

# Programación Orientada a Objetos (JAVA)

Ciclo Superior DAM – Programación 2018/2019 – Aula Nosa

Marcos Núñez Celeiro

# Entornos de desarrollo (IDEs) y versiones de Java

# Entornos de desarrollo (I)

**Los entornos de desarrollo más habituales para programar en Java son los siguientes:**

- **Eclipse: El IDE más popular para programación en Java.**

- Todas sus características son gratuitas y el entorno es de código abierto.
- Permite instalación de *plugins* para añadir funcionalidades al entorno.
- Mediante *plugins* soporta gran cantidad de lenguajes (*Python*, PHP, Scala, etc).
- Como punto flaco, los *plugins* e instalación de diferentes tecnologías de diferentes empresas tienden a dar diversos problemas en proyectos complejos.

- **NetBeans: Es el IDE oficial para programación en Java.**

- Es de código abierto.
- Permite programar en una gran cantidad de lenguajes (C, Java, PHP...). En algunos casos mediante la instalación de *plugins*. Provee una interfaz gráfica nativa para desarrollo de interfaces gráficas con *swing*.

# Entornos de desarrollo (II)

- **IntelliJ:** de la empresa JetBrains (desarrolladores del lenguaje Kotlin). Este IDE es muy similar a Eclipse y se ha basado en él para su desarrollo.
  - Es el IDE más moderno y probablemente el mejor IDE para programar no solo en Java, sino en casi cualquier lenguaje. Es el IDE oficial para programar para Android (Android Studio funciona sobre IntelliJ).
  - Tiene versión gratuita (*community*) y de pago (*ultimate*). Determinadas bibliotecas o *frameworks* complejos solo son soportados por el *ultimate* ([Consultar diferencias](#)).
  - Tiene *plugins* para trabajar con gran cantidad de lenguajes (Python, PHP, Javascript...).
  - Tiene un diseñador para desarrollar interfaces gráficas con *swing*. En el caso de JavaFX no permite la edición de CSS sin versión *ultimate*.
- **BlueJ:** IDE gratuito enfocado a principiantes y soportado por Oracle. Tiene una interfaz muy simple en comparación con los anteriores. **Permite ver de manera gráfica la jerarquía de clases Java.**

\* Más información: <http://www.mindfiresolutions.com/blog/2018/05/best-java-ides-2018/>

<https://www.javaworld.com/article/3114167/development-tools/choosing-your-java-ide.html?page=2>

# Versiones de JAVA

- Desde Java 9 se ha modificado la política de versiones del JDK de Oracle.
- Una de las desventajas de Java es que se adaptaba muy lentamente a los cambios en el mercado. El lenguaje suele ir atrasado en cuanto a funcionalidades respecto a otros lenguajes.
- Como respuesta a todo esto, desde Java 9 se ha acelerado el proceso de versiones “grandes” de dos por década a dos por año y una versión LTS (*long time support*) cada tres años.
- El JDK de Java 11 (septiembre, 2018) es la primera versión LTS que se distribuye.
- Desde Java 11, aunque el desarrollo con el JDK de Oracle sigue siendo gratuito, **hay que pagar si se desea poner aplicaciones en producción.**
  - Esto no es tan grave como parece. Existen *kits* de desarrollo (JDKs) nativos y gratuitos ([OpenJDK](#)), igual que el JDK de Oracle, compatibles con el estándar de Java.
  - Desarrollar con el JDK de Oracle sigue siendo gratuito, solo se cobran puestas en producción. Puede desarrollarse con el JDK de Oracle y hacer el cambio al *OpenJDK* antes de poner el programa a funcionar.

\* Más información:

<https://www.campusmvp.es/recursos/post/java-11-ya-esta-aqui-te-toca-pagar-a-oracle-o-cambiarte-a-otras-opciones.aspx>

# Programación en Java: Tipos de datos

# Tipos de datos primitivos

<b>byte</b>	<b>8-bit signed</b>	<b>-128 to 127</b>
<b>short</b>	<b>16-bit signed</b>	<b>-32,768 to 32,767</b>
<b>int</b>	<b>32-bit signed</b>	<b>-2<sup>31</sup> to 2<sup>31</sup> - 1</b>
<b>long</b>	<b>64-bit signed</b>	<b>-2<sup>63</sup> to 2<sup>63</sup> - 1</b>
<b>char</b>	<b>16-bit Unicode</b>	<b>0 to 2<sup>15</sup> - 1</b>
<b>double</b>	<b>64-bit IEEE 754 floating point</b>	<b>-2<sup>1074</sup> to 2<sup>1074</sup></b>
<b>float</b>	<b>32-bit IEEE 754 floating point</b>	

Imagen obtenida de artículo:

<https://medium.com/@madhupathy/a-beginners-guide-to-java-part-1-of-3-33edf47e47b4>

\* Se recomienda leer el artículo entero más adelante (una vez se haya impartido la mitad del temario)

# Tipos de datos

A continuación se muestran ejemplos de declaraciones de variables y constantes en Java con distintos tipos.

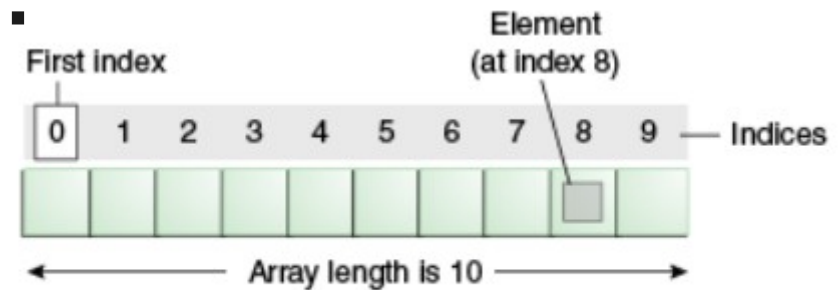
¿Hay un tipo de dato primitivo aquí? ¿Y no primitivo? ¿Por qué están fallando las líneas 14 y 15? Responde a las preguntas por orden.

```
5     private static long id;
6     private static final String SALUDO = "Hola";    // Atributo, estático, constante, string
7
8     public static void main(String[] args) {
9         final int entero = 4;
10        final String nombre = "Mauro";
11
12        byte numPequenho = -12;    // byte solo puede tener valores desde -128 a 127
13        byte numPequenho2 = -128;
14        byte numPequenho3 = -129;    // ERROR
15        byte numPequenho4 = 128;    // ERROR
16
17        short numcorto = 12981;
18
19        float real = 121.38294f;    // Por defecto con el "." lo coge como double. Se pone una f para float
20        double real2 = 121.38294;
21    }
22
23 }
```



# Tipos de datos (*arrays*)

- Un *array* permite almacenar conjuntos de elementos de un tipo concreto. Existen formas diversas de declarar un *array*:



An array of 10 elements.

```
// Inicializacion array sin datos  
char descripcion[] = new char[200];
```

```
// Inicializacion de arrays con datos  
int [] multiplosDe2 = { 2, 4, 6, 8 };  
int multiplosDe3[] = { 1, 3, 6, 9 };
```

Índice 0: 2 ==> multiplosDe2[0] = 2.  
Índice 1: 4 ==> multiplosDe2[1] = 4.  
Índice 2: 6 ==> multiplosDe2[2] = 6.  
Índice 3: 8 ==> multiplosDe2[3] = 8.

```
// Rellenando arrays  
double [] numerosReales = new double[400];  
numerosReales[0] = 1.5;  
numerosReales[1] = 200.23921;
```

En este ejemplo se almacena el valor 1.5 en la posición 0 y el 200.23921 en la posición 1.

\* Leer documentación oficial: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html>

# Tipos de datos (*arrays*)

Para acceder a un elemento de un *array* se utilizan los corchetes [].

Ejemplos:

- `primos[7] = 4` // almacena un 4 en la posición 7 del *array* "primos".
- `primos[8] = 2328` // almacena un 2328 en la posición 8 del *array* "primos".

En el código escrito a continuación se almacenan números en un *array* utilizando un bucle.

```
// Inicializando un array de numeros con un bucle
int numeros[] = new int[200]; // Reservo memoria para 200 valores
int numAIntroducir = 6;

for (int i = 0; i <= 6; i++) {
    numeros[i] = numAIntroducir--;
}
```

**Ejercicio:** Dibuja como quedaría el *array* al terminar de introducir los números (guiandote por la imagen de la diapositiva anterior).

# Ejercicio

## 1. ¿Qué hace el código que se muestra a continuación?

```
// Mostrando numeros almacenados (ahora con un while)
int i = 0;
while (i <= 6) {
    System.out.print(numeros[i++] + " ");
}

System.out.println(); // Salto de linea

// Mostrando numeros almacenados en posicion par
i = 0;
while (i <= 6) {
    if (i % 2 == 0) System.out.print(numeros[i] + " ");

    i++;
}
```

## 2. Crea un *array* y haz que, tenga los números que tenga almacenados, solo muestre los pares.

# Variables y constantes

## Definición de una variable:

```
double salarioEmpleado = 753.30;
```

## Definición de la misma como una constante (modificador *final*):

```
final double salarioEmpleado = 857.72;
```

\* En Java la convención es que las variables se pongan en formato camelCase.

# Tipos de datos (enumerados)

## Enumerados:

```
2 enum DiaSemana {
3     LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
4 }
5
6 public class Principal {
7     public static void main(String[] args) {
8
9         DiaSemana dia = DiaSemana.LUNES;
10
11         System.out.println("Hoy es: " + dia.toString());
12     }
13 }
```

# Tipos de datos (cadenas de caracteres)

## Cadenas de caracteres (strings): clase "String" de Java.

```
2 public class MainCadenas {
3
4     public static void main(String[] args) {
5         String nombre1;           // Por defecto tiene valor nulo (NULL)
6         nombre1 = "James Gosling";
7         String nombre2 = "James Gosling";
8
9         if (nombre1.equals(nombre2)) {
10            System.out.println("Los nombres son iguales.");
11        } else {
12            System.out.println("Los nombres son diferentes.");
13        }
14    }
15
16 }
```

\* La comparación con == compara direcciones de memoria. Es necesario utilizar equals.

# Tipos de datos (cadenas de caracteres)

## Otros métodos definidos en la clase String de Java:

```
char caracter = nombre1.charAt(5); // Devuelve el caracter de la posición 5  
int longitud = nombre1.length(); // Longitud de una cadena
```

```
String nombre3 = nombre1.concat(nombre2); // Concatena strings  
String s = String.valueOf(4); // Convierte un valor a tipo String  
nombre3 = nombre1.toLowerCase(); // Devuelve nombre1 en minúsculas  
nombre3 = nombre1.toUpperCase(); // Devuelve nombre1 en mayúsculas.  
nombre3 = nombre1.substring(1, 4);
```

# Tipos de datos (cadenas de caracteres)

## Probamos a sacar los métodos de la clase String por pantalla:

```
System.out.println("Caracter de la pos 3: " + nombre1.charAt(3));  
System.out.println("Concatenación n1 y n2: " + nombre1.concat(nombre2));  
System.out.println("Longitud string (length): " + nombre1.length());  
System.out.println("Convertir valor a String (valueOf): " + String.valueOf(4));  
System.out.println("Convertir a minúscula: " + nombre1.toLowerCase());  
System.out.println("Convertir a mayúscula: " + nombre1.toUpperCase());  
System.out.println("Substring de pos 1 a 4: " + nombre1.substring(1, 4));
```

**Salida:** Los nombres son iguales.  
Caracter de la pos 3: e  
Concatenación n1 y n2: James GoslingJames Gosling  
Longitud string (length): 13  
Convertir valor a String (valueOf): 4  
Convertir a minúscula: james gosling  
Convertir a mayúscula: JAMES GOSLING  
Substring de pos 1 a 4: ame



# Tipos de datos: preguntas

- **¿Qué diferencias notas entre los tipos de datos *int*, *char*, *float*, *double*, *char[]*, etc respecto al tipo de dato *String*? ¿Por qué crees que es?**
- **En los ejemplos anteriores se declara una variable de tipo *String* y después se utiliza el operador “.” para acceder a una serie de métodos o funciones.**  
**¿Crees que es posible hacer esto con un *int* o un *double*? Inténtalo hacerlo y razona por qué es o no es posible.**

# **Tipos de datos no primitivos: Clases y objetos**

# Clases y objetos: conceptos básicos

Puedes imaginarte una clase como una plantilla. Imagina que cubres una plantilla con tus datos para inscribirte en un puesto de trabajo o que simplemente rellenas los datos en un formulario en internet para registrarte en una página.

Esos datos normalmente son: nombre, apellidos, fecha de nacimiento, gustos, aficiones, etc.

A nivel de programación, es habitual representar esos datos encapsulados en un solo elemento (este elemento se denomina clase, en este ejemplo podría ser la clase “Usuario”).

Si 10 personas rellenan el formulario tendríamos 10 usuarios con distintos datos (distintos nombres, distintos apellidos, etc). Se puede decir que tenemos 10 objetos de tipo Usuario.

Cada elemento de la clase Usuario se denomina un objeto. Al igual que declaramos una variable *x* de tipo *int* y valor 4 (*int x = 4;*) también podemos declarar lo siguiente:

```
Usuario u = new Usuario("Pedro");
```

Si conoces los *structs* en C, una clase es algo similar. En la línea anterior se ha creado una variable de nombre “u” y de tipo “Usuario” y le he asignado (con el operador “=”) un nuevo objeto “Usuario”.

# Características de una clase (I)

- **Una clase es un tipo de dato “complejo” o “no primitivo”. Es decir, es un tipo de dato que contiene otros dentro. Por ejemplo, una clase Alumno podría tener un nombre (de tipo String), unos apellidos (de tipo String) y una edad (de tipo int). Esta es la razón de que se diga que es complejo o que no es un tipo primitivo. Un tipo complejo se compone de otros datos.**
- **Los miembros de una clase son: los atributos, los métodos y los constructores.**

# Características de una clase (I)

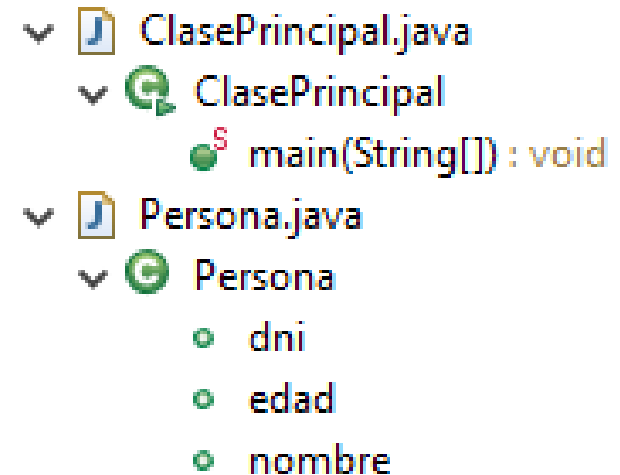
- **Atributos**: definen las características de los objetos. Por ejemplo: los atributos de un ordenador podrían ser su procesador, su memoria y su capacidad de almacenamiento. A su vez, el procesador se podría definir (si se quiere) como otra clase.
- **Métodos**: definen el comportamiento de los objetos. Un método es como se denomina a las funciones que están dentro de una clase.
- **Constructores**: es una manera de inicializar un objeto de una clase concreta. Por ejemplo, si creo un Usuario nuevo puedo permitir que todos sus atributos (nombre, apellidos, edad...) estén vacíos inicialmente o puedo forzar a que sea obligatorio darle unos valores desde el inicio. Para llamar a un constructor se utiliza la palabra clave *new* (consulta el ejemplo expuesto en las diapositivas anteriores).

# Clases en Java: atributos (I)

## • Ejemplo con la definición de dos clases:

- Persona (define la plantilla sobre la que se crean diferentes personas)
- ClasePrincipal (es la clase que tendrá el método o función *main*). Recuerda que en un programa, el método *main* es el punto de entrada al mismo. Es decir, cuando se ejecuta un programa se ejecuta su *main*.

**En este caso, la clase Persona es muy simple. Si conoces los *structs* de C, verás que hasta aquí es lo mismo. En posteriores diapositivas se ampliará mucho este concepto.**



```

3 public class Persona {
4     public String dni;
5     public String nombre;
6     public int edad;
7 }
```

# Clases en Java: atributos (II)

A continuación, se presenta el método *main* situado en la clase “ClasePrincipal”.

```
public static void main(String[] args) {  
    // Creando una persona  
    Persona p1 = new Persona();  
    p1.dni = "82749388M";  
    p1.nombre = "Richard";  
    p1.edad = 45;  
  
    // Creando una segunda persona  
    Persona p2 = new Persona();  
  
    System.out.println("La primera persona se llama: " + p1.nombre);  
    System.out.println("La segunda persona se llama: " + p2.nombre);  
}
```

# Ejercicio

**Escribe el código de las clases anteriores utilizando tu entorno de desarrollo (IDE):**

- 1) ¿Que datos crees que sacarán por pantalla las dos últimas líneas?**
- 2) Los atributos de la clase “Persona” se han definido como “public”. Prueba a cambiar *public* por *private* y por *protected* y comprueba lo que ocurre. Intenta explicarlo.**
- 3) Divide la declaración y la asignación de la Persona p1 en dos líneas. En la primera línea declara la persona y en la segunda asígnale el nuevo objeto.**
- 4) Utiliza el *debugger* de tu IDE para depurar el programa.**
  - 1) ¿Qué valor tienen p1 y sus atributos después de ser declarada pero antes de llamar a su constructor (*de hacer el new*)?
  - 2) ¿Qué valor tienen p1 y sus atributos después de asignarle un nuevo objeto de la clase Persona?
  - 3) ¿Qué ocurre si, después de crear p1 y antes de llamar a su constructor, intentas mostrar por pantalla p1.nombre? ¿Por qué crees que ocurre esto?



# Clases en Java: constructores (I)

En el momento en que se crea un nuevo objeto (*new*), lo que realmente se está haciendo es una llamada a su constructor.

A continuación se muestra la clase `Persona` con un constructor vacío:

```
3 public class Persona {  
4     public String dni;  
5     public String nombre;  
6     public int edad;  
7  
8     public Persona() {  
9  
10 }
```

.....  
`Persona p2 = new Persona();`



# Clases en Java: constructores (II)

Si pruebas a crear el constructor vacío verás que en realidad no se produce ningún cambio real en el programa. Esto es porque si no hay ningún constructor creado en una clase, Java ya proporciona un constructor vacío por defecto.

```
3 public class Persona {
4     public String dni;
5     public String nombre;
6     public int edad;
7
8     public Persona() {
9
10 }
```

```
Persona p2 = new Persona();
```



# Clases en Java: constructores (III)

```
3 public class Persona {
4     public String dni;
5     public String nombre;
6     public int edad;
7
8     public Persona() {
9         System.out.println("Creando nueva persona...");
10    }
```

Se dice que un constructor es vacío cuando no recibe parámetros.

Ejercicio: Intenta añadir un mensaje al constructor para comprobar que la llamada se realiza correctamente.

```
Persona p2 = new Persona();
```

# Clases en Java: constructores (IV)

**En Java se llama constructor vacío a uno que no recibe nada como parámetro. Un constructor, en cambio, se denomina parametrizado cuando sí que recibe parámetros. Ejemplo:**

```
public Persona(String nuevoDni) {  
    dni = nuevoDni;  
}
```

**En este caso, el constructor de Persona recibe un parámetro de tipo “String”.**

- **¿Qué ocurre si copias este constructor y comentas el anterior? ¿Por qué crees que ocurre? Investiga en internet sobre el funcionamiento de los constructores en Java para responder a estas preguntas.**
- **¿Como crearías un nuevo objeto (p3) utilizando este constructor? Intenta hacerlo.**

# Clases en Java: constructores (V)

```
3 public class Persona {
4     public String dni;
5     public String nombre;
6     public int edad;
7
8     public Persona() {
9         System.out.println("Creando nueva persona...");
10    }
11
12    public Persona(String nuevoDni) {
13        dni = nuevoDni;
14    }
15 }
```

```
Persona p2 = new Persona(); // Llamada al constructor vacío
Persona p3 = new Persona("33445598S"); // Llamada al constructor parametrizado
```

# Clases en Java: constructores (VI)

## Observaciones:

- Un constructor siempre se llama igual que el nombre de la clase.
- Los constructores se parecen mucho a un método. Tienen un nombre, un modificador de acceso y unos parámetros.
- La diferencia entre un constructor y un método normal es que los constructores no tienen tipo de dato devuelto.
- En la última diapositiva se ha dejado la clase “Persona” con dos constructores con el mismo nombre y distintos parámetros (es decir, para llamarlos son diferenciables por los parámetros que reciben). A esto se le llama sobrecarga de constructores. Se puede hacer lo mismo con los métodos normales (sobrecarga de métodos o funciones).

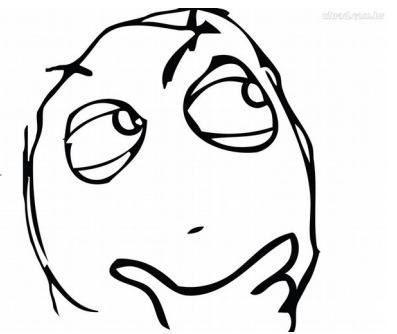
# Clases en Java: métodos (I)

**Se le llama método a una función situada dentro de una clase. A continuación, se muestra un ejemplo de un método que suma años a la edad de una Persona:**

```
public void cumpleAños() {  
    edad++; // edad = edad + 1  
}
```

**Ahora, un método que permita cambiar el dni de una persona:**

```
public void cambiarDni(String dni) {  
    this.dni = dni;  
}
```



# Ejercicios

- 1) Añade a la clase Persona el método creado anteriormente (cumpleAños()) y utilízalo desde el *main*. Muestra los datos de la persona p1 y muestra su edad antes y después de llamar al método.**
- 2) El método “cumpleAños” anterior suma un año a la edad actual de una persona. Crea un nuevo método “cumpleAños” que permita sumar los años que se deseen (tiene que recibir un parámetro indicando los años que se quieren sumar). Utilízalo de nuevo desde el main en p1.**

**Concepto: cuando termines los dos ejercicios anteriores tendrás en la clase “Persona” dos métodos con el mismo nombre y distintos parámetros. Esto se llama sobrecargar un método (sobrecarga de métodos).**



# Ejercicios

## 3) Crea una clase Dirección con los siguientes atributos:

- Código postal
- Ciudad
- Calle
- Numero
- Puerta (char)

**Posteriormente, haz que la clase Persona tenga una dirección.**

# Ejercicios

## 4) Crea un nuevo proyecto y copia en él las clases “Persona” y “Direccion”. En este proyecto debes:

- Pedir al usuario que introduzca todos sus datos (también su dirección).
- Una vez pedidos e introducidos, deben mostrarse todos los datos del usuario línea por línea.
- Finalmente, crea un método en la clase Persona (puedes llamarle “indicarMayorDeEdad”) que escriba que es mayor de edad si supera los 18 años o que es menor en caso contrario.

# Clases en Java: palabra clave *this* (I)

Si te fijas, añadiendo el código de los métodos anteriores se pasa a tener dos variables con el mismo nombre: dni.

El primer dni que se declara es un atributo del objeto, el segundo es una variable local al método “cambiarDni”.

```
3 public class Persona {
4     public String dni;
5     public String nombre;
6     public int edad;
7
8     ...
9
10    ...
11
12    ...
13
14    ...
15
16    ...
17
18    ...
19
20    public void cambiarDni(String dni) {
21        this.dni = dni;
22    }
23 }
```

**Atributo de la clase**

**Variable local al método o función**

# Clases en Java: palabra clave *this* (II)

**Si quieres igualar el atributo dni de la persona al nuevo dni pasado como parámetro, según la lógica y los conocimientos impartidos hasta ahora habría que escribir:**

```
dni = dni;
```

Esto obviamente es confuso. En este caso, ambas variables se refieren a la variable local y no al atributo.

**La palabra clave this referencia al objeto actual.** En caso de utilizar dos variables con el mismo nombre es necesario referirse al atributo con this.

# Clases en Java: Modificadores de acceso (I)

En Java existen varios modificadores de acceso.

- Privado (*private*): accesible desde la misma clase.
- Protegido (*protected*): accesible desde la misma clase, las clases que heredan de ella y las que están en su mismo paquete (*package*).
- Público (*public*): accesible desde cualquier parte.

Las clases, atributos y métodos de las clases Java pueden tener un modificador de acceso asociado (en el ejemplo anterior se ha jugado con ellos en los atributos de la clase *Persona*). También pueden no tener modificador de acceso (lo que significaría que el elemento solo puede ser accedido desde el mismo paquete).

# Clases en Java: Modificadores de acceso (II)

Access Modifier	Same Package			Different Package	
	Class	Sub Class	Other Class	Sub Class	Other Class
no modifier	YES	YES	YES		
public	YES	YES	YES	YES	YES
protected	YES	YES	YES	YES	
private	YES				

Imagen obtenida de artículo:

<https://medium.com/@madhupathy/a-beginners-guide-to-java-part-1-of-3-33edf47e47b4>

# Ejercicios

- 1) ¿Qué tendrías que añadir al constructor de la clase Persona si quisieras convertirla en un método?**
- 2) Crea un nuevo proyecto y copia en él las clases que has creado (“Persona”, “Direccion” y “ClasePrincipal”). En este nuevo proyecto haz que todos los atributos de las clases Persona y Direccion sean privados.**
  - ¿Qué ocurre en el programa al hacer esto? ¿Hay errores?
  - ¿Se te ocurre alguna manera de solucionarlos sin volver a hacer públicos los atributos?
- 3) Deja los atributos de la clase Persona y Direccion como privados y haz que solo puedan obtenerse o modificarse sus valores mediante métodos (puedes crear en la clase “Persona” tantos métodos como consideres necesario).**
- 4) Crea un nuevo objeto “Persona” de nombre p4 que te represente a ti (tus datos) y muéstralos con un mensaje. Después, cambia tu nombre y tus apellidos a mayúscula, redúctete dos años y muéstralos de nuevo.**

# Clases y objetos en profundidad



# Clases en Java

**Redefinición de la clase “Persona” con tres atributos, dos constructores y una serie de métodos:**

```
1 package clases.persona;
2
3 public class Persona {
4     // Los atributos privados no son accesibles desde fuera de la clase
5     private String dni;
6     private String nombre;
7     private int edad;
8
9     // Constructor vacío, permite crear instancias de esta clase (es decir, objetos)
10    // Para crear un objeto se utiliza new Persona();
11    public Persona() {
12    }
13
14    // Constructor parametrizado, permite crear instancias de esta clase (es decir, objetos)
15    // Para crear un objeto se utiliza new Persona(String, String, int);
16    public Persona(String dni, String nombre, int edad) {
17        this.dni = dni;
18        this.nombre = nombre;
19        this.edad = edad;
20    }
21
22    public String getDni() {
23        return dni;
24    }
25
26    public void setDni(String dni) {
27        this.dni = dni;
28    }
29
30    public String getNombre() {
31        return nombre;
32    }
33
34    public void setNombre(String nombre) {
35        this.nombre = nombre;
36    }
37
38    public int getEdad() {
39        return edad;
40    }
41
42    public void setEdad(int edad) {
43        this.edad = edad;
44    }
45
46    @Override
47    public String toString() {
48        String toret = "Los datos de la persona son: " +
49            "\nDNI: " + dni +
50            "\nNombre: " + nombre +
51            "\nEdad: " + edad + "\n";
52
53        return toret;
54    }
55 }
```

# Clases en Java (por partes)

Las clases se definen de la siguiente manera:

```
public class Persona { }
```

**Atributos de una clase (suelen llevar un modificador de acceso que puede ser *public*, *private* o *protected*):**

```
// Los atributos privados no son accesibles desde fuera de la clase
private String dni;
private String nombre;
private int edad;
```

**Constructores (constructor vacío)**

```
// Constructor vacío, permite crear instancias de esta clase (es decir, objetos)
// Para crear un objeto se utiliza new Persona();
public Persona() {
}
```

# Clases en Java

## Constructor (parametrizado o con parámetros):

```
// Constructor parametrizado, permite crear
// instancias de esta clase (es decir, objetos).
// Para crear un objeto se utiliza new Persona(String, String, int)
public Persona(String dni, String nombre, int edad) {
    this.dni = dni;
    this.nombre = nombre;
    this.edad = edad;
}
```

**Getters y Setters:** los atributos suelen ponerse privados (private). Para obtener o modificar estos atributos se utilizan métodos. Una manera de modificar el dni sería con el método modificarDni(String nuevoDni) y para obtenerlo se podría crear el método obtenerDni(). A estos métodos, en vez de modificar y obtener suele llamárseles, por convención get y set (getDni y setDni).

```
public String getDni() {
    return dni;
}

public void setDni(String dni) {
    this.dni = dni;
}
```

# Clases en Java: método toString()

**Método toString:** `@Override`  
`public String toString() {`  
    `String toret = "Los datos de la persona son: " +`  
        `"\nDNI: " + dni +`  
        `"\nNombre: " + nombre +`  
        `"\nEdad: " + edad + "\n";`  
  
    `return toret;`  
`}`

```
public class Main {  
    public static void main(String[] args) {  
        // Creando personas  
        Persona p1 = new Persona();  
  
        p1.setDni("82749388M");  
        p1.setNombre("Richard");  
        p1.setEdad(45);  
  
        System.out.println(p1);  
    }  
}
```

Recordemos que **dni, nombre y edad** de la clase **Persona** **son private** (no puede accederse a ellos desde fuera). Para cambiarlos utilizamos **los métodos set** que hemos creado.

---

```
Los datos de la persona son:  
DNI: 82749388M  
Nombre: Richard  
Edad: 45
```

# Clases en Java: método equals(...)

## Método equals

```
// Permite comprobar si dos objetos son iguales.
@Override
public boolean equals(Object obj) {
    if (obj == null || dni == null) return false;

    Persona p = (Persona)obj;
    return dni.equals(p.getDni());
}
```

## En el *main* hacemos la prueba:

```
// Método equals
Persona p2 = new Persona("77629180L", "Ana", 33);
System.out.println("p1 es igual a p2? " + p1.equals(p2));
```

## Resultado:

```
p1 es igual a p2? false
```

# Clases en Java: Herencia (I)

## Cambiamos los modificadores de acceso de Persona:

```
public class Persona {  
    // Cambiamos los atributos a protected, de esta manera  
    // son accesibles desde sus descendientes  
    protected String dni;  
    protected String nombre;  
    protected int edad;
```

## Creamos la clase Empleado que extiende (es hija) de Persona:

```
public class Empleado extends Persona {}
```

# Clases en Java: Herencia (II)

Añadimos los atributos nuevos (salario) y creamos su constructor:

```
public class Empleado extends Persona {  
    private double salario;  
  
    // Tenemos que crear los constructores  
    public Empleado(String dni, String nombre, int edad, double salario) {  
        // Podemos acceder a los atributos porque es protected  
        // y extendemos de Persona  
        this.dni = dni;  
        this.nombre = nombre;  
        this.edad = edad;  
        this.salario = salario;  
    }  
  
    public Empleado(String dni, String nombre, int edad, double salario) {  
        // Podemos acceder a los atributos porque es protected  
        // y extendemos de Persona  
        super(dni, nombre, edad);  
        this.salario = salario;  
    }  
}
```

Hay código repetido (ya igualamos dni, nombre y edad en la superclase). Podemos evitar igualar todo nuevamente llamando al constructor de la superclase (mediante la palabra clave **super**):

```
public Persona(String dni, String nombre, int edad)
```

# Clases en Java: Herencia (III)

**Nos faltan los getters, setters y el método toString con el nuevo atributo “salario”. No necesitamos *equals* porque comparamos solamente el DNI.**

```
public double getSalario() {
    return salario;
}

public void setSalario(double salario) {
    this.salario = salario;
}

@Override
public String toString() {
    String toret = super.toString() + "Salario: " + salario + "\n";

    return toret;
}
```



# Clases en Java: Herencia (IV)

**Creamos un nuevo empleado en el método *main* con el constructor parametrizado de la clase Empleado:**

```
1 package clases.herencia.empleados;
2
3 public class MainHerencia {
4     public static void main(String[] args) {
5         Empleado e = new Empleado("44112229U", "Ana", 40, 800.00);
6
7         System.out.println(e.toString());
8     }
9 }
```

**La salida del método *toString()* es la siguiente:**

```
Los datos de la persona son:
DNI: 44112229U
Nombre: Ana
Edad: 40
Salario: 800.0
```

# Clases en Java: Herencia múltiple (V)

Hay lenguajes (por ejemplo C++) que permiten lo que se llama herencia múltiple. En Java no es posible que una clase extienda a dos clases a la vez, es decir, en Java no está permitida la herencia múltiple.

~~class Persona extends A, B~~ no es válido.

# Documentación adicional

**Lee la documentación sobre clases y objetos  
aportada en la siguiente página:**

**<https://www.programarya.com/Cursos/Java/Objetos-y-Clases>**

# Preguntas (1)

- **¿Cómo declaras una cadena de caracteres en Java?**
- **Si se desea declarar una constante, que palabra clave debe añadirse en la declaración de la misma?**
- **¿Qué es una clase? ¿Y un objeto?**
- **¿En que tres partes podrías dividir una clase?  
Explícalas**
- **¿Cómo “llamas” al constructor de una clase?**
- **¿Para que sirve la herencia? ¿Se te ocurre algún ejemplo?**

## Preguntas (2)

**¿Cómo declaras una cadena de caracteres en Java?**

- **String cadena;**

**Si se desea declarar una constante, que palabra clave debe añadirse en la declaración de la misma?**

- **La palabra clave es “final”. Por ejemplo: final int valor = 4;**

# Preguntas (3)

## ¿Qué es una clase? ¿Y un objeto?

- **Una clase es una plantilla sobre la que se crean objetos. Por ejemplo, en una aplicación de un concesionario se podría guardar el catálogo de coches disponibles. La clase “Coche” sería una plantilla (un tipo de dato) y cada coche diferente que almaceno es un “objeto” de tipo “Coche”.**
- **Por ejemplo: Coche c; // Coche es el tipo de dato y c es un objeto de tipo Coche**

## ¿En que tres partes podrías dividir una clase? Explicálas

**Atributos, constructores y métodos. Los atributos son características que definen a los objetos (un coche puede tener un color, un número de puertas y un modelo). Se llama métodos a las funciones que están dentro de las clases. Por último, los constructores son métodos que no devuelven nada y tienen el mismo nombre que la clase. Se llaman mediante `new NombreClase(parámetros)`.**

# Preguntas (4)

## ¿Cómo “llamas” al constructor de una clase?

- **Mediante la palabra clave *new*. Utilizando el ejemplo anterior, podría llamarse al constructor vacío de Coche de la siguiente manera:**
  - `Coche c = new Coche();`
- **Si el constructor recibe como parámetro de tipo `String` con la marca del coche (por ejemplo), se llamaría así:**
  - `Coche c = new Coche("Opel");`

## ¿Para que sirve la herencia? ¿Se te ocurre algún ejemplo?

Se puede hacer que una clase herede de otra mediante la palabra clave *extends*. La herencia sirve para que las clases hijas puedan compartir atributos, métodos y constructores con las clases padre. Esto permite reutilizar gran cantidad de código (revisar ejemplo anterior de Empleado con Persona).

Otro ejemplo podría ser una clase “Vehículo” y sus subclases (o clases hijas): “Coche”, “Camión” y “Avión”. Vehículo contendría los atributos y métodos comunes de ambos (marca, modelo, fecha de matriculación...) y las subclases tendrían las características únicas (del camión podría interesar almacenar la capacidad de carga y del avión el número de motores).

# Herencia (modificador abstract)

Podemos añadir el modificador *abstract* a la clase para no permitir instanciarla (de esta manera podemos dejar crear objetos unicamente de las clases hijas). Por ejemplo, imaginemos que queremos que se puedan crear objetos Empleado pero no Persona:

```
public abstract class Persona { ... }
```

De esta manera ya no es posible hacer *new Persona(...)*;



# Herencia (modificador abstract)

**Dentro de clases abstractas se puede añadir el modificador *abstract* también a métodos(\*). Un ejemplo de un método abstracto podría ser el siguiente:**

```
public abstract int hacerAlgo();
```

**Haciendo esto SE OBLIGA a las subclases a implementar este método. Si intentas extender una clase que tenga un método abstracto sin implementarlo en la subclase se mostrará un mensaje de error.**

\* ¿Recuerdas que es un método? Revisa qué es un atributo y un método y como se declaran ambos.

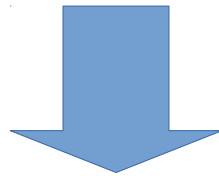
\*\* Documentación extra sobre clases abstractas:

<https://docs.oracle.com/javase/tutorial/java/landl/abstract.html>

# Polimorfismo

**Se puede declarar una variable del tipo de la superclase y asignarle un objeto de la subclase. Por ejemplo, podríamos crear una variable de tipo Persona y asignarle un objeto de tipo Empleado (su subclase):**

```
Persona persona = new Empleado("44112229U", "Ana", 40, 800.00);  
System.out.println(persona.toString());
```



```
Los datos de la persona son:  
DNI: 44112229U  
Nombre: Ana  
Edad: 40  
Salario: 800.0
```

# Interfaces (I)

Es posible declarar “contratos” que deben seguirse, para esto se utilizan interfaces. En este caso, arriba se declara la interfaz y abajo una clase que “implementa” esa interfaz (como veis hay errores):

```
1 package interfaces;
2
3 import clases.herencia.empleados.Empleado;
4
5 public interface IEmpleadoDAO {
6     public Empleado obtener(int indice);
7     public void anhadir(Empleado e);
8     public void eliminar(int indice);
9 }
10
11 public class EmpleadoDAO implements IEmpleadoDAO {
12
13 }
```

# Interfaces (II)

El primer problema aquí es que la interfaz y la clase que la implementa deben estar definidas en ficheros separados, el segundo error es que, si implementamos una interfaz, tenemos que implementar todos los métodos que se definen en la interfaz:

```
public class EmpleadoDAO implements IEmpleadoDAO {  
  
    @Override  
    public Empleado obtener(int indice) {  
        return null;  
    }  
  
    @Override  
    public void anhadir(Empleado e) {  
  
    }  
  
    @Override  
    public void eliminar(int indice) {  
  
    }  
  
}
```

# Interfaces (III)

**Recordemos que para herencia se utilizaba la palabra clave “extends” y para interfaces hablamos de implementar (“implements”).**

**Es importante saber que en Java es posible implementar múltiples interfaces pero solo es posible extender una única clase.**

# Polimorfismo y genéricos (I)

## Notas:

- Todas las clases en Java heredan de la clase *Object* (*java.lang.Object*).
- Mediante polimorfismo puedes declarar una variable de un tipo y crear una instancia de un objeto cuya clase herede de ese tipo. Utilizando el ejemplo de estas diapositivas:

```
6 public class ProgramaPrincipal {
7     public static void main(String[] args) {
8         Persona x = new Empleado("", "", 18, 1298.50);
9     }
10 }
```

Empleado hereda de Persona, por lo que (mediante lo que denominamos polimorfismo) podemos declarar una variable “x” de tipo “Persona” y asignarle un nuevo objeto de tipo “Empleado” (ya que hereda de Persona).

A continuación probamos lo mismo

con Object y vemos que funciona `Object y = new Empleado ("", "", 18, 1292.15);` correctamente:

\* Revisa la documentación oficial sobre la clase Object:  
<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

# Ejercicios

- **Intenta, utilizando tus clases Persona y Empleado, crear un empleado utilizando polimorfismo.**
  - Comprueba si consigues acceder a todos sus métodos correctamente y explica por qué puedes hacerlo o por qué no.
- **Crea una interfaz denominada ICalculadora que tenga los métodos:**
  - RealizarSuma: recibe dos parámetros de tipo double y devuelve un double.
  - RealizarElImprimirSuma: recibe dos parámetros de tipo double y no devuelve datos.
- **Posteriormente, crea una clase Calculadora que implemente la interfaz y prueba a utilizar polimorfismo con ella.**

# Polimorfismo y genéricos (II)

- En el primer ejercicio de la diapositiva anterior se ha declarado la variable “x” como tipo “Persona”. Esto implica que no tenga el método `getSalario()`. Para solucionar esto se puede hacer un *cast* (forzar una conversión) al tipo `Empleado`:

```
Persona x = new Empleado("", "", 18, 1298.50);  
Empleado emp = (Empleado)x;
```

- Esta aproximación, aunque en ocasiones es válida, puede crear diversos problemas. Por ejemplo, si haces el *cast* a un tipo equivocado (en vez de a un `Empleado` conviertes a un `String`) esto puede dar lugar a problemas inesperados en tiempo de ejecución (`ClassCastException`). Por tema de asegurar el funcionamiento del código en tiempo de compilación, en Java 5 se ha creado lo que se denomina como “genéricos” (se explicarán más adelante).



# Variables estáticas

**Si una variable o constante de una clase (es decir, un atributo de la clase) se declara como *static* se considera que el atributo pertenece a la propia clase y NO AL OBJETO. Esto es, para una clase “Casa”, si ponemos el atributo “metrosCuadrados” como estático, todas las casas que se creen tendrán los mismos metros cuadrados (porque el atributo es de la clase “Casa” y no de los objetos).**

# Métodos estáticos

**Al igual que los atributos, los métodos estáticos también pertenecen a la clase y no a los objetos de la clase (es importante entender esto).**

**Pueden (y deberían) ser llamados sin instanciar la clase. Es decir, un método estático puede llamarse sin hacer un *new* de la clase (sin crear un objeto de ella).**

**En lenguajes más modernos (como Scala o Kotlin) se sustituyen los atributos y métodos estáticos por una aproximación diferente denominada “*companion objects*”.**

# Métodos estáticos

En el siguiente ejemplo se muestra una llamada correcta a un método estático. Si te fijas, no se ha instanciado la clase “Calculadora”.

¿Crees que este código es correcto? ¿Podrías crear un objeto de tipo “Calculadora” previamente y después llamar al método? ¿Qué crees que sería más correcto?

`package` estaticos; **Clase “Calculadora”**

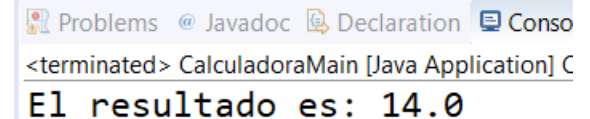
```
public class Calculadora {
    public static double sumar(double n1, double n2) {
        return n1 + n2;
    }
}
```

`package` estaticos; **Clase “Main”**

```
public class CalculadoraMain {
    public static void main(String[] args) {
        double resultado = Calculadora.sumar(4.0, 10.0);
        // No es necesario hacer Calculadora c = new Calculadora() y c.sumar(...)
        // debido a que el método tiene el modificador static

        System.out.println("El resultado es: " + resultado);
    }
}
```

## Resultado



Problems @ Javadoc Declaration Conso  
<terminated> CalculadoraMain [Java Application] C  
El resultado es: 14.0

# Ejercicio repaso (clases, objetos, bucles y estáticos)

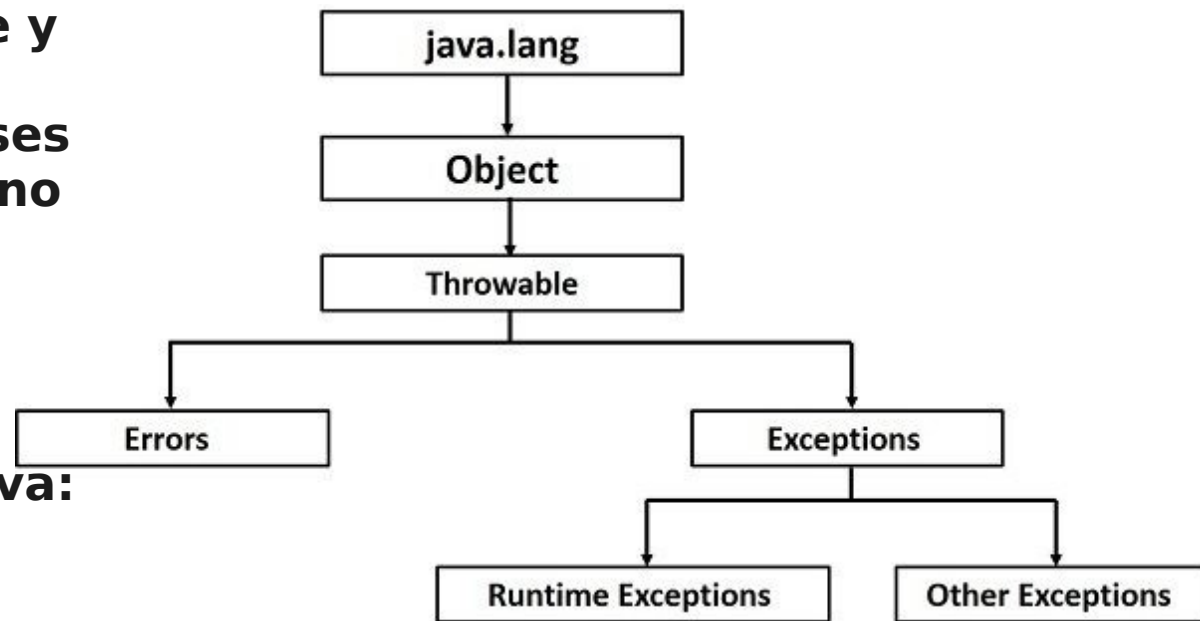
- **Crea un nuevo proyecto con una clase “Direccion” que tenga como atributos: calle, numero y ciudad. Haz que sus atributos sean privados (que solo sea posible acceder a ellos mediante métodos).**
- **Crea una clase Casa que tenga un atributo de tipo “Direccion”. La casa además tendrá un atributo metrosCuadrados y numeroHabitaciones. Sus atributos deben ser privados y solo accesibles mediante métodos.**
- **Haz un bucle *do-while* que pida datos de casas y, cada vez que se introduzca una, pregunte si se quiere introducir otra más.**
- **Cuando el usuario no introduzca más se mostrarán las casas almacenadas en el *array*.**
- **Por último, prueba a cambiar el atributo metrosCuadrados a público y estático y comprueba que metros cuadrados tiene cada casa que almacenas ahora. Explica por qué ocurre.**

# Excepciones (I)

En Java existe el concepto de “error” y de “excepción”. Si se produce un error el estado del programa no puede recuperarse y este se cierra. Las excepciones (Exception), en cambio, son clases que permiten detectar eventos no deseados, capturarlos y hacer algo en respuesta.

Las excepciones interrumpen el flujo normal del programa y existen dos tipos de estas en Java:

- Las que ocurren en tiempo de compilación (*checked exceptions*)
- Las que ocurren en tiempo de ejecución (*unchecked exceptions*).



\*Imagen extraída de: <https://www.quora.com/What-are-the-types-of-exceptions-in-Java>

# Excepciones (II)

```
try {
```

```
    Código a ejecutar
```

```
    ....
```

```
} catch (TipoExcepcion nombreExcepcion) {
```

```
    Código
```

```
    ...
```

```
}
```

**Si el código dentro del *try* falla, no seguirá ejecutándose y la ejecución del programa pasará directamente al bloque “catch”. En la siguiente diapositiva se mostrará esto con un ejemplo.**

# Excepciones (III)

¿Que crees que ocurrirá en el programa que se muestra a continuación? ¿Y si el divisor no fuese un 0?

```
public static void main(String[] args) {  
  
    int dividendo = 8;  
    int divisor = 0;  
  
    try {  
        // Dividimos 8 entre 0  
        int resultado = dividendo / divisor;  
        System.out.println("El resultado es: " + resultado);  
    } catch (ArithmeticException ae) {  
        System.out.println("ERROR: No es posible dividir entre 0.");  
    }  
}
```

# Excepciones (IV)

**try {**

Código normal a ejecutar

....

**} catch (TipoExcepcion nombreExcepcion) {**

Código que se ejecuta si falla algo en el "try"

...

**} finally {**

Código que se ejecuta siempre al final, **pase lo que pase.**

**}**



# Excepciones (V)

```
int dividendo = 8;
int divisor = 0;
int resultado;

try {
    // Dividimos 8 entre 0
    resultado = dividendo / divisor;
    System.out.println("El resultado es: " + resultado);
} catch (ArithmeticException ae) {
    System.out.println("ERROR: No es posible dividir entre 0.");
} finally {
    resultado = 120;
    System.out.println(
        "El programa finaliza. " +
        "El valor por defecto del resultado es: "
        + resultado
    );
}
```

¿Que ocurrirá?

¿Y si cambiamos el divisor?

## Excepciones (VI)

**Si una excepción se dispara dentro de código interior a un bloque “try”, el código salta al bloque “catch”.**

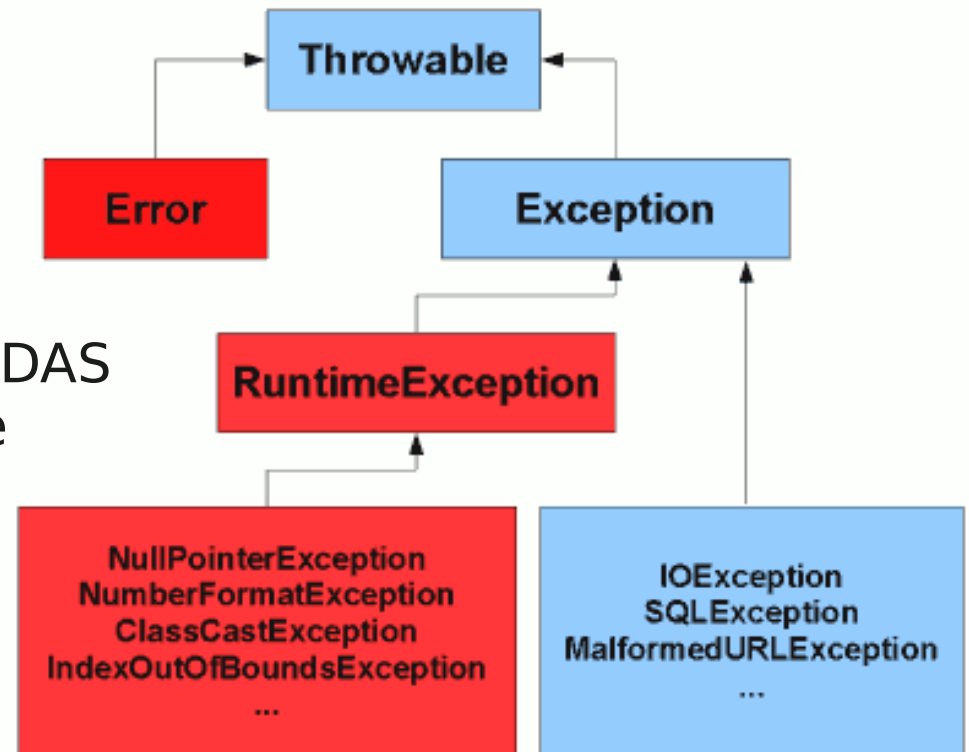
**En los ejemplos anteriores se captura una excepción denominada `ArithmeticException` que salta siempre que se intente dividir un número entre 0. Además de esta, existen muchos más tipos de excepciones y pueden crearse más.**

**Además del *try* y el *catch*, existe la palabra clave *throws* que veremos a continuación.**

# Excepciones (VII)

**En las excepciones existe una jerarquía de clases por herencia:**

- Todas las excepciones heredan de Exception.
- Si se captura una excepción de tipo “Exception”, se están capturando TODAS las que heredan de ellas (es decir, se valida cualquier tipo).
- El compilador no obliga a capturar excepciones que sean de tipo *RuntimeException* (*unchecked exceptions*).
- El compilador obliga a capturar las demás excepciones que compartan el tipo “Exception” (*checked exceptions*).



# Excepciones: ejercicio (I)

**Copia en tu IDE el ejemplo anterior y crea un objeto con valor nulo dentro del try. Después, llama a un método de ese objeto. Responde a estas preguntas:**

- 1) Antes de ejecutar el programa piensa, ¿Qué crees que ocurrirá en este?**
- 2) Ejecuta el programa e intenta explicar que ocurre.**
- 3) Intenta cambiar el tipo de la excepción a `NullPointerException` y ejecútalo de nuevo.**
- 4) Haz el mismo proceso con los tipos `RuntimeException` y, finalmente, `Exception`.**

## Excepciones: ejercicio (II)

**Crea una clase con un método *main* y declara un *array* que reserve memoria para tres valores.**

- 1) Inserta 4 valores al array y comprueba que pasa (sin capturar excepciones).**
- 2) Captura la excepción (puedes consultar la diapositiva anterior para saber qué tipo necesitas capturar).**
- 3) Prueba el método *printStackTrace()* de la clase *Exception*.**

# Excepciones (VIII)

**Dentro de un método es posible hacer un *try {} catch {}*. También, si se desea, se puede lanzar la excepción hacia arriba y capturarla desde otro sitio (catch).**

**En este ejemplo, el método `setEdad(...)` valida que no se introduzca una edad negativa.**

```
class Persona {
    private int nombre;
    private int edad;

    public void setEdad(int e) {
        if (e > 0) edad = e;
        else throw new RuntimeException("La edad no puede ser negativa.");
    }
}
```

# Excepciones: ejercicio

- 1) Crea la clase Persona de la diapositiva anterior e intenta ejecutar el código. Haz que se muestre un mensaje de error en caso de que se muestre una edad negativa.**
- 2) Consulta la documentación de las diapositivas anteriores sobre “RuntimeException”. Una vez leída, cambia el tipo de excepción por el tipo “Exception”. ¿Qué ocurre? Solucióvalo.**

# Excepciones personalizadas (I)

Al igual que existen un gran número de excepciones que heredan de “Exception” y de “RuntimeException”, tu también puedes crear excepciones personalizadas.

```
class ExcepcionEvitaValoresNulos extends Exception {  
    private static final long serialVersionUID = 1L;  
  
    public ExcepcionEvitaValoresNulos(String mensaje) {  
        // Llamamos al constructor de la clase Exception  
        super(mensaje);  
    }  
}
```



# Excepciones personalizadas (II)

## Probamos la excepción anterior:

```
public static void main(String[] args) {  
    try {  
        String s = null;  
  
        if (s == null) throw new ExcepcionEvitaValoresNulos("ERROR: valor nulo");  
    } catch (ExcepcionEvitaValoresNulos e) {  
        System.out.println(e.getMessage());  
    }  
}
```

**Resultado:** `<terminated> ExcepcionesPersonal`  
ERROR: valor nulo

**Ejercicio:** crea una excepción para edades negativas y utilízala en el código del método `setEdad` de las diapositivas anteriores.

# Exploración de las principales clases incorporadas en Java

# Bibliotecas Java (I)

- Las bibliotecas o librerías Java (*libraries*) son una serie de clases que ya vienen incorporadas en el *kit* de desarrollo de Java. Para utilizar una biblioteca de Java se utiliza la palabra clave *import*.
- Por ejemplo, para introducir datos por teclado se utiliza la clase Scanner. Para poder utilizarla es necesario escribir la siguiente línea:

```
import java.util.Scanner;
```
- Las principales clases Java están situadas en los paquetes *java.util*, *java.lang* y *java.io*.

# Interacción con el usuario: clase Scanner (I)

```
1 package interaccion;
2
3 import java.util.Scanner;
4 import clases.persona.Persona;
5
6 public class Main {
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9
10        Persona p = new Persona();
11
12        System.out.println("Introduzca nombre: ");
13        String nombre = sc.nextLine();
14        p.setNombre(nombre);
15
16        System.out.println("Introduzca edad: ");
17        int edad = sc.nextInt();
18        p.setEdad(edad);
19
20        System.out.println("Introduzca DNI:");
21        sc.nextLine();
22        String dni = sc.nextLine();
23        p.setDni(dni);
24
25        System.out.println("DATOS INTRODUCIDOS: ");
26        System.out.println(p.toString());
27
28        sc.close();
29    }
30 }
```

- La clase Scanner se sitúa en el paquete java.util (línea 3).
- Se inicializa la clase Scanner para entrada por teclado (System.in).
- Se captura la entrada de teclado con los métodos: nextLine(), nextInt(), nextDouble(), etc.
- Este código se puede mejorar capturando excepciones y eliminando variables innecesarias, en la siguiente diapositiva vemos como.

# Interacción con el usuario: clase Scanner (2)

```
6 public class Main {
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9
10        try {
11            Persona p = new Persona();
12
13            System.out.println("Introduzca nombre: ");
14            p.setNombre(sc.nextLine());
15
16            System.out.println("Introduzca edad: ");
17            p.setEdad(sc.nextInt());
18
19            System.out.println("Introduzca DNI:");
20            sc.nextLine();
21            p.setDni(sc.nextLine());
22
23            System.out.println("DATOS INTRODUCIDOS: ");
24            System.out.println(p.toString());
25
26        } catch (Exception e) {
27            System.err.println("Ha ocurrido un error al capturar la entrada.");
28
29        } finally {
30            // Lo que ponga aquí se ejecuta siempre
31            sc.close();
32        }
33    }
34 }
```

- Eliminamos las variables “nombre”, “edad” y “dni” y los ponemos en las llamadas a los métodos directamente.
- Capturamos las excepciones que puedan producirse.

# Interacción con el usuario: BufferedReader (I)

```
package interaccion;

import java.io.BufferedReader;
import java.io.InputStreamReader;

import clases.persona.Persona;

public class MainReader {
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        Persona p = new Persona();

        try {
            System.out.println("Introduzca nombre: ");
            p.setNombre(br.readLine());

            System.out.println("Introduzca edad: ");
            p.setEdad(Integer.parseInt(br.readLine()));

            System.out.println("Introduzca DNI:");
            p.setDni(br.readLine());

            System.out.println("DATOS INTRODUCIDOS: ");
            System.out.println(p.toString());

        } catch (Exception ex) {
            // Puede darse si, por ejemplo, introducimos una cadena en vez de un entero
            System.out.println("Dato introducido no son válido.");
        }
    }
}
```

Similar a lo anterior, pero esta vez utilizamos la clase **BufferedReader**.

**BufferedReader**, **InputStreamReader** así como sus análogas **Writer** se encuentran en el paquete *java.io*.

**Integer** es una clase Java. Utilizando su función *parseInt* podemos convertir un **String** en un número entero.

No estamos cerrando el buffer (**br.close()**)

# Interacción con el usuario: BufferedReader (II)

```
9 public class MainReader {
10     public static void main(String[] args) {
11         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
12
13         Persona p = new Persona();
14
15         try {
16             System.out.println("Introduzca nombre: ");
17             p.setNombre(br.readLine());
18
19             System.out.println("Introduzca edad: ");
20             p.setEdad(Integer.parseInt(br.readLine()));
21
22             System.out.println("Introduzca DNI:");
23             p.setDni(br.readLine());
24
25             System.out.println("DATOS INTRODUCIDOS: ");
26             System.out.println(p.toString());
27
28         } catch (IOException ioe) {
29             // Capturamos excepcion de entrada/salida
30             System.out.println("Error de entrada/salida.");
31
32         } catch (NumberFormatException nfe) {
33             // Capturamos excepcion que puede saltar al llamar a Integer.parseInt()
34             System.out.println("El numero introducido no es un entero.");
35         }
36     }
37 }
```

**Capturamos dos excepciones anidadas. Una si se produce un error de entrada salida (IOException) y otra si en la línea 20 el usuario ha introducido un valor que no es de tipo *Int*.**

**Pero hay un problema, nos falta cerrar el buffer**

# Interacción con el usuario: BufferedReader (III)

```
public class MainReader {
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Persona p = new Persona();

        try {
            System.out.println("Introduzca nombre: ");
            p.setNombre(br.readLine());

            System.out.println("Introduzca edad: ");
            p.setEdad(Integer.parseInt(br.readLine()));

            System.out.println("Introduzca DNI:");
            p.setDni(br.readLine());

            System.out.println("DATOS INTRODUCIDOS: ");
            System.out.println(p.toString());

        } catch (IOException ioe) {
            // Capturamos excepcion de entrada/salida
            System.out.println("Error de entrada/salida.");
        } catch (NumberFormatException nfe) {
            // Capturamos excepcion que puede saltar al llamar a Integer.parseInt()
            System.out.println("El numero introducido no es un entero.");
        } finally {
            try {
                if (br != null) br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

**Ahora cerramos el stream (br.close()) pero...**

**¡Tenemos que capturar otra excepción dentro del finally! Menudo lío. Esto era necesario en Java 6.**

**A partir de Java 7 se puede hacer que el objeto se cierre sin hacer el código tan complejo, lo vemos en la siguiente diapositiva.**



# Interacción con el usuario: BufferedReader (IV)

```
public class MainReader {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in))) {
            Persona p = new Persona();

            System.out.println("Introduzca nombre: ");
            p.setNombre(br.readLine());

            System.out.println("Introduzca edad: ");
            p.setEdad(Integer.parseInt(br.readLine()));

            System.out.println("Introduzca DNI:");
            p.setDni(br.readLine());

            System.out.println("DATOS INTRODUCIDOS: ");
            System.out.println(p.toString());

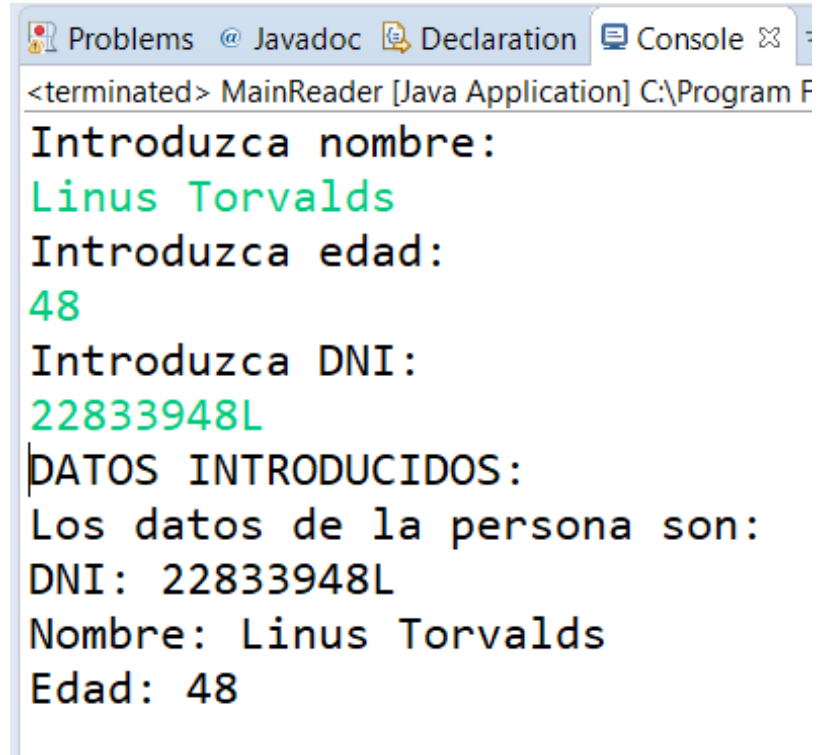
        } catch (IOException ioe) {
            // Capturamos excepcion de entrada/salida
            System.out.println("Error de entrada/salida.");
        } catch (NumberFormatException nfe) {
            // Capturamos excepcion que puede saltar al llamar a Integer.parseInt()
            System.out.println("El numero introducido no es un entero.");
        }
    }
}
```

**Hemos eliminado el *finally* y cambiado el *try*. Ahora englobamos el objeto *BufferedReader* entre paréntesis y se realiza un cierre al final sin declararlo nosotros explícitamente. Esta es la mejor manera de hacerlo.**

# Interacción con el usuario: ejecución

Como se puede ver, hay dos formas de interactuar con el usuario (se puede hacer utilizando la clase `BufferedReader` o bien la clase `Scanner`).

Se han mostrado dos programas diferentes con sus mejoras progresivas, pero todos funcionan y tienen la misma salida, que es la de la imagen situada a la derecha de este texto.



```
Problems @ Javadoc Declaration Console
<terminated> MainReader [Java Application] C:\Program F
Introduzca nombre:
Linus Torvalds
Introduzca edad:
48
Introduzca DNI:
22833948L
DATOS INTRODUCIDOS:
Los datos de la persona son:
DNI: 22833948L
Nombre: Linus Torvalds
Edad: 48
```

# Preguntas (I)

**¿Que tres elementos forman parte de una clase?**

**¿Cómo llamas al constructor de una clase?**

## Preguntas (II)

**¿Que tres elementos forman parte de una clase?**

**Atributos, constructores y métodos.**

**¿Cómo llamas al constructor de una clase?**

**Mediante la palabra clave *new*. Por ejemplo, si se dispone de la clase “Camion”:**  
**`new Camion(parámetros);`**

# Preguntas (III)

De las siguientes líneas de código, indica qué crees que es cada parte y razónalo:

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
  
    Persona p = new Persona();  
    p.setNombre(sc.nextLine());  
}
```

# Estructuras estáticas: Arrays

```
package arrays;

import clases.herencia.empleados.Empleado;

public class MainArrays {
    public static void main(String[] args) {
        // Inicializo el array (tamaño fijo de 100 caracteres)
        Empleado[] empleados = new Empleado[100];

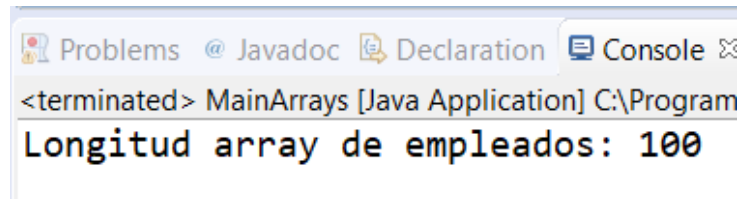
        // Introduzco tres empleados en el array
        for (int i = 0; i < 3; ++i) {
            Empleado e = new Empleado(null, null, 0, 0);

            empleados[i] = e;
        }

        System.out.println("Longitud array de empleados: " + empleados.length);
    }
}
```

**Definición de un array de objetos de tipo “Empleado”. Primero creamos el array de un tamaño MÁXIMO de 100 elementos, posteriormente rellenamos las primeras posiciones con tres empleados.**

**Resultado:**



```
Problems @ Javadoc Declaration Console
<terminated> MainArrays [Java Application] C:\Program
Longitud array de empleados: 100
```

# Estructuras estáticas: Arrays (desventajas)

- Se define la longitud máxima del array al declararlo y este tamaño **NO PUEDE CAMBIAR**. Ejemplo: se declara un array de 100 números enteros y se añaden 101 (el programa fallaría).
- Existen problemas al eliminar elementos de un array. Si en un array de 5 elementos eliminas el de la posición 2, esa posición quedaría vacía. Una manera de arreglar esto es poner el elemento de la última posición en la posición 2 y eliminar el último (pero esto tiene otro inconveniente, y es que si lo que se tiene es un array ordenado se pierde este orden).
- Tienes que almacenar el número de elementos que tienes en el array en otra variable. Esto es, si tu declaras un array de máximo 20 elementos, pero has introducido 4, tienes que almacenar la longitud en una variable para saber el número de elementos que tienes en ese momento.

**En Java existe la interfaz List (con las clases ArrayList y LinkedList). Estas clases nos permiten solucionar todos los problemas anteriores así como tratar con conjuntos de elementos de forma mucho más sencilla.**

# Ejercicio sobre desventajas de arrays

- 1) Crea un programa que pida al usuario números hasta que se introduzca el -1 (declara un *array* de tamaño 10).**
- 2) Muestra al usuario los valores del *array*.**
- 3) Pide al usuario que introduzca una posición y elimina el elemento de esa posición. Posteriormente vuelve a mostrar los datos del array. ¿Qué ocurre? ¿Como lo solucionas?**
- 4) Crea un nuevo *array* del mismo tamaño que el anterior que almacene los elementos del anterior *array* ordenados de menor a mayor.**
- 5) Muestra los valores del nuevo *array*.**
- 6) Elimina un valor e introduce dos nuevos valores en el nuevo *array*. Posteriormente vuelve a mostrar los datos. ¿Que problema encuentras ahora?**



# Estructuras dinámicas: listas

En Java se permite declarar listas de elementos. La interfaz `List` se sitúa en el paquete `java.util` y contiene métodos de añadido, modificación y borrado de elementos así como muchos otros. Ejemplo:

```
public class Main {
    public static void main(String[] args) {
        ArrayList<String> listaNombres = new ArrayList<>();
        listaNombres.add("Richard Stallman");
        listaNombres.add("Linus Torvalds");
        listaNombres.add("Dennis Ritchie");

        System.out.println(
            "Numero de elementos: " + listaNombres.size() + "\n" +
            "Primer elemento: " + listaNombres.get(0) + "\n" +
            "Segundo elemento: " + listaNombres.get(1) + "\n"
        );

        listaNombres.remove(0); // Elimina el elemento de la posicion 0
        String s = listaNombres.get(0); // Obtiene el elemento de la posicion 0
    }
}
```

# Estructuras dinámicas: listas

**La salida del código anterior es la siguiente:**

```
Numero de elementos: 3  
Primer elemento: Richard Stallman  
Segundo elemento: Linus Torvalds
```

# Estructuras dinámicas: listas (recorrido)

Hay varias maneras de recorrer listas. Una forma es utilizando un bucle for desde 0 hasta `lista.size()`, otra es utilizando un *foreach*:

```
// Recorrer una lista y mostrar los nombres
for (int i = 0; i < listaNombres.size(); ++i) {
    System.out.println(listaNombres.get(i));
}
```

```
System.out.println("\n\n");
```

```
// Otra forma (utilizando un foreach)
for (String nombre : listaNombres) {
    System.out.println(nombre);
}
```



Salida

Linus Torvalds  
Dennis Ritchie

Linus Torvalds  
Dennis Ritchie

# Estructuras dinámicas: listas e iteradores

Nos creamos dos objetos Empleado y los añadimos a una lista. También introducimos otra manera de recorrer estos objetos: los iteradores:

```
List<Empleado> listaEmps = new ArrayList<>();  
Empleado e1 = new Empleado("44112229U", "Ana", 40, 800.00);  
Empleado e2 = new Empleado("54113459L", "Paco", 30, 800.00);
```

```
listaEmps.add(e1);  
listaEmps.add(e2);
```

```
Iterator<Empleado> i = listaEmps.iterator();  
while (i.hasNext()) {  
    System.out.println(i.next().getNombre());  
}
```

SALIDA

Ana  
Paco

i.next() recorre el iterador devolviendo cada objeto hasta que hasNext() devuelva false

# Estructuras dinámicas: listas (ventajas)

- **No es necesario decir el tamaño máximo desde un inicio. Simplemente se crea la lista vacía y se van añadiendo y eliminando elementos.**
- **Añadido, actualización y eliminación de elementos de manera simple.**
- **La longitud de la lista se mantiene dentro del propio objeto (se puede llamar a `miLista.size()`).**
- **Existen gran cantidad de estructuras dinámicas con diferentes características: Lists, Sets, Maps, Stacks, Queues, etc. Se recomienda investigar sobre ellas.**

# Estructuras dinámicas: conjuntos (I)

**Los conjuntos (en inglés sets) sirven para almacenar elementos (igual que las listas o los arrays). Se diferencian de estas en lo siguiente:**

- **No mantienen el orden de los elementos.**
- **No se pueden repetir elementos.**
- **No tiene un método *get* para obtener el elemento de una posición del array (ya que no está ordenado). Por tanto, solo puede recorrerse o bien con un iterador o bien con un bucle *foreach*.**
- **En Java tiende a utilizarse la interfaz Set y su implementación HashSet. Existen otras implementaciones como LinkedHashSet y TreeSet que sí ordenan los elementos (no las utilizaremos).**

# Estructuras dinámicas: conjuntos (II)

## Ejemplo de manejo de operaciones de un conjunto de elementos:

```
Set<Integer> edades = new HashSet<>();
edades.add(18);
edades.add(23);
edades.add(23);
edades.add(47);

System.out.println("Hay un total de: " + edades.size() + " edades.");

// Elimina el num 23
edades.remove(23);

if (edades.contains(18)) {
    System.out.println("El 23 está en el conjunto.");
} else {
    System.out.println("El 23 no está en el conjunto.");
}
```

¿Qué resultado saldrá por pantalla en el primer println?

¿Qué mostrará después de borrar el número 23?

# Estructuras dinámicas: ejercicios

- **¿Qué diferencias hay entre una lista y un conjunto?**
- **¿En que se parece una lista a un conjunto?  
¿Para qué sirven? ¿cuándo utilizarías una u otra?**
- **En Java, ¿qué interfaz se utiliza para listas y cuál para conjuntos?**
- **¿Puedes recorrer los elementos de un conjunto (set) con un bucle while/for? ¿y de una lista?**



# Estructuras dinámicas: Tablas Hash (I)

- **De manera genérica se llaman Tablas Hash. También se les suele denominar como diccionarios (Python y PHP) o maps (Java).**
- **Un Map es una estructura que almacena pares de elementos clave → valor. Por ejemplo:**

# Estructuras dinámicas: Tablas Hash (II)

A continuación se muestra un ejemplo de uso de un Map en Java.

```
public static void main(String[] args) {  
    // Creo un map cuya clave es un String y el valor es un String  
    Map<String, String> persona = new HashMap<>();  
  
    // Relleno el map con claves y valores  
    persona.put("nombre", "Alan");  
    persona.put("apellido", "Turing");  
    persona.put("nacionalidad", "británica");  
  
    // Obtengo el valor almacenado en la clave nombre.  
    String nombre = persona.get("nombre");  
    System.out.println(nombre);  
  
    // Recorro el map con un foreach (Voy guardando los pares clave-valor en la variable par)  
    for (Entry<String, String> par : persona.entrySet()) {  
        System.out.println("Clave: " + par.getKey());  
        System.out.println("Valor: " + par.getValue());  
    }  
}
```

# Estructuras dinámicas: Tablas Hash (III)

## Ejercicio:

- **Crea un Map que almacene los nombres de países con su población (clave el nombre del país y valor su población). Puedes consultarlos en <https://countrycode.org/>**
- **Realiza búsqueda de elementos y recorrido de los mismos con un bucle.**

# Manejo de ficheros (I)

Las principales clases de manejo de ficheros en Java son las siguientes:

- **File**: representación de un fichero en Java. La siguiente línea carga el fichero situado en la ruta pasada como parámetro.

```
File fichero = new File("nominas.txt");
```

- Algunos métodos útiles de esta clase son:
  - exists(): devuelve *true* o *false* dependiendo de si existe o no el fichero.
  - canRead(), canWrite(), canExecute(): devuelve *true* o *false* si se pueden realizar las operaciones de leer, escribir o ejecutar el fichero.
  - isFile(), isDirectory(), isHidden(), lastModified().
  - getCanonicalPath(), getAbsolutePath(), etc.
  - createNewFile(): si no existe el fichero, lo crea.

# Manejo de ficheros (II)

Las principales clases de manejo de ficheros en Java son las siguientes:

- **FileWriter**: clase que se encarga de manejar la escritura en ficheros. A continuación se crea un objeto de tipo `FileWriter` que carga los datos del fichero “`nominas.txt`” y se prepara para escribir sobre él:

```
FileWriter writer = new FileWriter("nominas.txt");
```

- Algunos métodos útiles de esta clase son:
  - `write(char[] frase)`: escribe un array de caracteres en el fichero sobre el que se ha creado el objeto.
  - `close()`: cierra el fichero.

# Manejo de ficheros (III)

Las principales clases de manejo de ficheros en Java son las siguientes:

- FileReader: clase que se encarga de manejar la lectura en ficheros. A continuación se crea un objeto de tipo `FileReader` que carga los datos del fichero “`nominas.txt`” y se prepara para leer sobre él:

```
FileReader reader = new FileReader("nominas.txt");
```

- Algunos métodos útiles de esta clase son:
  - `read(char[] frase)`: lee del fichero y almacena los datos en la variable “frase”.
  - `close()`: cierra el fichero.

# Manejo de ficheros (IV)

Las principales clases de manejo de ficheros en Java son las siguientes:

- **BufferedReader**: clase que se encarga de manejar la lectura en ficheros. A continuación se crea un objeto de tipo **BufferedReader** que carga los datos del fichero “nominas.txt” y se prepara para leer sobre él:

```
FileReader fr = new FileReader("nominas.txt");  
BufferedReader br = new BufferedReader(fr); // Recibe un FileReader
```

- Algunos métodos útiles de esta clase son:
  - read(char[] frase): lee del fichero y almacena los datos en la variable “frase”.
  - readLine(): lee el fichero línea a línea.
  - close(): cierra el fichero.

# Manejo de ficheros (V)

**Ejemplo de uso de BufferedReader. Este ejemplo lee dos líneas de un fichero y las escribe en consola.**

```
try {  
    BufferedReader br = new BufferedReader(new FileReader("asdf.txt"));  
  
    String linea1 = br.readLine();  
    String linea2 = br.readLine();  
  
    System.out.println(linea1);  
    System.out.println(linea2);  
  
    br.close(); // ¿Que ocurre con este close?  
  
} catch (IOException e) {  
    e.printStackTrace();  
}
```



# Manejo de ficheros (VI)

**Ejercicio 1**: mejora el código anterior de manera que utilice un `try-with-resources`. De esta manera el cerrado del fichero se realiza siempre.

**Ejercicio 2**: mejora el código anterior para que, en vez de leer dos líneas, lea cualquier número de líneas que haya en el fichero (**ver ejemplo**).

# Ejercicio: ficheros, maps, strings

## Ejercicio: Crea el siguiente fichero “config.txt”:

```
url_bbdd=localhost:3306/personas_bbdd
usuario_bbdd=miusuario
contrasenha_bbdd=mipassword

tipo_bd=memoria
```

## Carga el fichero, procésalo y guarda los datos en un Map.

*\* Este ejercicio sirve para practicar con arrays, strings (método split), ficheros, estructuras de datos y excepciones.*

# Ficheros y excepciones (I)

Con la sentencia `try { } catch (Exception e) {}` capturamos las excepciones que se produzcan en el código.

Este fragmento de código crea un fichero en el directorio del proyecto.

```
package ficheros;

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

public class MainFicheros {
    // Almacenar en fichero
    public static void main(String[] args) {
        File fichero = new File("fichero.txt");
        BufferedWriter writer;
        try {
            writer = new BufferedWriter(new FileWriter(fichero));

            writer.write("Aquí escribo el texto del fichero.");
            writer.close();
        } catch (IOException e) {
            // Si el fichero no existe salta una excepción de la clase
            // IOException de Java. Aquí deberíamos manejar lo que
            // queremos que se haga si el programa falla.
            e.printStackTrace();

            System.err.println("No se ha podido crear el fichero.");
        }
    }
}
```

## Ficheros y excepciones (II)

**Vamos a extraer el código anterior que almacenaba un texto en un fichero para que todo quede más ordenado y sea reutilizable. Para esto creamos una clase nueva a la que ponemos el nombre de “UtilidadesFicheros” y un método estático para almacenar un texto en un fichero dado. Recordemos que para esto necesitamos pasarle el nombre del fichero y el texto a guardar.**

# Ficheros y excepciones (III)

## El código de la clase es el siguiente:

```
package ficheros;

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

public class UtilidadesFicheros {

    public static void escribirEnFichero(String nombreFichero, String texto)
        throws IOException {

        File fichero = new File(nombreFichero);
        BufferedWriter writer = new BufferedWriter(new FileWriter(fichero));

        writer.write(texto);
        writer.close();
    }
}
```

# Ficheros y excepciones (IV)

En el código de la clase anterior no capturábamos la excepción directamente sino que la LANZAMOS en la definición de la función (utilizamos la palabra clave `throws`). Esto lanza la excepción hacia arriba y deberá ser capturada con el *try/catch* donde se haga la llamada a la función:

```
package ficheros;
```

```
import java.io.IOException;
```

```
public class MainFicheros {  
    // Almacenar en fichero  
    public static void main(String[] args) {  
        // Utilizamos métodos estáticos y capturamos la excepción en el main  
        try {  
            UtilidadesFicheros.escribirEnFichero("fichero.txt", "texto");  
        } catch (IOException e) {  
            e.printStackTrace();  
            System.err.println("No se ha podido crear el fichero.");  
        }  
    }  
}
```

# Ejercicio

**Fíjate en cómo está definida la clase “UtilidadesFicheros” y el método de la diapositiva anterior y responde a las siguientes preguntas:**

- **¿Que significa el *static* de la cabecera de la función?**
- **¿Cuántos parámetros recibe la función? ¿Que tipo de dato devuelve?**
- **Después de contestar a estas dos preguntas, crea un proyecto de eclipse de nombre “pruebasficheros”. Copia en él la clase anterior y utilízala para guardar un string en un fichero.**

# Ficheros y excepciones (V)

- En el código anterior se utiliza la palabra clave ***static***. Una función “estática” permite llamar a un método de una clase sin instanciarla, es decir, hemos llamado a **UtilidadesFicheros.escribirEnFichero(...)** sin hacer un **new** (sin crear un objeto de esa clase).
- En Java existen las clases **Exception** y **RuntimeException** así como otras clases excepción que heredan de estas. Nosotros también podemos crear nuestras propias clases que hereden de ellas.



# Ejercicio repaso excepciones (I)

En la clase **Persona** presentada en diapositivas anteriores existía un atributo **edad**.

**1) Crea una clase excepción personalizada (heredando de `Exception`) para validar que la edad de una persona no sea inferior a 0 años.**

**¿Qué nombre le pondrías a la excepción?**

**¿En qué método o métodos crees que hay que validar la edad de la persona?**

**Pista: Las excepciones pueden lanzarse manualmente - *if (algo) throw new Exception(...)* - y capturarse con (*try-catch*).**

# Ejercicio repaso ficheros y excepciones

- **Modifica el código que has copiado de la clase UtilidadesFicheros del ejercicio anterior para que, en vez de lanzar una IOException lance una excepción personalizada que crees tu a mano.**

**Pista: Para crear una excepción personalizada debes heredar de otra clase excepción.**

**Prueba a crear una excepción (por ejemplo “ExcepcionFicheros”) y haz que extienda primero de “Exception” y posteriormente de “RuntimeException”.**

**Documenta el proceso y di que diferencias aprecias entre heredar de una clase excepción y de otra.**

# Acceso a bases de datos

# Acceso a datos con JDBC (I)

**JDBC es una especificación para conectar programas escritos en Java con bases de datos. A continuación se mostrarán los pasos para conectarse a una base de datos en Java utilizando JDBC:**

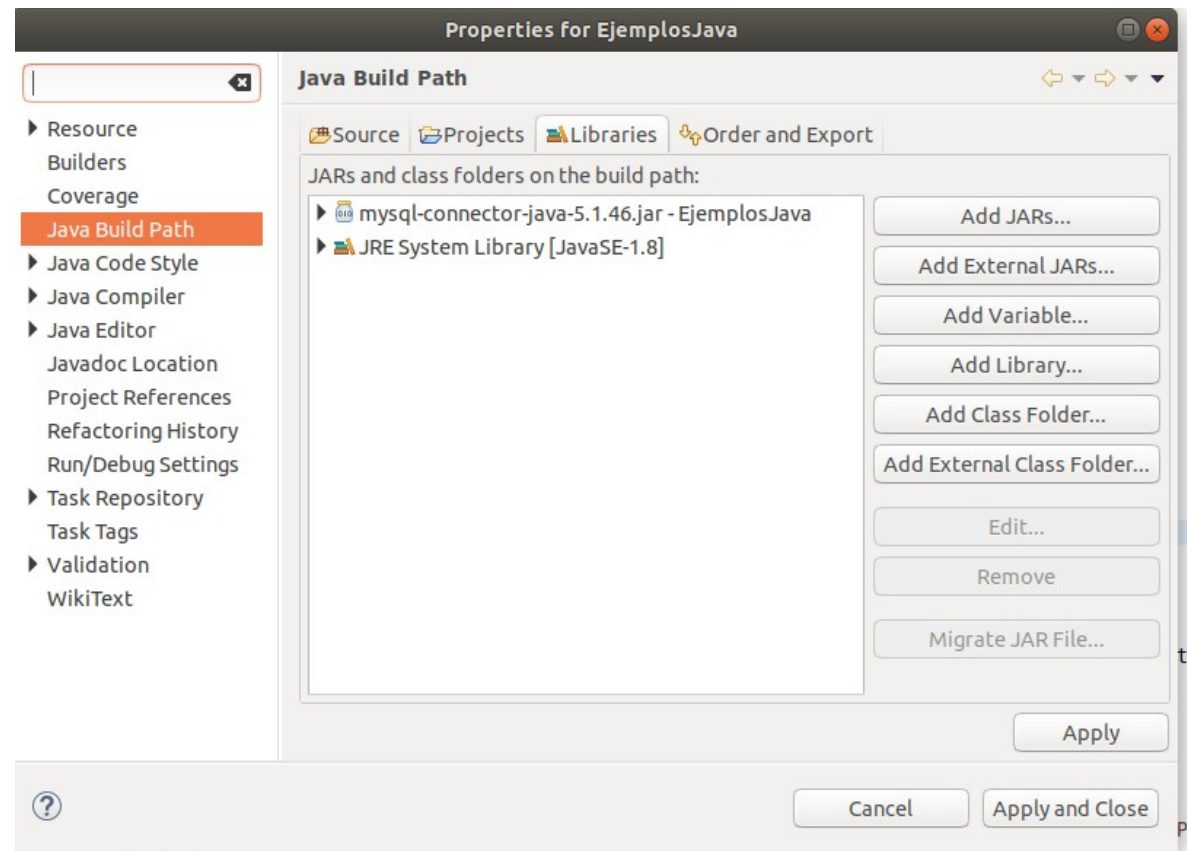
**1) Buscar en Google el conector para la base de datos que utilicemos (Oracle, MySQL, PostgreSQL, etc) y asegurarnos de que la versión del driver se adecúa a la versión de la base de datos. Por ejemplo, **para MySQL podemos buscar esto.****

\*<https://www.theserverside.com/definition/Java-Database-Connectivity-JDBC>

# Acceso a datos con JDBC (II)

**2) Debemos añadir la librería al *Java build path*. Para esto copiamos el JAR en la carpeta del proyecto.**

**3) Posteriormente, debe accederse a las propiedades del proyecto y pulsar en “Add JARs” (en caso de usar Eclipse) y añadir el driver descargado.**



# Acceso a datos con JDBC (III)

**4) Con los pasos anteriores ya tenemos el Driver de conexión a base de datos en el *build path*. Con esto es posible acceder y almacenar datos en una base de datos MySQL mediante código. Para hacerlo son necesarias:**

- Una base de datos.**
- Un usuario/contraseña.**
- La IP y el puerto de la base de datos (en este ejemplo se utilizará localhost y el puerto por defecto de MySQL (3306)).**

# Acceso a datos con JDBC (IV)

## Clases de acceso a base de datos en Java:

- **DriverManager**: mediante el método `getConnection` devuelve un objeto con los datos de conexión de la base de datos.
- **Connection**: contiene la información para conectarse (url, puerto, usuario, contraseña y nombre de la base de datos).
- **Statement y PreparedStatement**: a partir del objeto de tipo `Connection` se pueden crear *statements*. Un `Statement` realiza consultas SQL, p. ej:

```
statement.execSQL("insert into Persona ('nombre', 'edad') values ('Pepe', 25)");
```

- **ResultSet**: almacena conjuntos de datos (es un set, es decir, un conjunto) y debe recorrerse con un iterador.

```
Connection con = DriverManager.getConnection("jdbc:mysql://ip:puerto/nombrebd", "pepeuser", "pepepassword");
```

```
Statement sentencia = con.createStatement();
```

```
ResultSet rs = sentencia.execSQL("SELECT * FROM PERSONAS"); // También existe el método "execute"
```

# Acceso a datos con JDBC (V)

Este código de ejemplo tiene la siguiente salida:

ID: 1, nombre: marcos, edad: 30

A continuación se explicará el código en detalle.

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

        try {
            Connection conn = getConnection();
            Statement stmt = null;
            ResultSet rs = null;

            stmt = conn.createStatement();
            rs = stmt.executeQuery("SELECT * FROM empleado");
            while (rs.next()) {
                String id = rs.getString("id");
                String nombre = rs.getString("nombre");
                int edad = rs.getInt("edad");
                System.out.println("ID: " + id + ", nombre: " + nombre + ", edad: " + edad);
            }

        } catch (SQLException | ClassNotFoundException e) {
            System.err.println("Ha ocurrido un error: " + e.getMessage());
        }

    }

    public static Connection getConnection() throws SQLException, ClassNotFoundException {
        Connection conn = null;

        Class.forName("com.mysql.jdbc.Driver");
        String connectionUrl = "jdbc:mysql://localhost:3306/pruebas";
        String connectionUser = "mnceleiro";
        String connectionPassword = "passdeprueba";
        conn = DriverManager.getConnection(connectionUrl, connectionUser, connectionPassword);

        return conn;
    }
}
```



# Acceso a datos con JDBC (VI)

## 5) Las siguiente método se corresponde con la conexión a la BBDD:

```
32 public static Connection getConnection() throws SQLException, ClassNotFoundException {
33     Connection conn = null;
34
35     Class.forName("com.mysql.jdbc.Driver");
36     String connectionUrl = "jdbc:mysql://localhost:3306/pruebas";
37     String connectionUser = "mnceleiro";
38     String connectionPassword = "passdeprueba";
39     conn = DriverManager.getConnection(connectionUrl, connectionUser, connectionPassword);
40
41     return conn;
42 }
43 }
```

- La línea 35 se encarga de buscar el Driver en el *build path* (se ha añadido en el paso 3).
- La línea 36 se corresponde con la cadena de conexión (se indica que se utiliza una BBDD MySQL, la dirección IP es localhost, el puerto el 3306 y el nombre de la base de datos es "pruebas").
- En las líneas 37 y 38 se definen dos Strings con el usuario y la contraseña de la base de datos.
- Por último, en la línea 39 se obtiene el objeto de conexión a la misma.

# Acceso a datos con JDBC (VII)

6) Por último, el siguiente código ejecuta una consulta (*query*) a la base de datos en la que se buscan todos los empleados (en este ejemplo solo hay uno insertado).

En este ejemplo se crea objeto *Statement*. Este objeto se utiliza para ejecutar consultas SQL y almacenar los resultados. Estos resultados se almacenan en un *ResultSet* que se va recorriendo en el bucle de la línea 19.

```
12     try {
13         Connection conn = getConnection();
14         Statement stmt = null;
15         ResultSet rs = null;
16
17         stmt = conn.createStatement();
18         rs = stmt.executeQuery("SELECT * FROM empleado");
19         while (rs.next()) {
20             String id = rs.getString("id");
21             String nombre = rs.getString("nombre");
22             int edad = rs.getInt("edad");
23             System.out.println("ID: " + id + ", nombre: " + nombre + ", edad: " + edad);
24         }
25
26     } catch (SQLException | ClassNotFoundException e) {
27         System.err.println("Ha ocurrido un error: " + e.getMessage());
28     }
```

# Programación Orientada a Objetos (JAVA)

Ciclo Superior DAM – Programación 2018/2019 – Aula Nosa

Marcos Núñez Celeiro