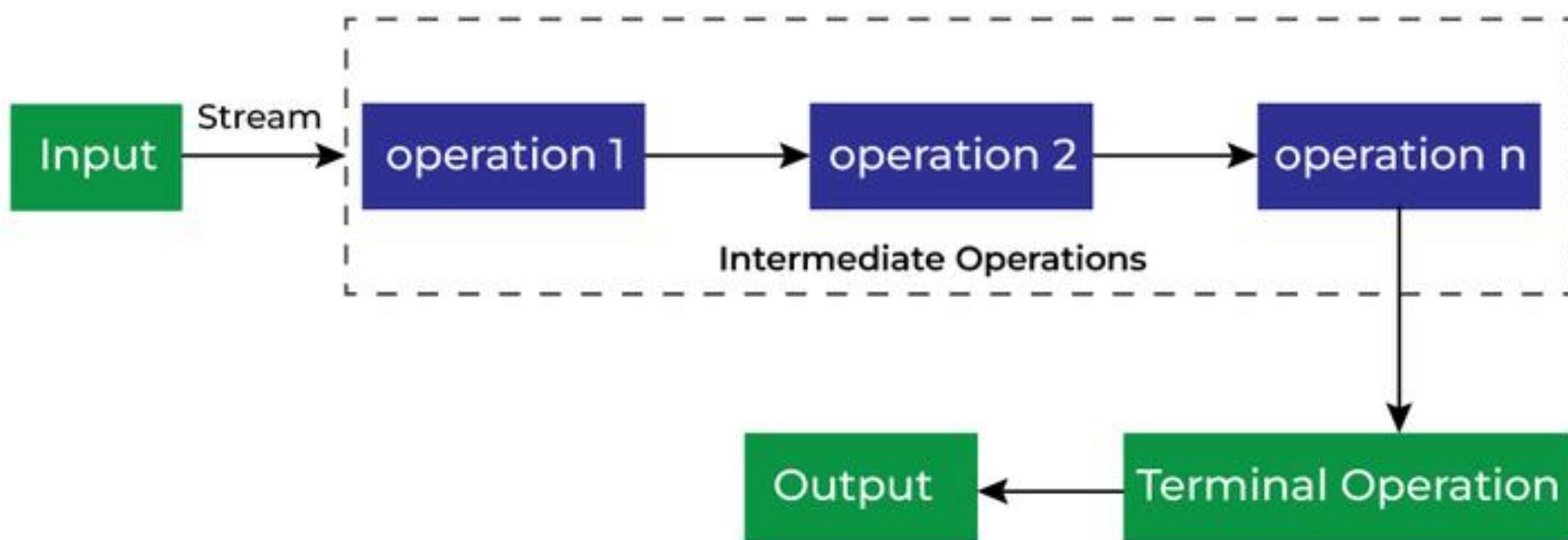
Interface	Duplicates Allowed?	Null Values Allowed?	Insertion order preserved?	Iterator	Data Structure
List	Yes	Yes, Multiple null values are allowed	Yes and can retrieve using index	Iterator, ListIterator	Array
Set	No	Yes but only once	No	Iterator	Underlying Map implementation
Map	Not for keys	Yes but only once for keys, can have multiple null values	No	Through keyset, value and entry set	Hashing techniques

EJERCICIO

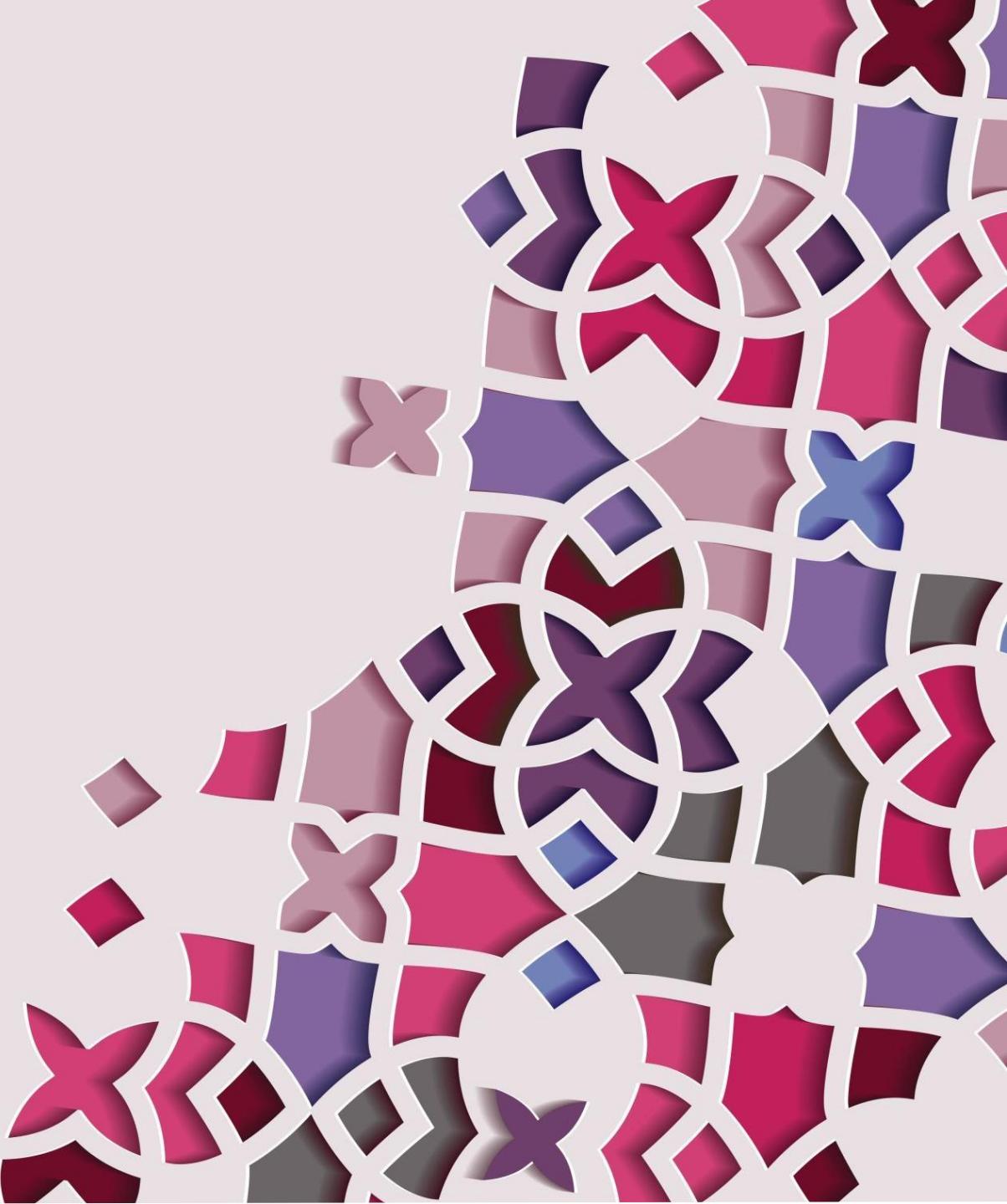


LAMBDAS

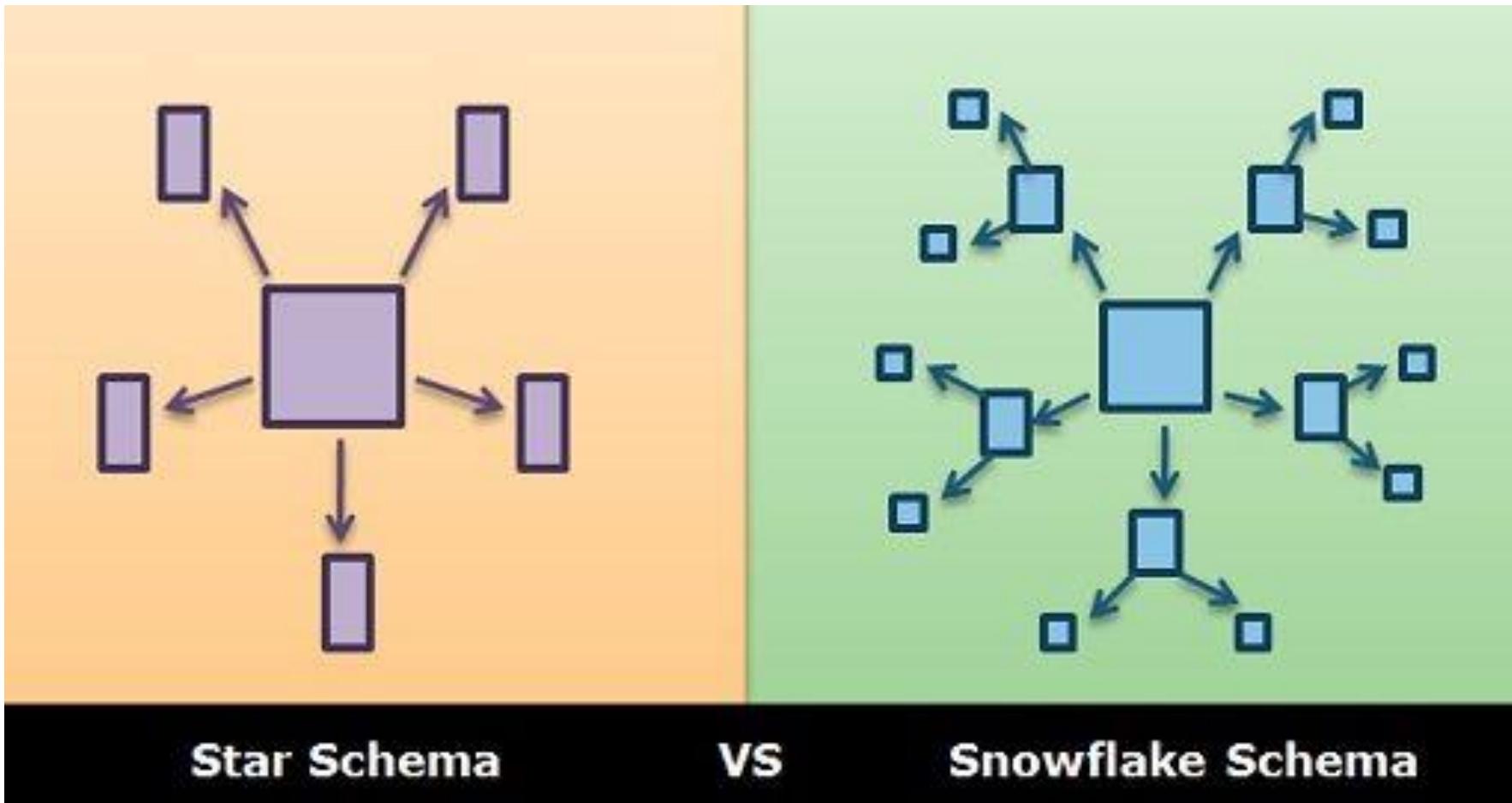
```
import static java.lang.System.out;  
  
() -> out.println("Bienvenidos al blog");  
(int base, int altura) ->{return (base*altura)/2};  
(String cadena) -> {String c = cadena;  
    c = cadena.toUpperCase();  
    return c;  
};  
Integer entero -> entero.min(7,4) >= 4;  
String longitud -> longitud.length() > 10;
```

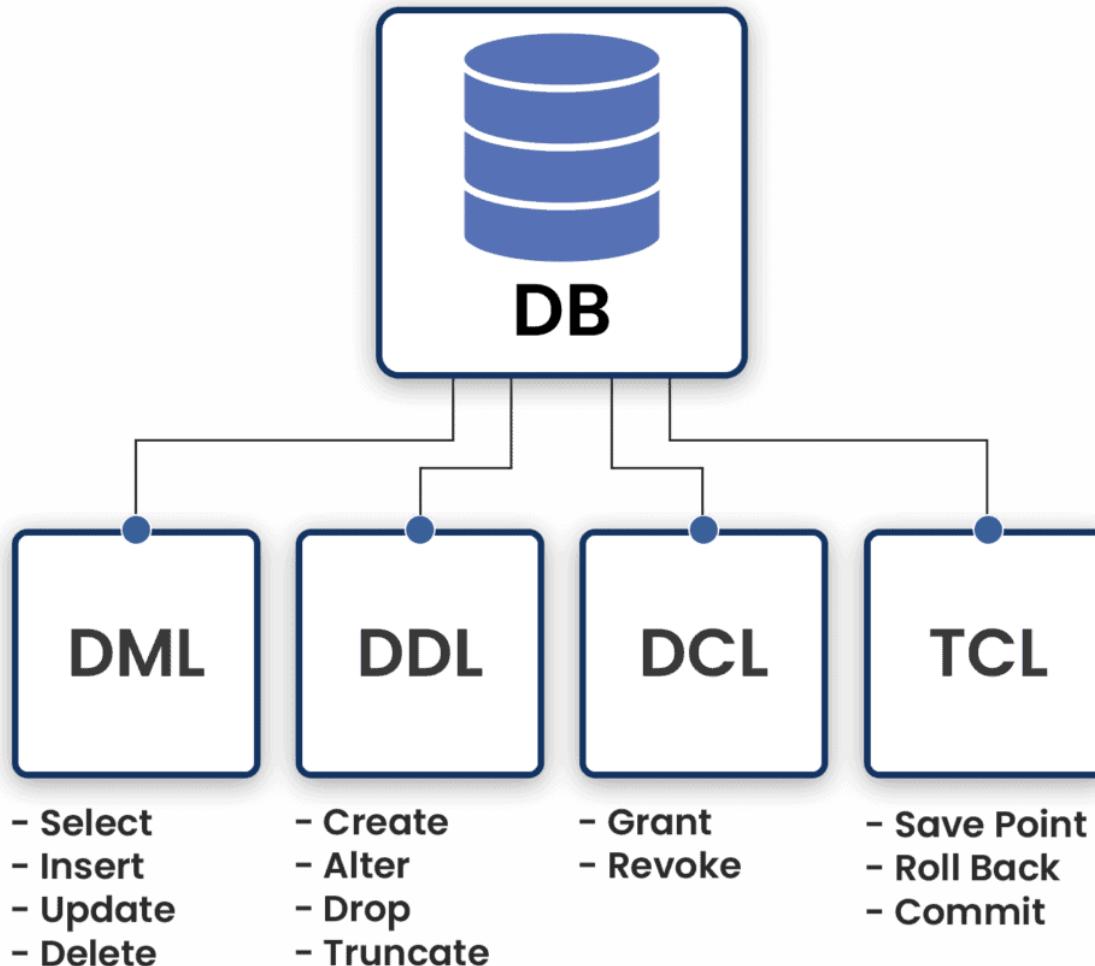


EJERCICIO



BBDD

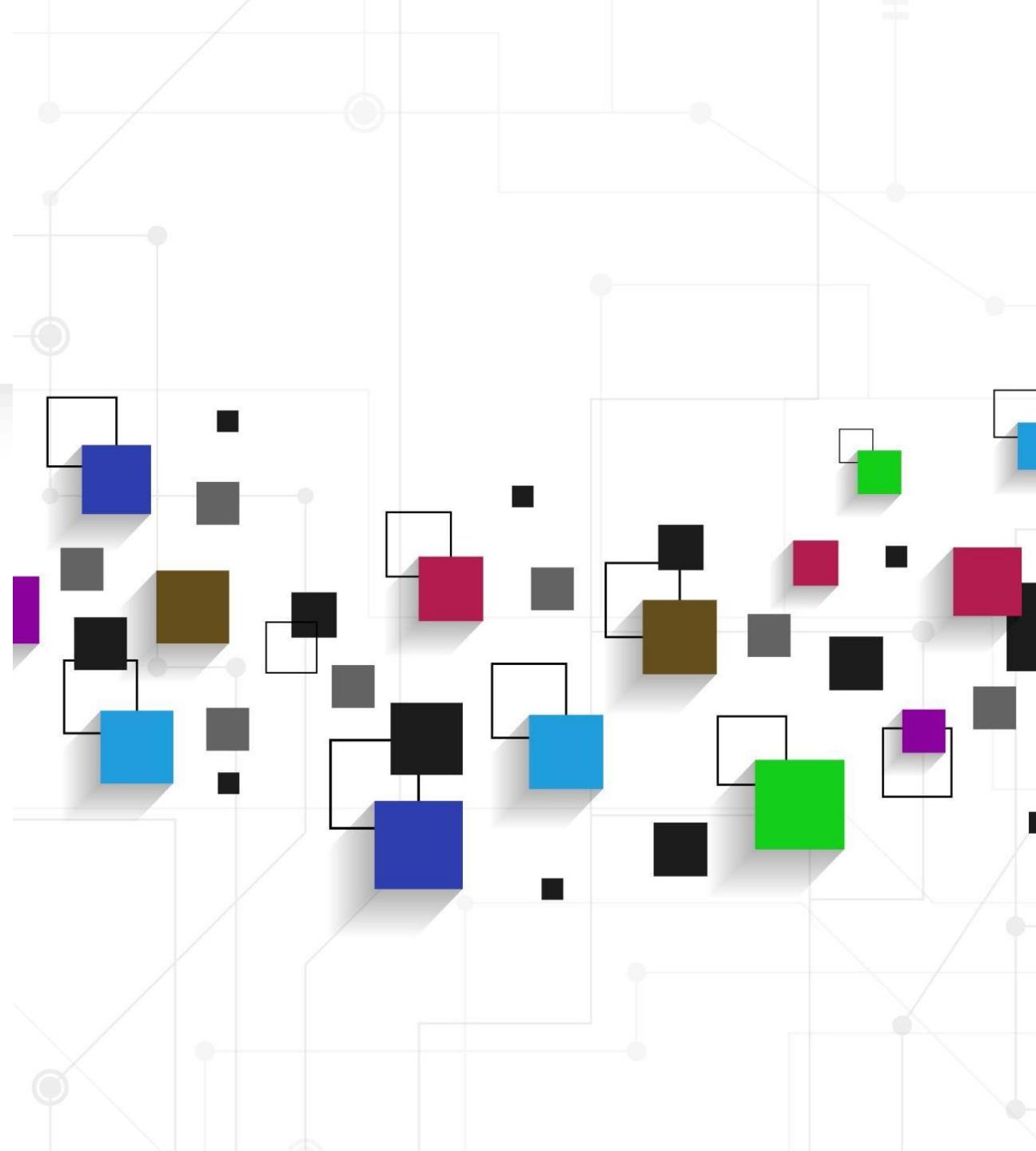




SQL CREATE

Para crear una base de datos, así como una tabla en esta, utilizaremos la siguiente sintaxis:

```
CREATE DATABASE mibasededatos;  
USE mibasededatos; CREATE TABLE  
mitabla1 (Columna1 (TipoDeDato),  
Columna2 (TipoDeDato)...);
```

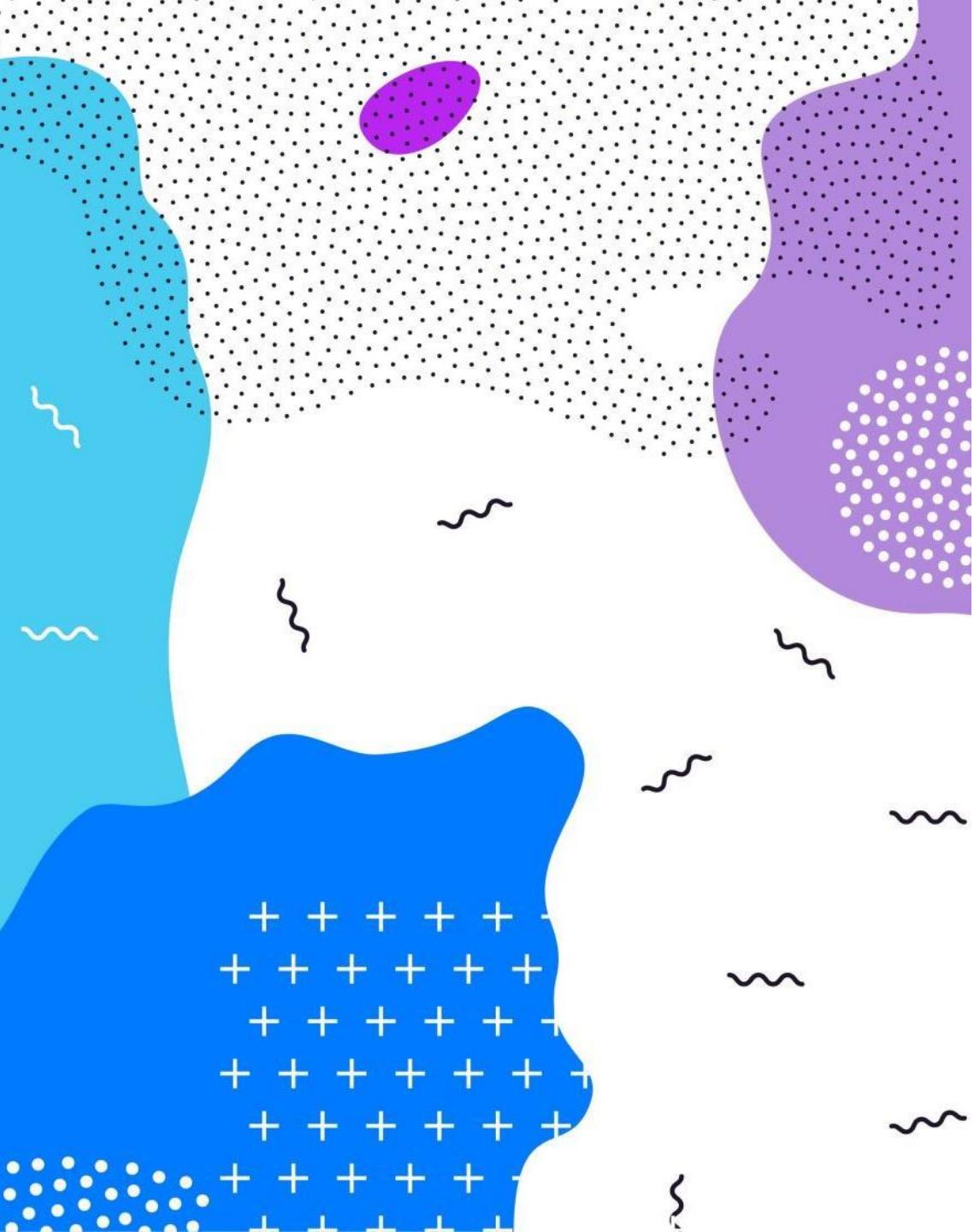


SQL SELECT

- Una de las sentencias SQL más importantes es SELECT, pues nos permite realizar consultas sobre los datos almacenados en la base de datos. La sintaxis de la consulta SELECT es la siguiente:

```
SELECT * FROM nombretabla;  
SELECT columna1, columna2 FROM  
nombretabla;
```





SQL WHERE

- La cláusula WHERE se utiliza para aplicar filtros a las consultas, es decir, para seleccionar únicamente aquellas filas de la tabla que cumplan una determinada condición.
- El valor de la condición debe situarse entre comillas simples (").
- Así, por ejemplo, aplicaremos un filtro para seleccionar aquellas personas cuyo nombre sea ANTONIO:

```
SELECT * FROM personas WHERE nombre =  
'ANTONIO';
```

SQL INSERT

- La sentencia INSERT INTO nos permite insertar nuevas filas en una tabla. Para ello, podemos proceder de dos formas distintas:

```
INSERT INTO nombretabla VALUES  
(valor1, valor2, valor3...);  
INSERT  
INTO nombretabla (columna1,  
columna2, columna3...) VALUES  
(valor1, valor2, valor3...);
```



SQL UPDATE

- La sentencia UPDATE se utiliza para modificar valores en una tabla. Su sintaxis es la siguiente:

UPDATE nombretabla SET columna1 = valor1, columna 2 = valor2 WHERE condición;
La cláusula SET establece los nuevos valores para las columnas indicadas, mientras que la cláusula WHERE selecciona las filas que queremos modificar.

- Atención: si omitimos la cláusula WHERE, se modificarán los valores en todas las filas de la tabla por defecto.

The image shows a blackboard covered in handwritten mathematical work. At the top right, there's a diagram of a rectangle divided into four quadrants with labels P, H, T, and I. Below it, a formula is written: $D(x) = a + b + 4.31447$. To the left, there's a circled equation $\sqrt{a^2 + b^2} = x^2$ with a note 'nx'. Further down, another circled equation shows $x^2 - y^2 = ab + 4c$. On the right side of the board, there's a system of equations involving $xy = c$, $cx - cy = 0$, and $2\pi = c$. In the center, there's a complex fraction: $\frac{24 \frac{fx}{y} + \frac{d^2 + 3^2}{c}}{c}$. Below this, a circled 'men' is followed by $= 584. + n^{av}$. At the bottom left, there's a circled 'x=9.23' next to a circled 'y=14!'. To the right, there's a large circled formula involving N_{30} and x . At the very bottom, there's a circled 'A' and a circled 'B = 9 + x'.

SQL DELETE

- La sentencia DELETE está destinada a borrar filas de una tabla. Su sintaxis es la siguiente:

DELETE FROM nombretabla WHERE
nombrecolumna = valor; Si queremos
borrar todos los registros o filas de una
tabla, utilizaremos la sentencia:
DELETE * FROM nombre tabla;

$y = g(x)$

Secant Lines

$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$

$f(x) = \lim_{h \rightarrow 0} \frac{(x+h)^2 - x^2}{h}$

$= \lim_{h \rightarrow 0} \frac{x^2 + 2xh + h^2 - x^2}{h}$

$= \lim_{h \rightarrow 0} \frac{2xh + h^2}{h}$

$\frac{g(x+h) - g(x)}{h}$

$= \lim_{h \rightarrow 0} h(2x + h)$

API JDBC

- **DriverManager**. Esta clase está destinada a registrar el controlador de un tipo de bases de datos específico (por ejemplo, MySQL), así como a establecer una conexión entre la base de datos y el servidor a través de su método `getConnection()`.
- **Connection**. Esta interfaz representa el establecimiento de una conexión de la base de datos (sesión) desde la que podemos crear declaraciones para ejecutar consultas y recuperar resultados, obtener metadatos sobre la base de datos, cerrar sesión, etc.
- **Statement y PreparedStatement**. Estas interfaces se utilizan para ejecutar, respectivamente, consultas SQL estáticas y parametrizadas. **Statement** es la superinterfaz de **PreparedStatement** y sus **métodos más utilizados** son:
 - **boolean execute(String sql)**. Ejecuta una declaración SQL general. Devuelve true si la consulta devuelve un `ResultSet`, y false si la consulta devuelve un recuento de actualizaciones o no devuelve nada. Este método solo se puede utilizar con `Statement`.
 - **int executeUpdate(String sql)**. Ejecuta una declaración INSERT, UPDATE o DELETE y devuelve una cuenta de actualización que indica el número de filas afectadas (por ejemplo, 1 fila insertada, 2 filas actualizadas, 0 filas afectadas...).
 - **ResultSet executeQuery(String sql)**. Ejecuta una instrucción SELECT y devuelve un `ResultSet` con los resultados obtenidos mediante la consulta.
- **ResultSet**. Contiene datos de la tabla devueltos por una consulta SELECT.
- A continuación, nos ocuparemos de cómo crear una aplicación en Java que realice las operaciones CRUD en la base de datos con JDBC.



EJERCICIO





INTRODUCCIÓN A PRUEBAS



vs.



**Pruebas
dinámicas**

**Pruebas
estáticas**

TIPOS DE SOFTWARE TESTING

Pruebas funcionales

- Pruebas unitarias
- Pruebas de integración
- Pruebas de sistema
- Pruebas de sanidad
- Pruebas de humo
- Pruebas de interfaz
- Pruebas de regresión
- Pruebas de aceptación

Pruebas no funcionales

- Pruebas de rendimiento
- Prueba de carga
- Pruebas de estrés
- Pruebas de volumen
- Pruebas de seguridad
- Pruebas de compatibilidad
- Pruebas de instalación
- Pruebas de recuperación
- Pruebas de confiabilidad
- Pruebas de usabilidad
- Pruebas de conformidad
- Pruebas de localización



Test de Caja Negra



Test de Caja Blanca

Estrategia de Pruebas

- Planificación: Definición de estrategias, alcance, recursos y calendario de pruebas.
- Diseño de Casos de Prueba: Creación de escenarios y casos de prueba basados en requisitos.
- Ejecución de Pruebas: Ejecución de casos de prueba y registro de resultados.
- Análisis de Resultados: Evaluación de los resultados de las pruebas y seguimiento de los defectos encontrados.
- Informe y Retest: Creación de informes de pruebas y reevaluación tras la corrección de defectos.



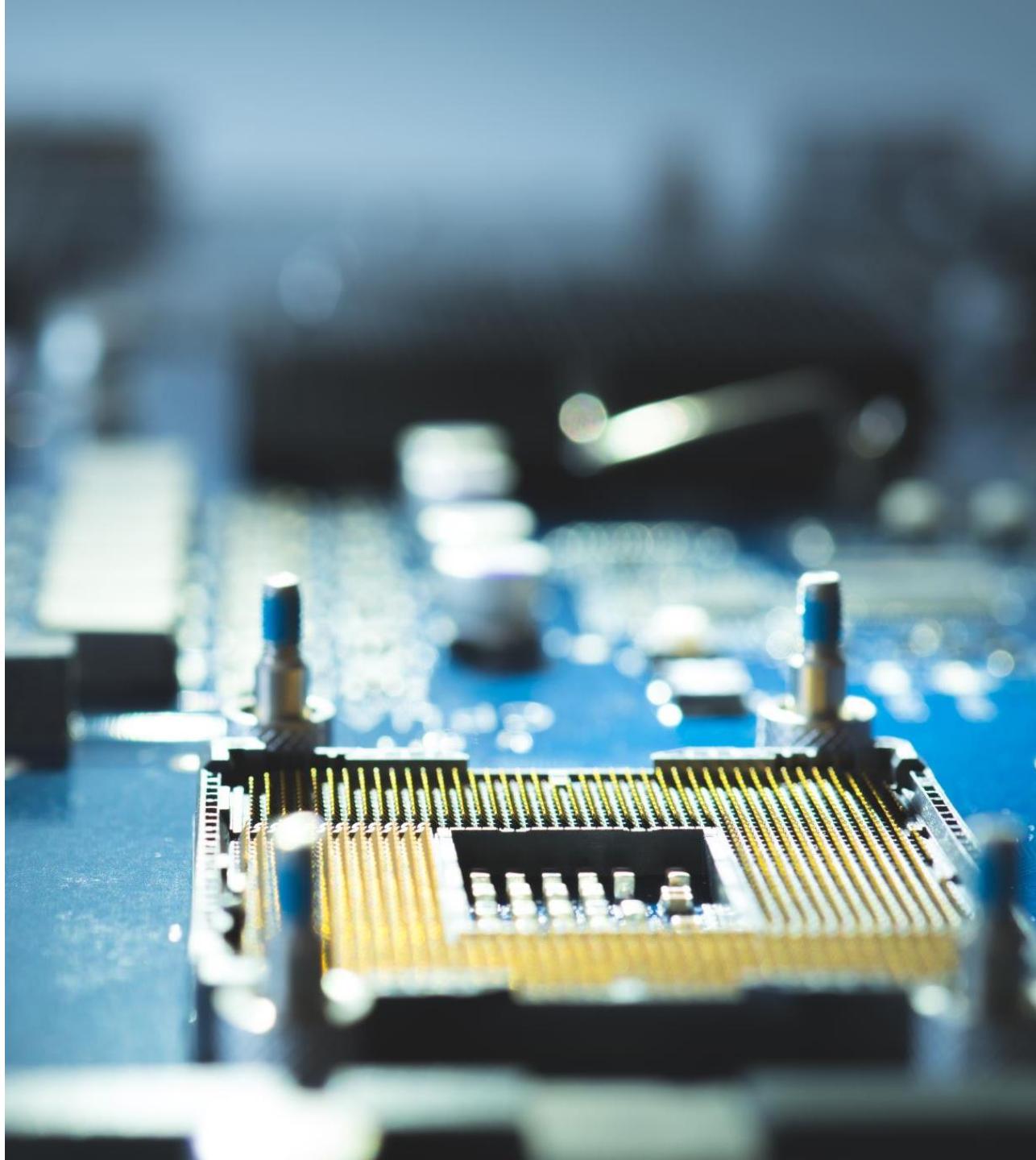
Diseño de Pruebas

- Conocimiento de los Requisitos
- Reducción de la funcionalidad a probar
- Definición de Casos de Prueba
- Relacionar con los Casos de Usuario (historias)
- Pensar bien los costes de las pruebas
- Enfocarse en las funcionalidades más críticas
- Cada prueba requiere su herramienta



HERRAMIENTAS

- Pruebas Unitarias: Junit
- Pruebas de Integración: Mockito
- Pruebas de Aceptación/Funcionalidad Web: Selenium
- Bases de Datos/Web Rendimiento: Jmeter/Blazemeter/Gatling
- Servidor de Integración Contínua: Jenkins/TravisCI/CircleCI
- Herramienta de Seguimiento de Proyectos e Incidencias: Jira/Mantis
- Automatizador de la Compilación, Despliegue y Ejecución de proyectos: Maven/Gradle/NPM
- Despliegue de aplicaciones: Docker/Kubernetes



AUTOMATIZACIÓN DE LAS PRUEBAS





EJEMPLO

```
import org.junit.Test;  
import static org.junit.Assert.assertEquals;  
public class MiClaseTest {  
  
    @Test  
    public void pruebaSuma() {  
        int resultado = MiClase.suma(3, 5);  
        assertEquals(8, resultado);  
    }  
}
```



MENSAJES PERSONALIZADOS

```
assertEquals(2, calculadora.suma(1, 1), "La  
suma debería ser 2");
```

```
assertEquals(2, calculadora.suma(1, 1), () ->  
"La suma debería ser 2")
```

LA ANATOMIA DE JUNIT

```
package examples.nbank;

public class Conversion {

    public double tempConversion (double temperature, String unit) {
        if (unit.equals("F"))
            return (temperature - 32) * (5.0/9.0);
        else
            return (temperature * (9.0/5.0)) + 32;
    }
}
```

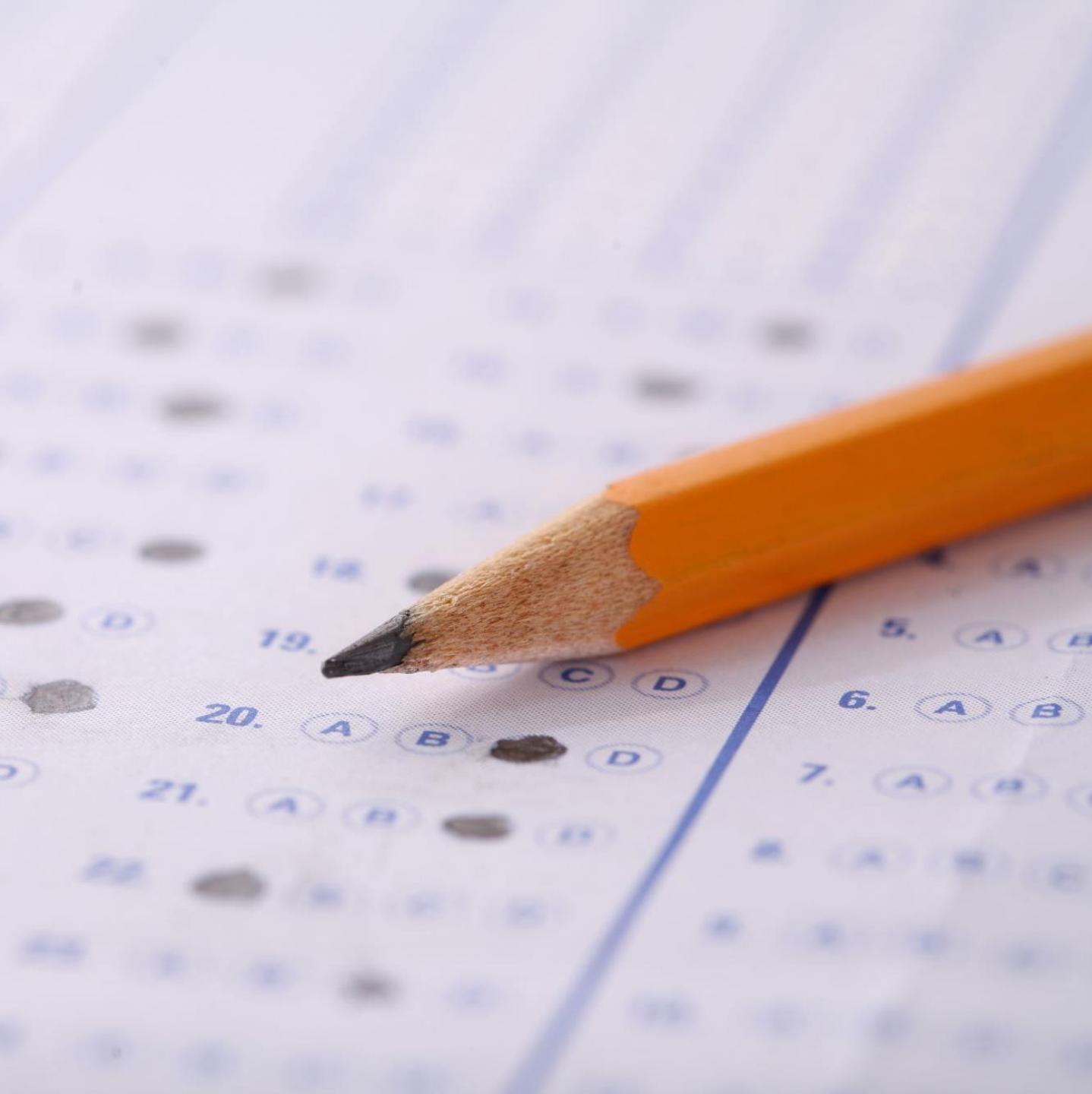
```
package examples.nbank;

1import static org.junit.Assert.assertEquals;
2import org.junit.*;
3public class ConversionTest {

    4@Test
    5public void testTempConversion() throws Throwable {
        // Given
        6Conversion underTest = new Conversion();

        // When
        7double temperature = 80.0d;
        String unit = "";
        8double result = underTest.tempConversion(temperature, unit);

        // Then - assertions for result of method tempConversion(double, String)
        9assertEquals(176.0d, result, 0.0);
    }
}
```



TRABAJAN
DO CON
TEST



EXCEPCIONES CONTROLADAS

- JUnit nos permite comprobar que un método lanza una excepción controlada.
 - Deben extender de Throwable. Por ejemplo RuntimeException.
 - `assertThrows(Exception.class, () -> {});`
 - `assertThrows(Exception.class, () -> {}, message);`

Assert all

- Difícil seguimiento de asserts cuando hay muchos en un test.
- Si un assert falla, no se ejecutan los siguientes y no sabemos su evaluación.
- assertAll ejecuta todos los assert independientemente del posible fallo de uno de ellos.
- Reporta todos los fallos. Dónde se han producido y por qué.



Ejemplo de uso

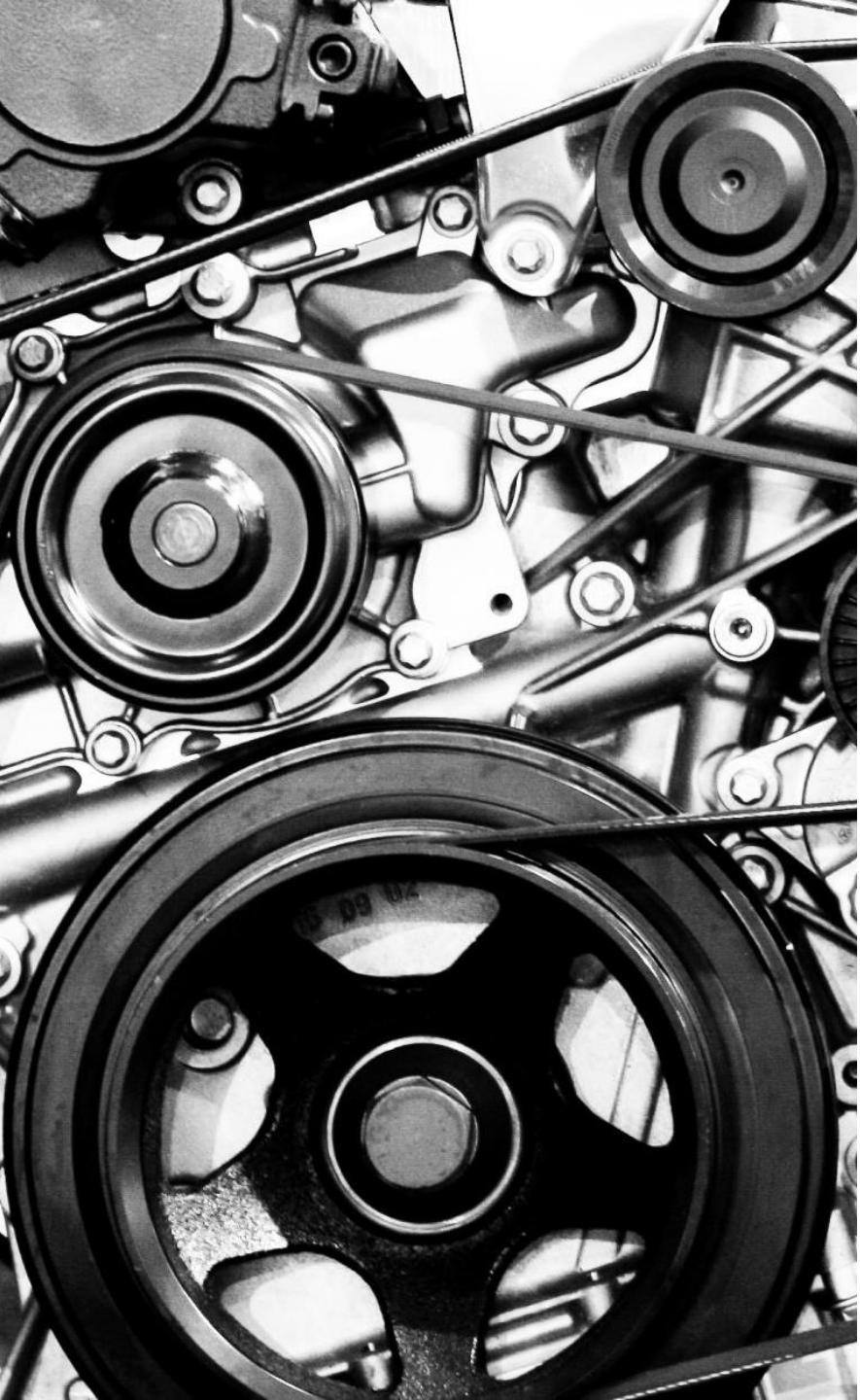
- `assertAll(String message, () -> {});`
- Pueden incluir varias expresiones lambda.
- `assertAll("probando compras", () ->
assertNotNull(store.getProducts()), () ->
assertEquals(2,
store.getProducts().size()), () ->
assertTrue(new
BigDecimal("300.00").compareTo(actual
Amount) == 0));`





Dar nombre a los tests

- JUnit 5 nombra a los tests con el nombre del método.
- Permite especificar un nombre personalizado anotando el método con `@DisplayName("Nombre del test");`
 - `@Test`
 - `@DisplayName("Comprobación suma calculadora")`
 - `void sumaCalculadora() { ... }`



CICLO DE VIDA

- Proceso por el cual se crea, se ejecuta y se destruye una instancia encargada de la realización de las pruebas
- Se encarga el motor de JUnit 5.
- Se crea una nueva instancia con cada test que se ejecuta.
- JUnit 5 permite ejecutar hooks en diferentes momentos del ciclo de vida.
- Hooks de JUnit 5: @BeforeAll / @AfterAll
@BeforeEach/ @AfterEach

@BeforeEach / @AfterEach

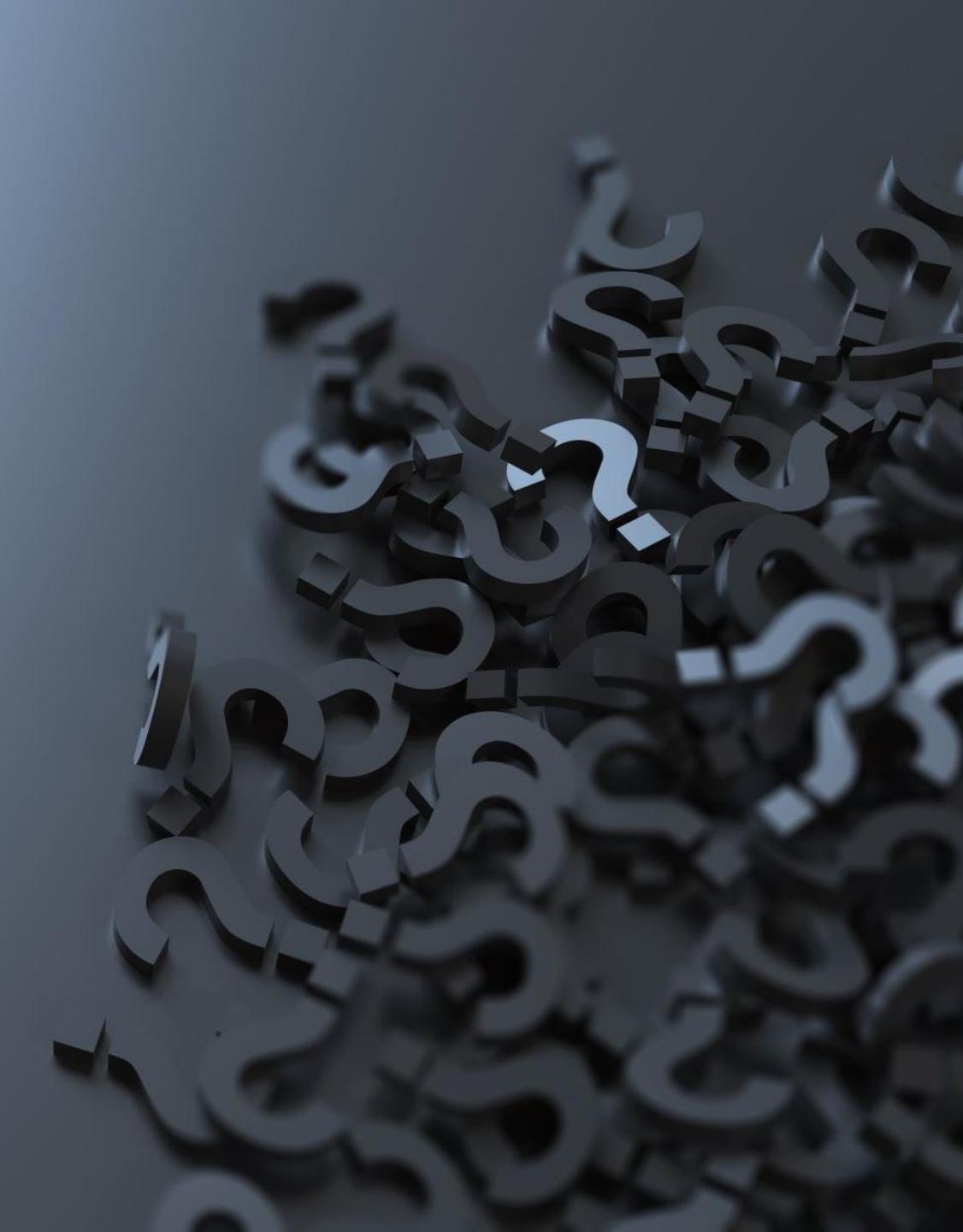
- Se ejecuta una vez que se crea una nueva instancia, es decir, cada vez que se ejecuta un test.
- `@BeforeEach` - Se ejecuta antes de la ejecución del test.
- `@AfterEach` - Se ejecuta después de la ejecución del test.

```
public class TiendaTest {  
    private List<String> products;  
@BeforeEach  
    void setup() {  
        products = Arrays.asList("product1",  
"product2");  
    }  
  
@AfterEach  
    void teardown() {  
        products.clear();  
    }  
}
```



@BeforeAll/@AfterAll

- Se ejecuta antes de crear/después de destruir la instancia, por lo que se implementa en un método estático.
- Si se anota en un método no estático, éste fallará, ya que la instancia no existe.
- Se puede forzar que la instancia sólo se cree una vez, aunque es mala práctica, pues comparte el estado de la clase entre tests.
- @Creando una instancia por clase, nos permite quitar el static a @BeforeAll/@AfterAll



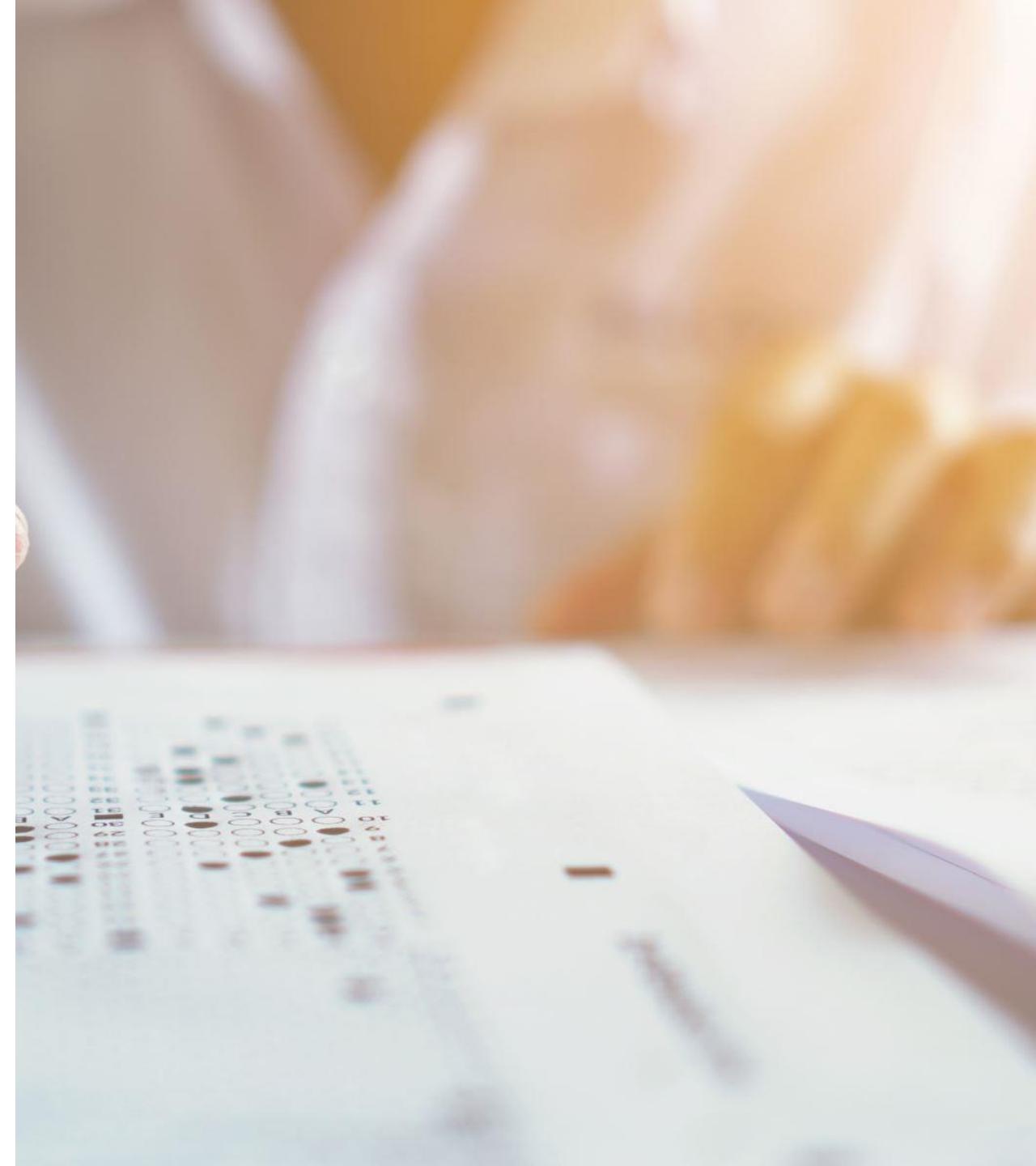
DESHABILITANDO TESTS UNITARIOS

- Los tests pueden deshabilitarse para evitar su ejecución.
- Anotamos el test con `@Disabled` o JUnit 5 recomienda especificar un motivo:
- `@Disabled("Se deshabilita este test hasta que el bug @B54 se resuelva")`

TESTS CONDICIONALES - ANOTACIONES

- Las pruebas unitarias se pueden ejecutar conforme a diferentes condiciones.
- JUnit ofrece anotaciones para habilitar o no dichos tests:
 @EnabledOnOS @EnabledOnJre @EnabledIfSystemProperty
 @EnabledIfEnvironmentVariable

```
@EnabledIf("hasStock")  
  
@Test  
  
void it_should_decrease_stock() {  
}  
  
boolean hasStock(){  
}
```



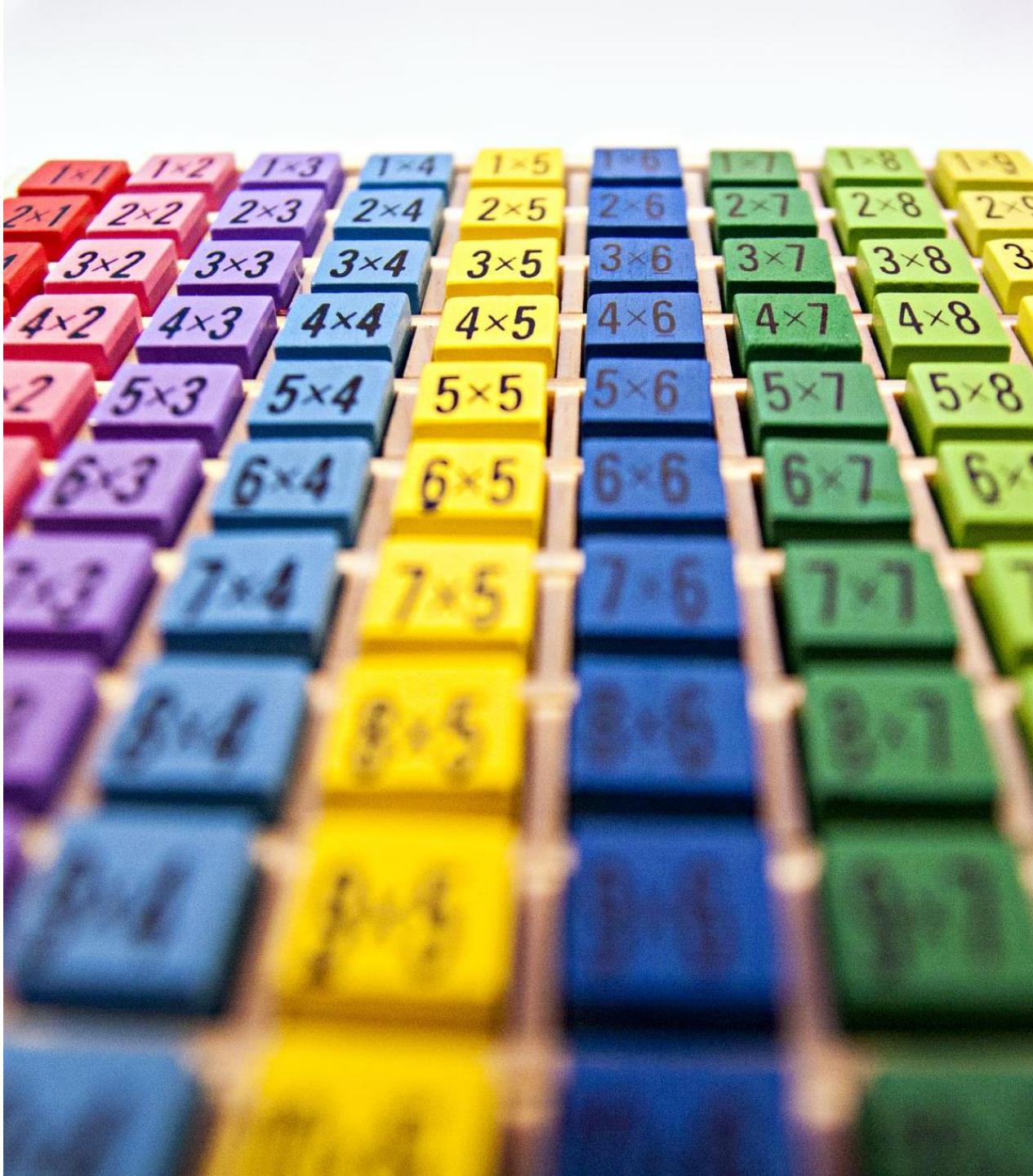
TESTS CONDICIONALES

- Permiten habilitar parte de un test en función de si se cumple una condición. `assumeTrue(boolean condition); assumeFalse(boolean condition);`
- Si la assumption no se cumple, el código a partir de ahí se deshabilita, evitando el fallo
- JUnit 5 permite ejecutar o no parte del método encapsulándolo en una expresión lambda. o `assumingThat(boolean, () -> {});`
- La expresión lambda sólo se va a ejecutar si la expresión evalúa a true.
- El código fuera de la expresión lambda, sí se ejecutará.



Clases anidadas

- Las clases anidadas permiten organizar los tests por diferentes criterios: funcionalidad, condicionalidad... Se anotan las clases con @Nested.
- Se puede incluir una descripción en dichas clases y los métodos que contiene con @DisplayName. Los tests aparecerán en el reporting agrupados por clases. Si falla un test de una clase @Nested, aparecerá como fallo el test y las clases contenedoras del mismo (hasta la clase raíz).



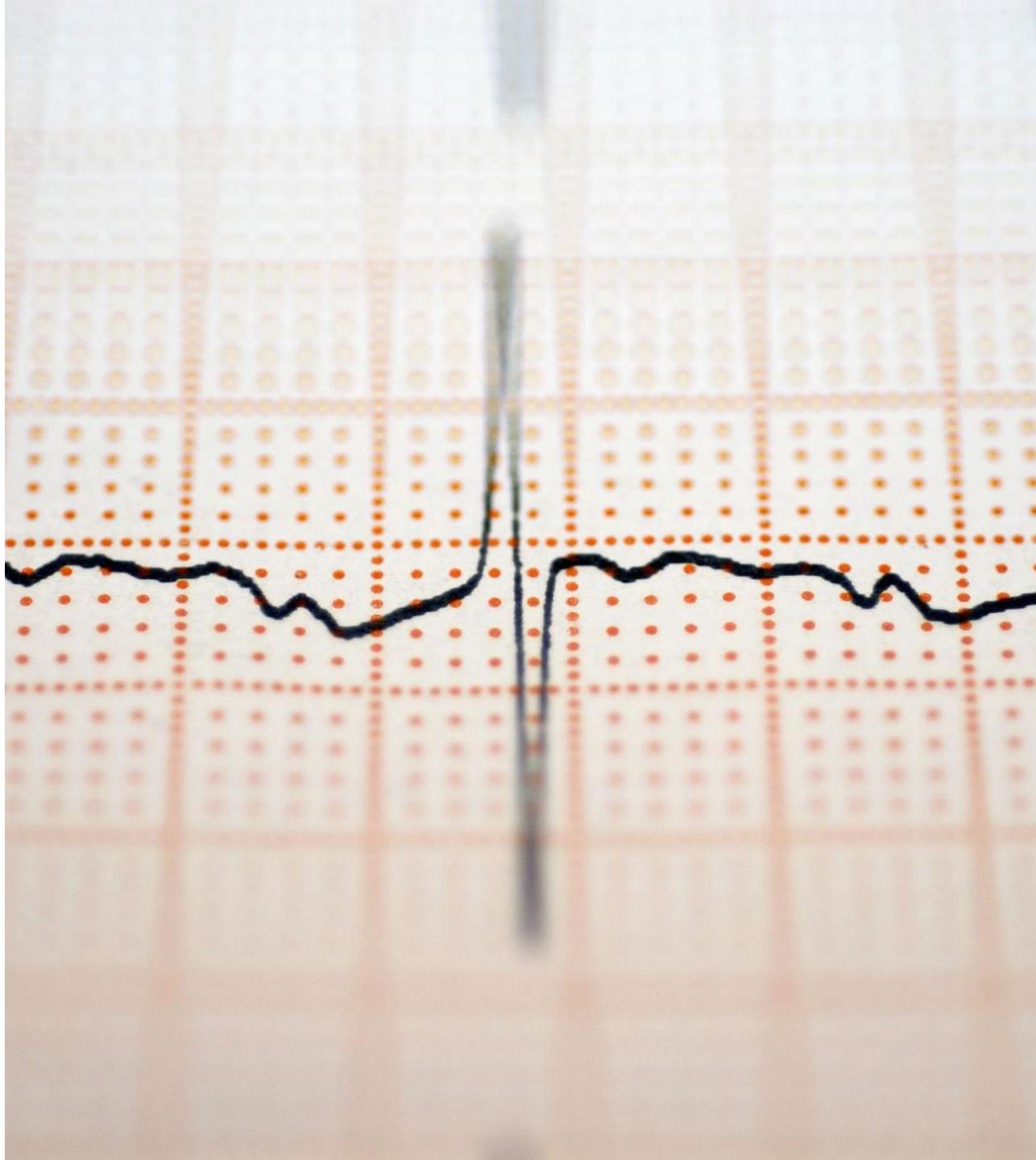
Repetir tests

- JUnit 5 permite ejecutar varias veces el test. Útil en métodos que presentan cierta aleatoriedad. o Por ejemplo, crean valores random. Se anotan con @RepeatedTest(int repetitions)
- En el reporting, aparece la ejecución de las repeticiones. Es personalizable el nombre en cada ejecución del test.
@RepeatedTest(int, message)
- Podemos usar variables para la creación de ese mensaje:
{currentRepetition}{totalRepetitions}
- Podemos combinar el nombre con @DisplayName.
@DisplayName: Será el título principal. @RepeatedTest: Nombre en cada repetición. @DisplayName puede ser injectado en el mensaje de @RepeatedTest: o
{displayName}



TESTS PARAMETRIZADOS

- Otra forma de repetir tests en JUnit 5.
- Permite que en cada repetición se ejecute con datos diferentes. Se inyectan mediante variables en los métodos.
- JUnit 5 permite proporcionar dichos datos mediante:
`@ValueSource(strings={})` Otros tipos: ints, doubles, booleans.
`@CsvSource({índice, valor})`
`@CsvFileSource(resources, delimiter, numLinesToSkip)` `@MethodSource(static methodName)`



Filtrar tests

- Permite ejecutar los tests de forma selectiva.
- Útil para identificar un test con el id de una tarea. Posibilidad de ejecutar grupos de tests.
- Se anota el test o la clase con @Tag. Se puede anotar con varios @Tag.
- Se especifican en el RunConfiguration / Maven.

- 
2. A B C D E
 3. A B C D E
 4. A B C D E
 5. A B C D E
 6. A B C D E
 7. A B C D E
 8. A B C D E
 9. A B C D E
 10. A B C D E
 11. A B C D E
 12. A B C D E
 13. A B C D E
 14. A B C D E
 15. A B C D E
 16. A B C D E
 17. A B C D E
 18. A B C D E
 19. A B C D E
 20. A B C D E
 21. A B C D E
 22. A B C D E
 23. A B C D E
 24. A B C D E
 27. A B C D E
 28. A B C D E
 29. A B C D E
 30. A B C D E
 31. A B C D E
 32. A B C D E
 33. A B C D E
 34. A B C D E
 35. A B C D E
 36. A B C D E
 37. A B C D E
 38. A B C D E
 39. A B C D E
 40. A B C D E
 41. A B C D E
 42. A B C D E
 43. A B C D E
 44. A B C D E
 45. A B C D E
 46. A B C D E
 47. A B C D E
 48. A B C D E
 49. A B C D E