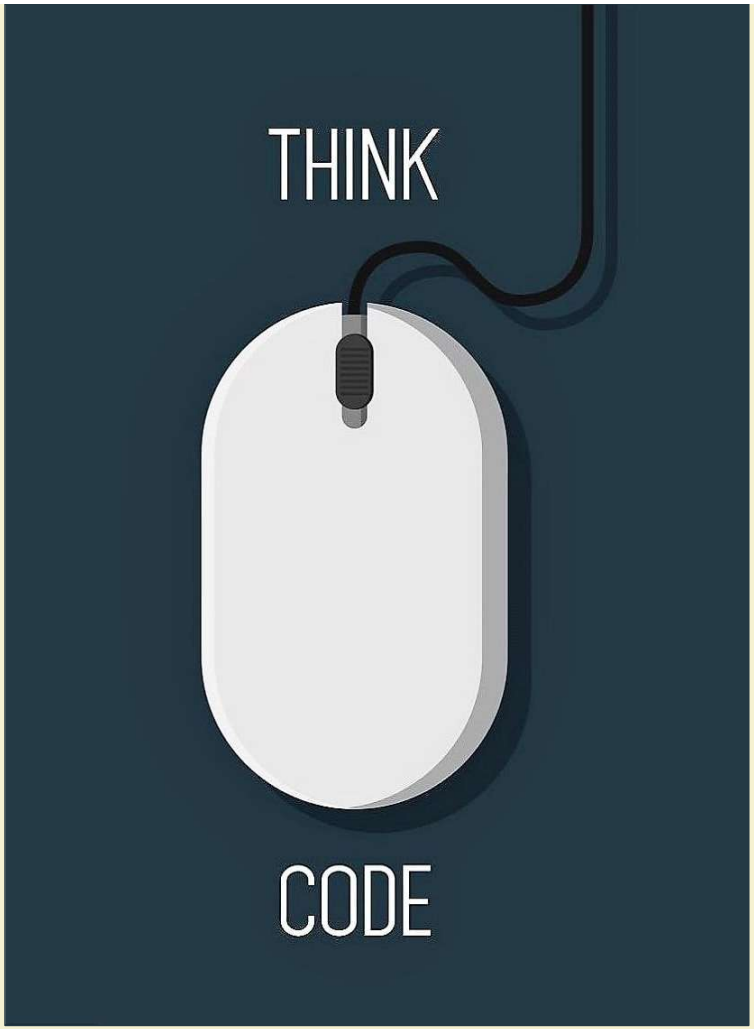
01

Unidad Didáctica 1: Introducción



Módulo: Programación

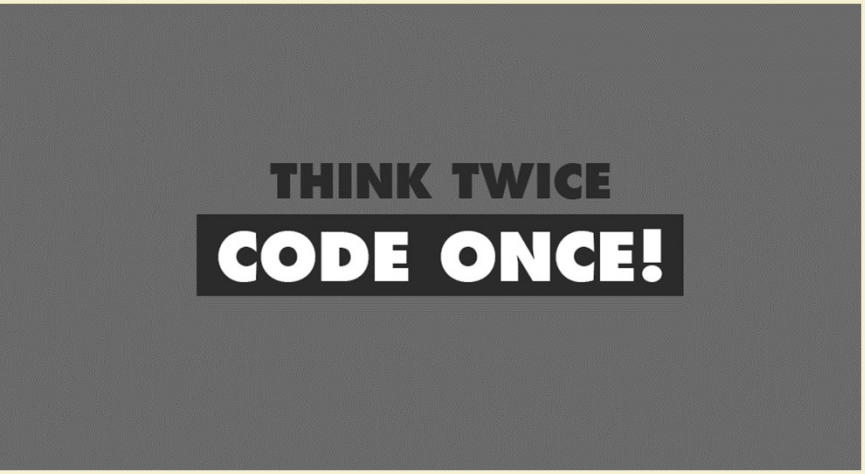
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

Índice



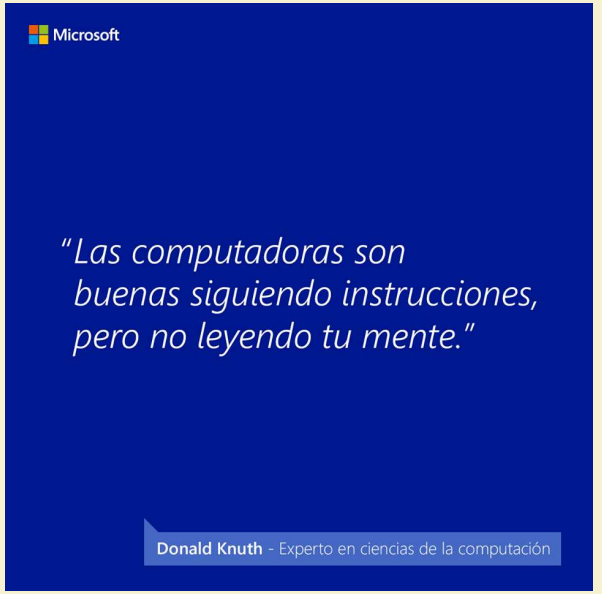
1: Conceptos

¿Qué es la programación?

- Acción de crear programas o aplicaciones a través del desarrollo de un código fuente, que se basa en el conjunto de instrucciones que sigue el ordenador para ejecutar un programa.
- La programación es la lógica que permite que un ordenador funcione y realice las tareas que el usuario solicita.
- Para que el ordenador entienda qué acciones debe realizar, los programadores necesitan comunicarse con el ordenador haciendo uso de uno o varios lenguajes, conocidos como lenguajes de programación.

¿Qué es un lenguaje de programación?

- El lenguaje de programación es un idioma artificial prediseñado formado por signos, palabras y símbolos que permite la comunicación entre el programador y el ordenador.
- Las instrucciones que sigue el ordenador para la ejecución de aplicaciones y programas están escritas en lenguaje de programación y luego son traducidas a un lenguaje de máquina que puede ser interpretado y ejecutado por el hardware del equipo (parte física).
- El código fuente está formado por líneas de texto que expresan en lenguaje de programación las instrucciones que debe llevar a cabo el ordenador. Este código es creado, diseñado, codificado, mantenido y depurado a través de la programación.
- Existen diferentes lenguajes de programación (Java, PHP, Python...) que se valen de diversos programas en los que se vuelcan las instrucciones. Estos lenguajes varían con el tiempo, se expanden y evolucionan.



Historia de la programación

- Los comienzos del desarrollo de la programación informática coinciden con la aparición de las primeras computadoras en la segunda mitad del siglo XX. La historia de la programación se puede describir a través del desarrollo de los diferentes lenguajes de programación:
- Lenguaje máquina. En este primer período se utilizaban lenguajes máquina muy básicos y limitados basados en el sistema binario (uso de los números 0 y 1 en distintas combinaciones) que es el lenguaje que los ordenadores reconocen, por lo que aún hoy todo lenguaje es convertido a este. Fue reemplazado, ya que resultaba una forma de programación tediosa y difícil.
- Lenguaje ensamblador. Más tarde comenzaron a surgir lenguajes que hacían uso de códigos de palabras. Se utilizaban palabras simples, mnemotécnicas y abreviaturas que tenían su correlativo y eran traducidas al código máquina. El lenguaje ensamblador fue incorporado porque resultaba más fácil de recordar y realizar por el usuario que el código máquina.
- Lenguaje de alto nivel. A finales de la década del 50 surgió el Fortran, un lenguaje de programación desarrollado por IBM que dio inicio a la aparición de lenguajes basados en conjuntos de algoritmos mucho más complejos. Estos lenguajes se adaptaban a distintos ordenadores y eran traducidos por medio de un software al lenguaje de máquina.

Para saber más: https://prezi.com/xk2p2mtpu6xo/evolucion-de-la-programacion-con-paradigmas/

Elementos de la programación

Existen ciertos elementos que son clave a la hora de conocer o ejecutar un lenguaje de programación, entre los más representativos están:

- Palabras reservadas. Palabras que dentro del lenguaje significan la ejecución de una instrucción determinada, por lo que no pueden ser utilizadas con otro fin.
- Operadores. Símbolos que indican la aplicación de operaciones lógicas o matemáticas.
- Variables. Datos que pueden variar durante la ejecución del programa.
- Constantes. Datos que no varían durante la ejecución del programa.
- Identificadores. Nombre que se le da a los diferentes elementos para identificarlos.

2: El lenguaje de programación Java

Java

- Java ha existido desde hace más de veinte años.

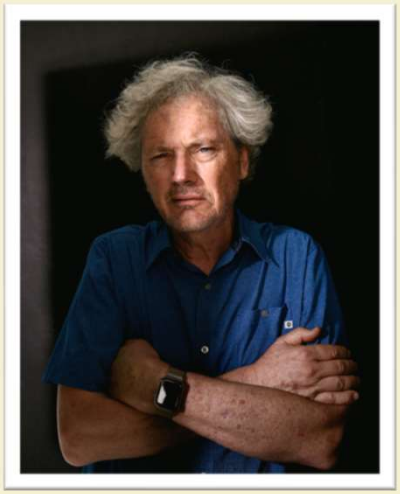
  Fue creado en 1995 por la empresa Sun Microsystems.

  Esta empresa fue comprada por Oracle en 2009.

- ¿Qué es Java?
  - Ante todo, es un lenguaje orientado a objetos (POO). Por lo tanto, el lenguaje Java permite escribir programas.
  - En segundo lugar, es una plataforma de desarrollo. Está compuesta por un conjunto de bibliotecas, un conjunto de especificaciones (JSR - Java Specification Requests) que describen los diferentes API que forman la plataforma y un conjunto de herramientas para compilar, ejecutar, documentar, etc., sus programas. El conjunto forma el JDK (Java SE Development Kit).

¿Por qué Java?

- Bill Joy, ingeniero de Sun Microsystems, y su equipo de investigadores trabajaban en el proyecto «Green», que consistía en desarrollar aplicaciones para sistemas embebidos (en particular, teléfonos móviles y televisores interactivos). Convencidos de las ventajas de la programación orientada a objetos (POO), optaron por desarrollar en C++, que ya había demostrado sus capacidades.
- Pero, para este tipo de proyecto, C++ mostró pronto sus lagunas y sus límites. En efecto, aparecieron numerosos problemas de incompatibilidad con las diferentes arquitecturas físicas (procesadores, tamaño de memoria) y los sistemas operativos encontrados, así como también a nivel de la adaptación de la interfaz gráfica de las aplicaciones y de la interconexión entre los diferentes dispositivos.
- Debido a las dificultades encontradas con C++, era preferible crear un nuevo lenguaje basado en una nueva plataforma de desarrollo. Dos desarrolladores de Sun, James Gosling y Patrick Naughton, se pusieron manos a la obra.



Bill Joy, Cofundador de Sun Microsystems, creador del SSOO Unix en sus años universitarios e inspirador del lenguaje Java.

¿Por qué Java?

- La creación de este lenguaje y plataforma se inspiró en las interesantes funcionalidades propuestas por otros lenguajes tales como C++, Eiffel, Smalltalk, Objective C, Cedar/ Mesa, Ada, Perl. El resultado es una plataforma y un lenguaje idóneos para el desarrollo de aplicaciones seguras, distribuidas y portables en numerosos periféricos y sistemas embebidos interconectados en red, y también en Internet (clientes ligeros), así como en estaciones de trabajo (clientes pesados).
- Llamado originalmente Green Talk, más tarde OAK (un nombre ya utilizado en informática), Gosling lo bautizó finalmente como Java, la isla que produjo por primera vez café, debido a las cantidades de café tomadas por los programadores y diseñadores del lenguaje. Y así, en 1991, nació el lenguaje Java, que en 1995 se consolidaría.
- En 2009, Sun Microsystems cayó en banca rota y Oracle compró los derechos de Java.

ORACLE

Java

- En realidad, hay varias plataformas.
  - La plataforma básica se llama Java SE (Java Standard Edition) como se indicó anteriormente. Cumple con la mayoría de las necesidades.
  - La segunda plataforma es la plataforma Java EE Su objetivo es permitir la creación de aplicaciones distribuidas y, en particular, aplicaciones web. Esta plataforma se basa en la plataforma Java SE, pero también en software de terceros, los servidores de aplicaciones. Desde septiembre de 2017 y la versión Java EE 8, la plataforma se ha vendido a la fundación Eclipse. La plataforma ahora se llama Jakarta EE y la primera versión estuvo disponible en 2019.
  - La última plataforma se llama Java Embedded . Estrictamente hablando, no es una plataforma única, sino más bien un conjunto de plataformas. Permiten realizar aplicaciones ligeras que se pueden ejecutar en dispositivos integrados. También se usan en el Internet de las cosas. Es posible citar de manera más particular las plataformas Java TV y Java Card. Estas son una especie de adaptaciones ligeras de la plataforma Java SE.

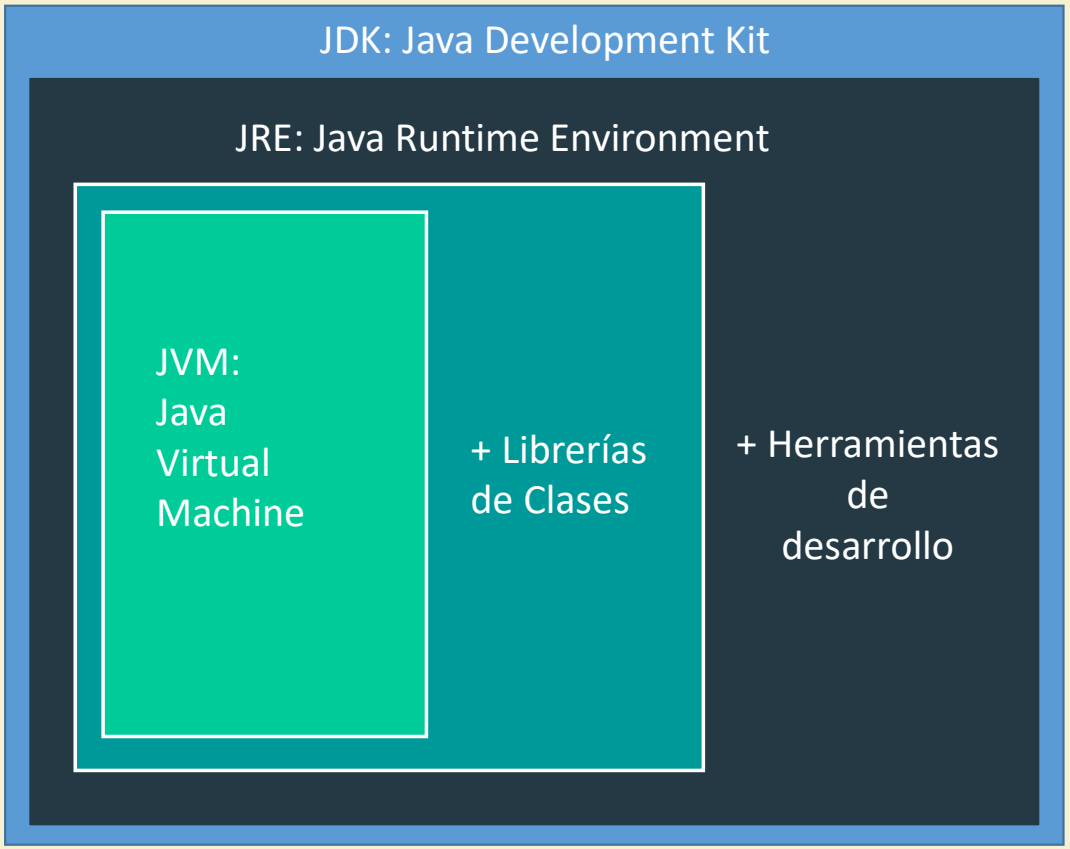
Java

- Finalmente, es un entorno de ejecución (JRE - Java Runtime Environment).

  Permite ejecutar programas a través de la máquina virtual de Java (JVM - Java Virtual Machine). Se requiere la instalación de JRE en máquinas que ejecutan un programa desarrollado en Java.

- Un desarrollador solo necesita instalar el JDK en su puesto de trabajo porque incorpora el JRE.

Java: Arquitectura JDK



JDK = JRE + Herramientas de Desarrollo
JRE = JVM + Librerías de Clases

Entonces... ¿Cómo funciona Java?

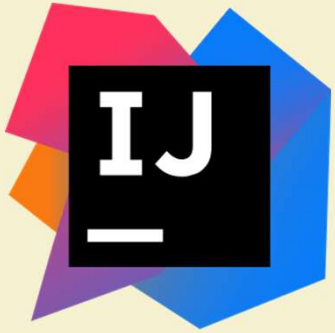
- Independiente de la plataforma: El compilador construye un código neutro “bytecode” que no dependía del tipo de CPU, el cual se ejecutaba sobre una “máquina virtual” denominada Java Virtual Machine (JVM):
- Los programas Java compilados sólo se pueden ejecutar sobre el procesador virtual JVM
- Por ello un mismo código Java compilado se puede ejecutar sobre cualquier plataforma, con sólo cargar previamente un emulador de JVM; este proceso se llama “interpretación”.
- El compilador traduce instrucciones Java de alto nivel a instrucciones JVM de bajo nivel.
- El intérprete traduce instrucciones de JVM a instrucciones de un procesador concreto (p. e. Intel Core), y las ejecuta.

3: Instalación de nuestro IDE

¿Qué es un entorno de desarrollo?

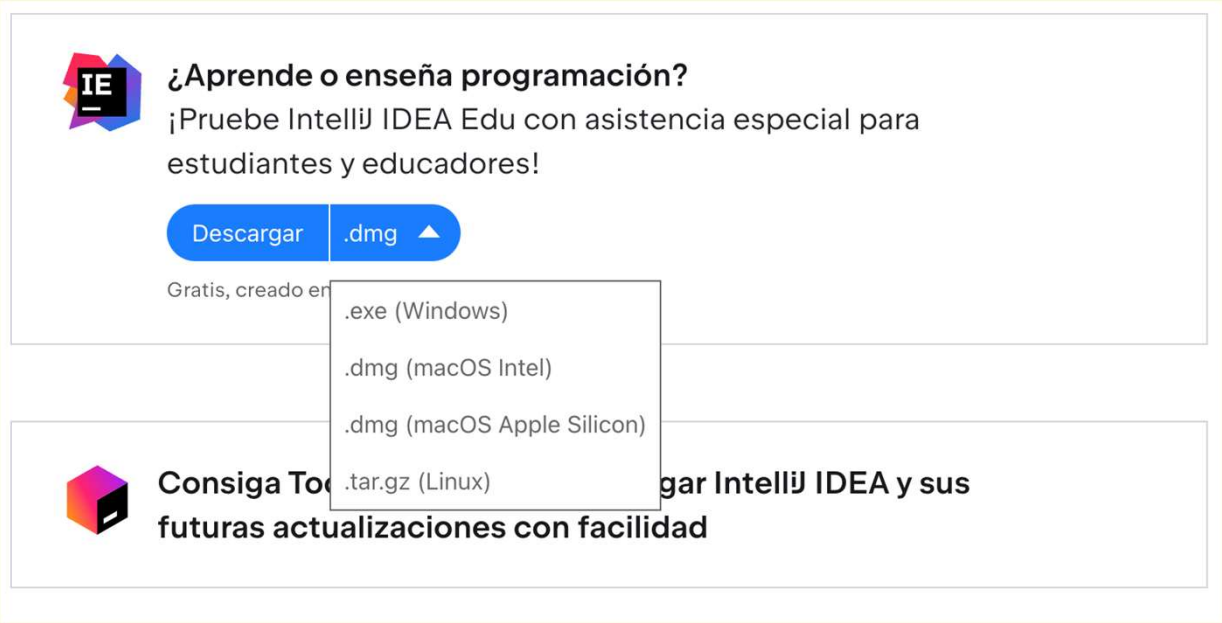
- Un entorno de desarrollo integrado (**IDE**) es una aplicación de software que ayuda a los programadores a desarrollar código de software de manera eficiente. Aumenta la productividad de los desarrolladores al combinar capacidades como editar, crear, probar y empaquetar software en una aplicación fácil de usar.
- El nuestro va a ser IntelliJ IDEA Edu de la compañía JetBrains.
- Antes de descargarlo debemos tener nuestro JDK instalado:

https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html



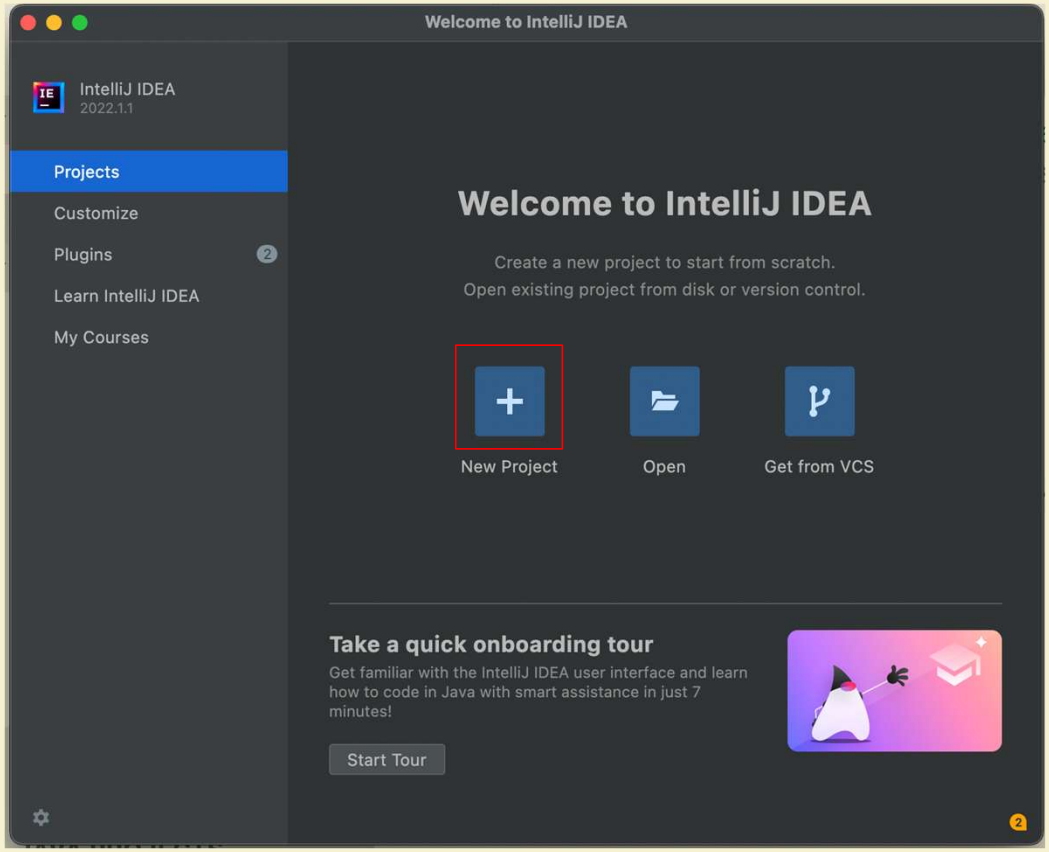
Instalación del entorno de desarrollo

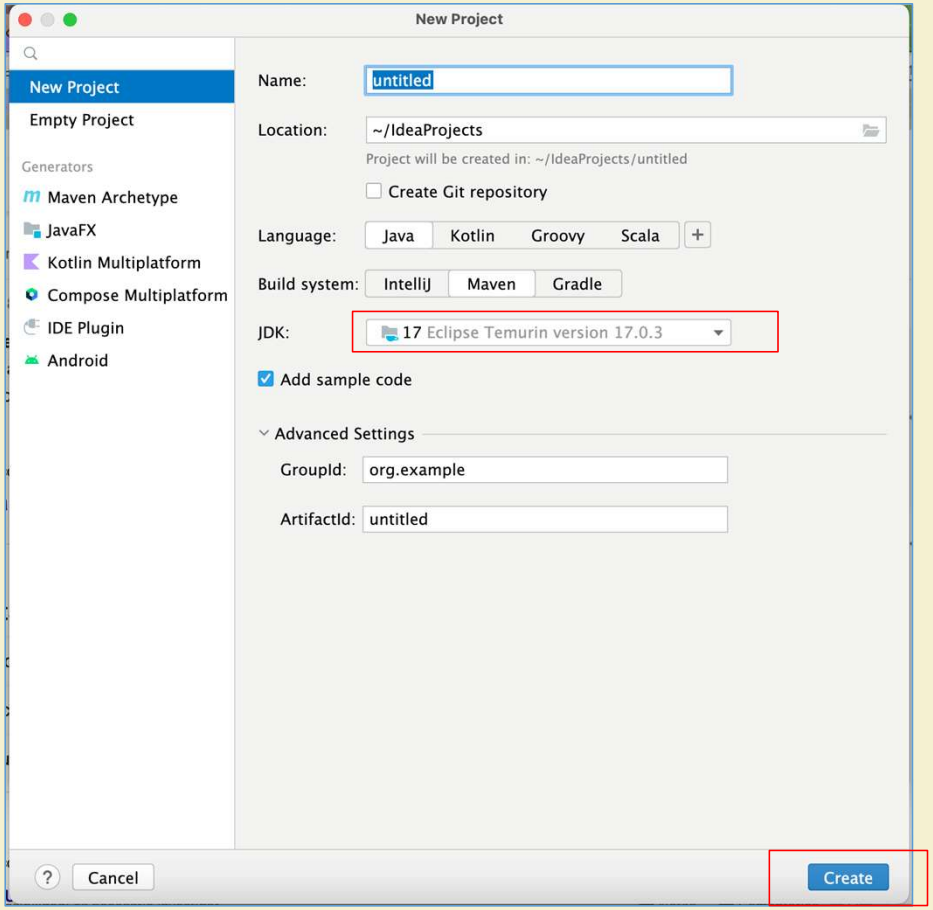
- https://www.jetbrains.com/es-es/idea/download/



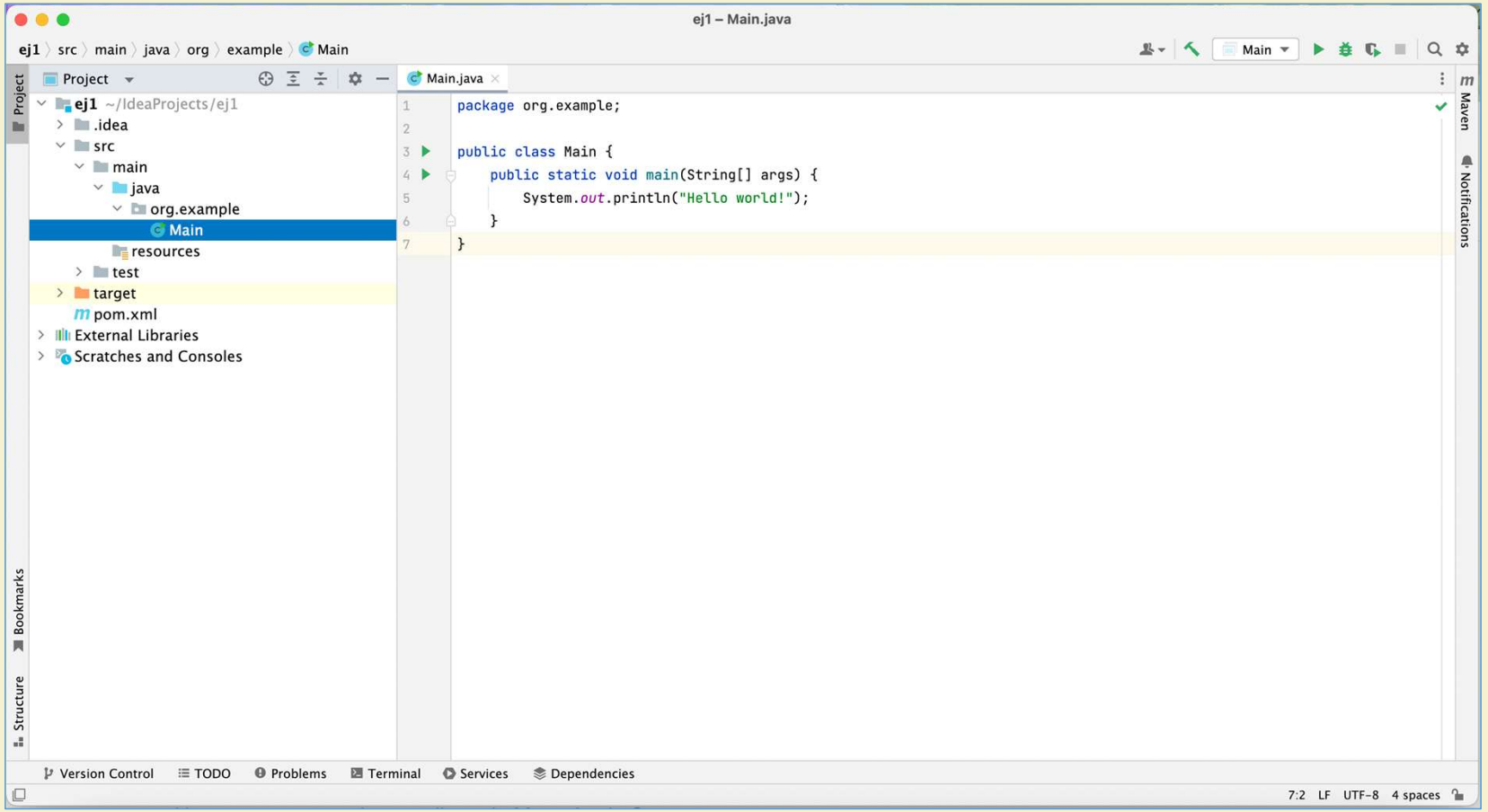
Y seguir los pasos de la instalación de forma clásica ... (Siguiente... Siguiente...Acepto, etc.)

Una vez instalado....
¡Vamos a crear nuestro primer proyecto!





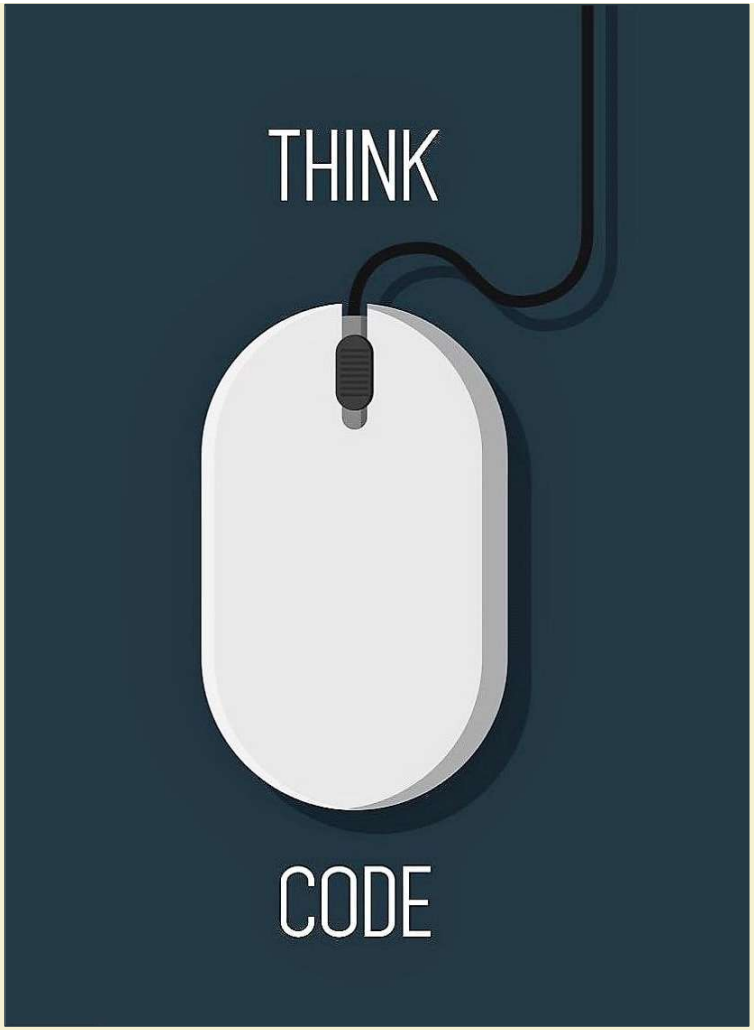
Una vez instalado....
¡Vamos a crear nuestro primer proyecto!



¿Preguntas?

02

Unidad Didáctica 2: Introducción al lenguaje Java - Fundamentos



Módulo: Programación

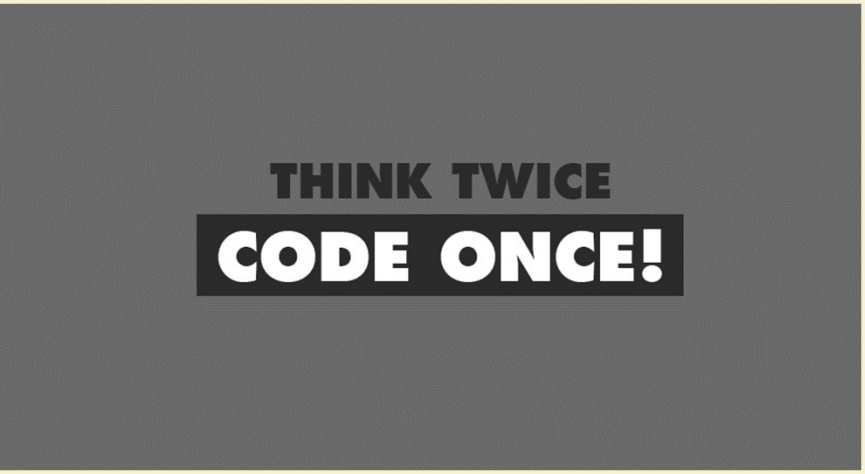
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

Índice



Elementos del Lenguaje

Identificadores

- Palabras que se usan para designar nombres de entidades (clases, métodos, atributos, variables, etc.).
- Debe comenzar con una letra, "_" o "$", seguido de letras, dígitos o ambos.
- Distingue entre mayúsculas y minúsculas.
- Como excepción, no se recomienda utilizar caracteres internacionales en los nombres de clases (por su correspondencia con el archivo).
- No puede incluir espacios.
- No puede coincidir con ninguna palabra reservada del lenguaje.
- *null*, *true* y *false* son formalmente literales, pero ningún identificador puede coincidir con ellos.
- Reglas de Estilo:

Tipo de identificador	Convención	Ejemplo
Nombre de clase (Tipo)	Comienza por letra mayúscula	String, Integer, Figura
Método	Comienza por letra minúscula	calcularArea, getAño
Variable	Comienza por letra minúscula	area, año
Constante (Variable static y final)	Todas en letras mayúsculas	PI, MAX

Elementos del Lenguaje

Palabras Reservadas

- Palabras propias del lenguaje con un significado especial. Por ejemplo, if, while, etc.

Separadores:

Separador	Descripción
()	Listas de parámetros, precedencia en las expresiones y conversiones de tipo.
{}	Inicializar matrices, bloque de código, clases, métodos y ámbitos locales.
[]	Declarar arrays y acceder sus elementos.
;	Separa sentencias.
,	Separa identificadores consecutivos.
.	Separar nombres de paquetes de subpaquetes y clases. Acceso a atributos o métodos.

Comentarios:

- // Comentario de una sola línea
- /* Comentario de más de una línea */
- /**Comentario de más de una línea para procesamiento de Java*/

Tipos de Datos

Objetos

- En Java, todos los elementos son objetos, excepto los tipos primitivos. Existen clases de envoltura que envuelven como objetos tipos primitivos.

Tipos Primitivos:

Tipo	Descripción
boolean	true o false
char	Carácter Unicode 1.1.5, 16 bits
byte	Entero de complemento a dos con signo, 8 bits
short	Entero de complemento a dos con signo, 16 bits
int	Entero de complemento a dos con signo, 32 bits
long	Entero de complemento a dos con signo, 64 bits
float	Número en coma flotante IEEE 754-1985, 32 bits
double	Número en coma flotante IEEE 754-1985, 64 bits

Tipos de Datos: Clases Envoltura

- Para cada uno de los tipos de datos básicos, Java proporciona una clase que lo representa. A estas clases se las conoce como clases de envoltorio, y sirven para dos propósitos principales:
- Encapsular un dato básico en un objeto, es decir, proporcionar un mecanismo para "envolver" valores primitivos en un Objeto para que los primitivos puedan ser incluidos en actividades reservadas para los objetos
- Proporcionar un conjunto de funciones útiles para los primitivos.
- Hay una clase de envoltorio para cada primitivo.

Tipo	Clase de envoltura	Argumentos del constructor
boolean	Boolean	boolean o String
char	Character	char
byte	Byte	byte o String
short	Short	short o String
int	Integer	int o String
long	Long	long o String
float	Float	float, double o String
double	Double	double o String

Tipos de Datos

Tipos de Objeto definidos por el usuario

- Deberán construirse a través de las Clases. No permite crear tipos (sí variables) fuera del sistema de clases.

Cambios de tipos a través de Casting

- Java es un lenguaje fuertemente tipado. La conversión de tipos no es automática, sino explícita."
- Ejemplo:

```
double d = 1.0;
int i = (int) d;
Persona p = new Persona("Rafael Nadal");
Deportista tenista = (Deportista) p;
```

Literales

Un literal es la representación textual de un valor de un tipo.

- Referencias de objeto. Son variables de tipo Clase o Interface. La única referencia literal a un objeto es null.
- Booleanos (boolean). Los literales sólo son dos: true y false.
- Enteros (byte, short, int, long). Los literales son cadenas de dígitos decimales, octales o hexadecimales. Por ejemplo, los números siguientes tienen todos el mismo valor: 29 035 0x1D 0X1d. Por defecto las constantes enteras son de tipo int, para que sean de tipo long deben terminar en L o l, como 29L. La conversión del tipo int a short o byte es implícita si su valor está dentro del rango válido.

Literales

- Reales - Coma flotante (float, double). Los literales son números decimales con coma decimal opcional, seguidos opcionalmente por un exponente. Los siguientes literales denotan el mismo número: 18. 1.8e1. 18E2. Debe estar presente como mínimo un dígito (no e2). Seguido de f o F denota precisión simple (float). Seguido de d o D denota doble precisión (double). Los literales en coma flotante son de tipo double por defecto.
- Caracteres (char). Los literales van entre comillas sencillas, como 'Q'. También código Unicode: \ddd (octal), \udddd (hexadecimal). Ciertos caracteres especiales se pueden representar mediante una secuencia de escape como: \n (retorno de línea), \t (tabulador), \b (retroceso), \f (salto pág), \\, \', \".
- Cadenas de caracteres. No son tipos primitivos (objeto de la clase String). Aparecen entre comillas dobles: "cadena de caracteres". Todas las secuencias de escape para los literales de carácter son válidas aquí.

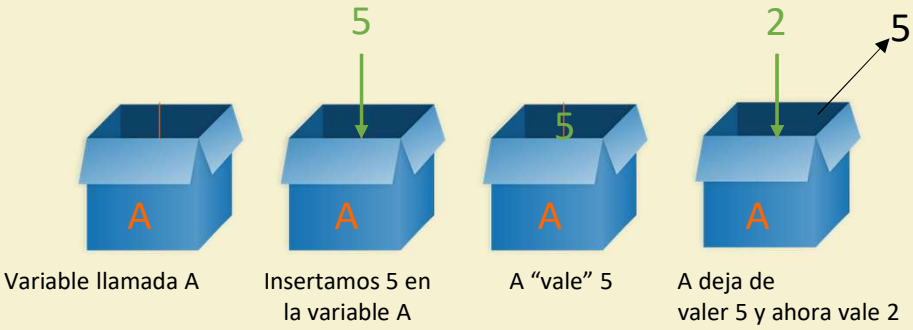
Variables

En Java, el concepto de variable es el mismo que en otros lenguajes de programación: *Instancias de un tipo, es decir, un espacio de almacenamiento con un nombre que pueden cambiar su valor*.

Pueden alojar valores (tipos primitivos) o referencias a objetos (clases o interfaces).

Variables de valor:

- Almacenan un valor de tipo primitivo.
- Sigue la sintaxis usual.



Por eso se llaman variables, porque su valor puede variar

Variables

Variables referencia

- Almacenan la referencia (no el valor) de un objeto o el literal null si no apunta aún a ningún objeto.
- Se declaran y asignan normalmente a la vez, invocando al constructor que reservará memoria para el Objeto mediante el operador new.
- No se puede referenciar a tipos primitivos.
- Las únicas operaciones válidas con referencias son :
  - asignación =
  - comparación con == y !=
  - String el operador + para concatenar.
  - Castings
  - instanceof
  - Operador .(punto)

Variables

Declaración y Asignación

- **Sintaxis:**
  - [<modificador>] <tipo> <lista de identificadores> [= valor_ini];
  - Se pueden declarar en cualquier punto del código, pero siempre antes de usarse.
  - Sólo se podrán usar hasta la finalización del bloque en la que son declaradas.
  - Si no se le asigna ningún valor inicial, toman un valor por defecto.
  - Podemos modelar una constante como una variable o atributo final.
- Ejemplos:

```
boolean esTemprano,haceSol = false, true;
long sueldo = 100000L;
String nombre = "Raúl";
Empleado e = new Empleado("María González", "23234256L");
final float PI = 3.14F; // Esto es una constante
```

Constantes

- En una aplicación, con frecuencia sucede que se utilizan valores numéricos o cadenas de caracteres, que no se modificarán durante el funcionamiento de la aplicación. Para facilitar la lectura del código, es recomendable definir estos valores en forma de constantes.
- La definición de una constante se realiza agregando la palabra clave final antes de la declaración de una variable. Es obligatorio inicializar la constante en el momento de su declaración (este es el único lugar donde es posible hacer una asignación a la constante).
- Las reglas relativas a la vida útil y el alcance de las constantes son idénticas a las relativas a las variables.
- El valor de una constante también se puede calcular a partir de otra constante.
- Muchas constantes ya están definidas a nivel del lenguaje Java. Se definen como miembros static de las muchas clases del lenguaje. Por convención, los nombres de las constantes se escriben completamente en mayúsculas.

```
public class Producto {

   final double PORCIVA = 1.21;

   final int TOTAL = 100;
   final int MEDIO = TOTAL / 2;
                  . . .

}
```

Enumeraciones

- Una enumeración nos permitirá definir un conjunto de constantes que están funcionalmente vinculados entre sí. La declaración se realiza de la siguiente manera:

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

- El primer valor de la enumeración se inicializa a cero. Las siguientes constantes se inicializan con un incremento de uno. Por lo tanto, la declaración anterior se podría haber escrito :

```
public class Dia {
   public static final int DOMINGO = 0;
   public static final int LUNES = 1;
   public static final int MARTES = 2;
   public static final int MIERCOLES = 3;
   public static final int JUEVES = 4;
   public static final int VIERNES = 5;
   public static final int SABADO = 6;
}
```

Operadores y Aritmética

Operadores generales

- **Aritméticos:** +, -, *, /, %
- **Relacionales:** <, <=,>, >=, !=, ==
- **Lógicos:** !, &&, ||
- **Aritméticos con Asignación e incremento (sólo a enteros y a char):** +=, -=, *=, /=, ++ postfijo, - - postfijo
- **Otros Operadores:** Casting, instanceof, new, ””+“” n(concatenación).

Aritmética Entera

- La aritmética de enteros es modular de complemento a dos. Esto quiere decir que si un valor excede el rango de su tipo (int o long) es reducido a su módulo. Por tanto, no hay desbordamientos.
- La división por cero o el módulo (%) por cero no son válidos, elevándose una excepción del tipo ArithmeticException.

Operadores y Aritmética

Aritmética coma flotante

- Java usa el estándar IEEE 754-1985 sobre coma flotante, tanto para la representación como para la aritmética.
- Se pueden producir tanto desbordamientos como subdesbordamientos.
- Existen las constantes infinito y menos infinito, así como una representación denominada NaN (Not a Number) para los resultados de expresiones no válidas.
- Según el estándar, el cero puede ser positivo (0.0) o negativo (-0.0). Ejemplo 1d/0d es infinito --- 1d/-0d es –infinito
- La división por cero en esta aritmética no produce la elevación de ninguna excepción.
- Una constante double no se puede asignar directamente a una variable float aunque el valor de la misma esté en el rango permitido para float (casting explícito). Por ejemplo:

```
double d = 2.2;
float f;
f = d; //Incorrecto: error de compilación
f = (float) d;  //Correcto: casting explícito
```

Expresiones

Una expresión es una combinación bien formada de variables, literales y operadores que devuelven un valor de un determinado tipo.

- Reglas de precedencia. Determinan el orden de aplicación de los operadores en una expresión.
- Reglas de Asociatividad. Determina el sentido de aplicación en caso de igualdad de prioridad.
- Si usamos paréntesis de modo explícito, estamos indicando nuestras propias reglas de precedencia y asociatividad. Por ejemplo:

x*y+3/z →(x*y)+(3/z)

x*y+3/z →x*(y+3)/z

Expresiones

- Tabla de Precedencia y Asociatividad

Precedencia	Operador	Asociatividad
+	! ++ -- - (unario)	derecha a izquierda
	* / %	izquierda a derecha
	+ -	izquierda a derecha
	< <= > >=	izquierda a derecha
	== !=	izquierda a derecha
	&&	izquierda a derecha
-	||	izquierda a derecha

Conversión de Tipos

Una expresión siempre es del tipo del operando de mayor rango:

```
int a = 5;
double b = 3.2;
double suma = a + b; //expresión de tipo double
```

Conversiones implícitas de tipos:

- De char a int
- De entero a coma flotante
- De entero a entero de mayor rango
- Las contrarias producen pérdidas de información.
- De Tipo/Objeto a String: cuando un operador + tenga al menos un argumento de tipo String.

Conversiones explícitas de tipos (Casting):

- Cuando no se pueda asignar implícitamente un tipo a otro se puede pedir una conversión explicita.
- (tipo) expresión →`int x = (int)b;`

Conversión de Tipos

- Ejemplos:

```
int entero, a = 5, b = 2;

double real = 3.20567;

entero = real; //entero vale 3

real = a/b; // real vale 2

real = (double)a/b; // real vale 2.5

float resultado = (float) retornaInt(); //Cualquier método que devuelva un entero
```

Estructura básica de un programa Java

- Un programa en Java debe incluir una y sólo una clase pública, que da nombre al fichero.
- En los primeros programas esta clase incluirá un único método público y estático con el nombre main.
- Si se necesitan importar paquetes enteros o partes de los mismos, se indicará al comienzo del fichero, antes de cualquier comentario con la sentencia *import*.
- El esquema general es el que se muestra en la imagen:

```
import ...;
public class NombreClase{
    //<Declaración de atributos>
    //<Métodos>

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

Instrucciones y bloques

Se considera un bloque en Java al conjunto de instrucciones encerradas entre una llave de apertura, { , y su correspondiente llave de cierre, } , en su mismo nivel de anidamiento. Los bloques además de agrupar instrucciones, marcan la vida y visibilidad de las variables que se declaran en él.

Las instrucciones que se consideran en Java son las siguientes:

- Instrucciones de declaración: nos permiten crear nuevas variables u objetos y en algunos casos darles un valor inicial:

```
int x;
double z = 4.6, r;
String nombre = new String("Lola");
```

- Instrucciones de acción:

```
r = z–(10*x);
System.out.println ("La persona se llama" + nombre);
```

- Instrucciones de control: Son las instrucciones que nos permiten dirigir el orden de ejecución. Hay dos tipos:
  - Selectivas o decisión (if, switch)
  - Repetitivas o bucles (for, while, do-while)

Instrucciones de Control

- **Estructura if :**

```
if(condición)instrucción;
```

- Si la condición vale true, la instrucción se ejecuta. La condición debe ser una expresión que, una vez evaluada, debe devolver un valor booleano true o false. Con esta sintaxis, solo la instrucción situada después del if será ejecutada si la condición vale true. Para ejecutar varias instrucciones en función de una condición, debemos utilizar la sintaxis siguiente.

```
if(condición) {
        instrucción 1;
        ...
        instrucción n;
}
```

En este caso, se ejecutará el grupo de instrucciones situado entre las llaves si la condición vale true.

Instrucciones de Control

- También podemos especificar una o varias instrucciones que se ejecutarán si la condición vale false añadiendo un bloque else.

```
if (condición) {
        instrucción 1;
        ...
        instrucción n;
} else {
        instrucción 1;
        ...
        instrucción n;
}
```

Instrucciones de Control

- También podemos anidar los if con los else :

- En este caso, se comprueba la primera condición. Si es verdadera, se ejecuta el bloque de código correspondiente, y si no, se comprueba la siguiente y así sucesivamente. Si no se verifica ninguna condición, se ejecutará el bloque de código definido a continuación del else. La instrucción el seno es obligatoria en esta estructura. Por ello, es posible que no se llegue a ejecutar instrucción alguna si ninguna de las condiciones es verdadera.

```
if (condición) {
        instrucción 1;
        ...
        instrucción n;
} else if(condición) {
        instrucción 1;
        ...
        instrucción n;
} else {
        instrucción 1;
        ...
        instrucción n;
}
```

Instrucciones de Control

- Estructura ternaria:
- Esta estructura particular se corresponde con una declaración if...else. Solo se puede usar para asignar un valor. Es una estructura breve que sustituye de manera ventajosa la instrucción if...else.

```
String mensaje = condición ? "si true" : "si false";
```

- El ? desencadena la evaluación de la condición. Si la condición vale true, entonces es el valor proporcionado después de ? el que se vuelve a activar; de lo contrario, es el valor proporcionado después de : es el que se devuelve.

```
if(condición)
        mensaje = "si true";
else
        mensaje = "si false";
```

Ejemplos:

```
if (x == 1)
                System.out.println("x vale 1");
```

```
if (x%2==0){
   System.out.println ("x es par");
   y = x/2;
}
```

```
if (x==1)
   System.out.println ("x vale 1");
else
   System.out.println ("x no vale 1");
```

```
if (x%2==0){
   System.out.println(x + " es par");
   y = x/2;
}else{
   System.out.println(x + " es impar");
   y = x+1;
}
```

```
if (x==1)
   System.out.println("x vale 1");
else if(x==2)
   System.out.println("x vale 2");
else if (x==3)
   System.out.println("x vale 3");
else
   System.out.println("x no vale 1,2 ni 3");
```

```
if (x==1){
   System.out.println("x vale 1");
   x++;
}else if (x==2){
   System.out.println("x vale 2");
   x = x + 2;
}else{
   System.out.println("x no vale 1 ni 2");
   x = 0;
}
```

Instrucciones de Control

- **Estructura switch:**

- La estructura switch permite un funcionamiento equivalente, pero ofrece una mejor legibilidad del código. La sintaxis histórica es la siguiente:
- El valor de la expresión se evalúa al principio de la estructura (por el switch) y, a continuación, se compara el valor obtenido con el valor especificado en el primer case.
- Si los dos valores son iguales, entonces el bloque de código adyacente se ejecuta.
- En caso contrario, se compara el valor obtenido con el valor del case siguiente. Si hay correspondencia, se ejecuta el bloque de código y así sucesivamente hasta el último case.
- Si no se encuentra ningún valor concordante en los diferentes case, se ejecuta el bloque de código especificado por la palabra clave default. Cada uno de los bloques de código debe terminarse con la instrucción break. Si no es el caso, la ejecución continuará por el bloque de código siguiente hasta encontrar una instrucción break o hasta el fin de la estructura switch. Esta solución se puede utilizar para poder ejecutar un mismo bloque de código cuando coinciden distintos valores.

```
switch (expresión) {
        case valor1:
                instrucción 1;
                ...
                instrucción n;
                break;
        case valor2:
                instrucción 1;
                ...
                instrucción n;
                break;
        default:
                instrucción 1;
                ...
                instrucción n;
}
```

Ejemplo Switch clásico:

```
switch (x){
   case 1: System.out.println("x vale 1");
      x++;
      break;
   case 2: System.out.println("x vale 2");
      x = x + 2;
      break;
   default: System.out.println("x no vale ni 1 ni 2");
      x = 0;
}
```

Instrucciones de Control

- El valor que se debe comprobar puede estar contenido en una variable, pero también puede ser el resultado de un cálculo. En tal caso, solo se ejecuta el cálculo una única vez al principio del switch. El tipo del valor probado puede ser numérico, entero, carácter, cadena de caracteres o enumeración. Por supuesto, es necesario que el tipo de la variable comparada corresponda al tipo de los valores en los diferentes case.
- Si la expresión es de tipo cadena de caracteres, se utiliza el método equals para verificar si es igual a los valores de los distintos case. La comparación hace por tanto distinción entre mayúsculas y minúsculas.

```
System.out.println("Responda sí, no o cualquier otra cosa:");
BufferedReader br;
br = new BufferedReader (new InputStreamReader(System.in));
String respuesta = "";
respuesta = br.readLine();

switch (respuesta){
   case "s":
   case "S":
                System.out.println("respuesta positiva");
                break;
   case "n":
   case "N":
                System.out.println("respuesta negativa");
                break;
   default:
                System.out.println("respuesta errónea");}
```

Instrucciones de Control

- Estructura switch nueva generación:
- La estructura switch histórica tenía algunas carencias:
- Obligación de repetir la palaba clave case para cada valor a evaluar.
- Obligación de utilizar la palabra clave break para salir del switch.
- Imposibilidad de inicializar simplemente una variable de esta estructura.
- La nueva versión de switch aborda estos problemas. A continuación, se presenta la nueva sintaxis de la estructura (la sintaxis anterior, por supuesto, sigue siendo compatible):

```
[TipoRetorno variableRetorno=]
switch (expresión) {
   case valor1, valor2 -> instrucción
   case valor3 -> {
      TipoRetorno valorRetorno;
      instrucción 1,
      ...
      instrucción n
      [yield valorRetorno]
   }
   default -> instrucción
}
```

Instrucciones de Control

A continuación, se presentan algunas novedades:

- Los valores para los que el procesamiento es idéntico, se pueden separar por una coma.
- El operador que separa los valores de las instrucciones a ejecutar, ya no son los dos puntos (:) sino el operador arrow (flecha ->). Se trata de una expresión lambda.
- Si se tiene que realizar varias declaraciones, se deben colocar en un bloque de código delimitado por llaves.
- Opcionalmente, el switch puede devolver un valor. Este valor se corresponde con el resultado de las instrucciones. Si se ejecuta una sola instrucción, el valor devuelto es implícitamente el resultado de la misma. La palabra clave return no es útil. Si hay varias instrucciones y el compilador no detecta el retorno implícito, debe usar la palabra clave yield para indicar el valor a devolver.

Ejemplo Switch “moderno”:

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

```
Dia unDia = Dia.DOMINGO;
```

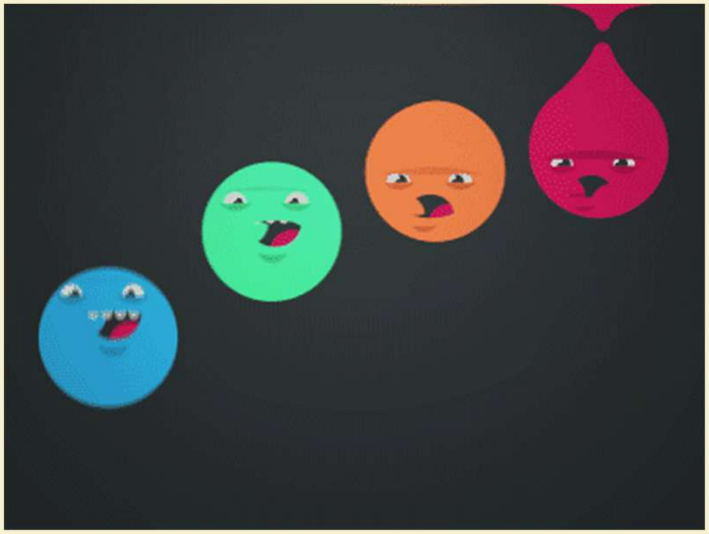
```
switch (unDia) {
   case LUNES, MARTES, MIERCOLES, JUEVES, VIERNES
         -> System.out.println("Estamos durante la semana");
   case SABADO, DOMINGO
         -> System.out.println("Estamos en fin de semana");
}
```

```
int tipoDia =
      switch(unDia) {
         case LUNES, MARTES, MIERCOLES, JUEVES, VIERNES -> 0;
         case SABADO, DOMINGO -> 1;
      };
System.out.println(tipoDia);
```

```
int tipoDiaSiContieneUnaM =
      switch(unDia) {
         case MARTES, MIERCOLES, DOMINGO -> {
            if(unDia==Dia.MARTES || unDia==Dia.MIERCOLES) {
               yield 0;
            }else {
               yield 1;
            }
         }
         default -> -1;
      };
System.out.println(tipoDiaSiContieneUnaM);
```

Bucles: Instrucciones de Control Repetitivas

- **Todas tienen como objetivo ejecutar un bloque de código un determinado número de veces en función de una condición.**
- **Disponemos de tres estructuras:**
  - while (condición)
  - do ... while(condición)
  - for



Bucles: Instrucciones de Control Repetitivas

- Estructura while:
- Esta estructura ejecuta un bloque de manera repetitiva mientras la condición sea verdadera.

```
while (condición)
{
   instrucción 1;
   ...
   instrucción n;
}
```

- La condición se evalúa antes del primer ciclo. Si el resultado es falso en este momento, el bloque de código no se ejecuta. Después de cada ejecución del bloque de código, la condición vuelve a evaluarse para comprobar si es necesario volver a ejecutar el bloque de código. Se recomienda que la ejecución del bloque de código contenga una o varias instrucciones capaces de hacer evolucionar la condición. Si no es el caso, el bucle se ejecutará sin fin. Bajo ningún concepto se debe poner el carácter ;al final de la línea del while porque, en este caso, el bloque de código dejará de asociarse con el bucle.

Bucles: Ejemplos while

```
int i=0;
while (i<=10)  //No hay ; aquí. El bloque adyacente no está
               //relacionado con el bucle
{
  System.out.println(i);
   i++;
}
```

Bucles: Instrucciones de Control Repetitivas

- Estructura do ... while:

```
do{
   instrucción 1;
   ...
   instrucción n;
}while (condición);
```

- Esta estructura tiene un funcionamiento idéntico a la anterior. La diferencia reside en el hecho de que la condición se examina después de la ejecución del bloque de código. Nos permite garantizar que el bloque de código se ejecutará al menos una vez, ya que la condición se comprueba por primera vez después de la primera ejecución del bloque de código. Si la condición es verdadera, el bloque se ejecuta de nuevo hasta que la condición sea falsa. Ten cuidado de no olvidar el punto y coma después del while; en caso contrario, el compilador detectará un error de sintaxis.

Bucles: Ejemplos do...while

```
do
   System.out.println(i++);
while (i<10);
```

```
do{
   System.out.println(i);
   i++;
}while (i<10);
```

Bucles: Instrucciones de Control Repetitivas

- Estructura for:
- Cuando sabemos el número de iteraciones que debe realizar un bucle, es preferible utilizar la estructura for. Para poder utilizar esta instrucción, se debe declarar una variable de contador. Es posible declarar esta variable en la estructura foro en el exterior, y en tal caso se debe declarar antes de la estructura for.
- La sintaxis general es la siguiente:

```
for(inicialización; condición de parada; instrucción de actualización){
   instrucción 1;
   ...
   instrucción n;
}
```

- La inicialización se ejecuta una única vez durante la entrada en el bucle. La condición se evalúa en el momento de la entrada en bucle y, a continuación, con cada iteración. El resultado de la evaluación de la condición de parada determina si el bloque de código se ejecuta. Si el resultado es verdadero, se realiza una nueva iteración del bucle. Después de la ejecución del bloque de código, se realiza la de la iteración. A continuación se comprueba de nuevo la condición y así sucesivamente mientras la condición sea verdadera.

Bucles: Ejemplos for

```
for (int i=1; i<=5; i++)
   System.out.println (i);
```

```
int cont = 0;
for (int i=1; i<=5; i++){
   System.out.println (i);
   cont = cont +1;
}
```

- El ejemplo muestra dos bucles for en acción para visualizar una tabla de multiplicar.

```
int multiplicador;
for(multiplicador=1;multiplicador<=10;multiplicador++){
   for (int tabla = 1; tabla <= 10; tabla++){
      System.out.print(table*multiplicador + "\t");
   }
   System.out.println();
}
```

Bucles: Instrucciones de Control Repetitivas

- Algunos ejemplos:

Factorial con while:

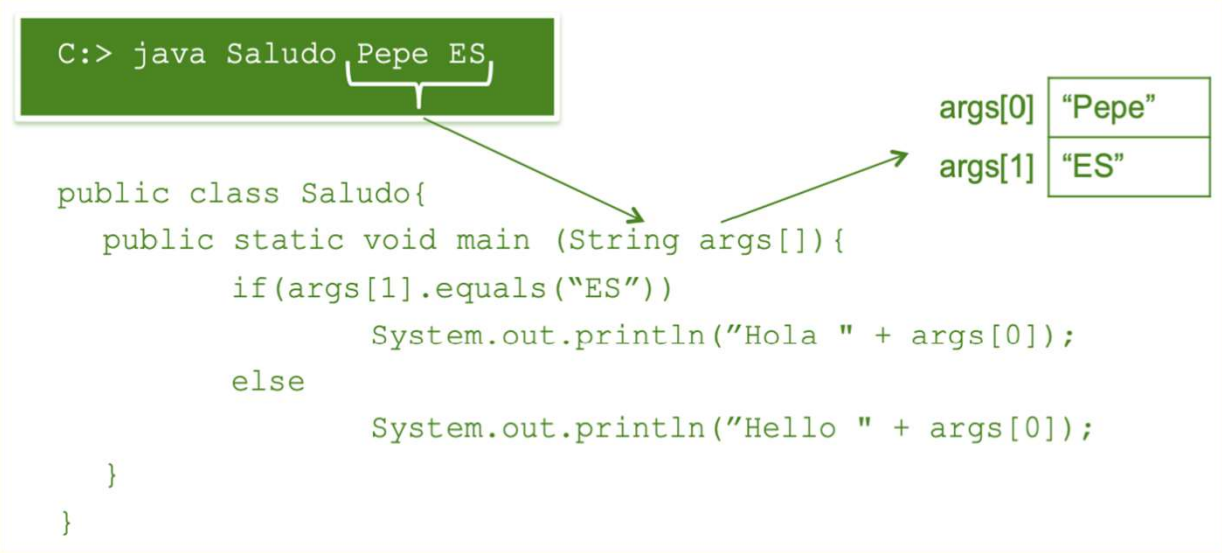
```
int numero = 5;
long resultado = 1;
while (numero > 0) {
   resultado = resultado * numero;
   numero--;
}
System.out.println("El factorial es " + resultado);
```

Factorial con for:

```
int numero = 5, i;
long resultado = 1;
for(i=1;i<=numero;i++)
   resultado = resultado*i;
System.out.println("El factorial es "+resultado);
```

Argumentos de un Programa en Java

- Es posible ejecutar un programa en Java pasándole parámetros de usuario.
- Cada parámetro tras el comando de ejecución del programa se almacena en una tabla de cadenas (Strings)



Argumentos de un Programa en Java

- Si necesitamos usar los parámetros como datos de distinto tipo al String, es necesario transformarlo al tipo adecuado.

```
C:> java Suma 10 20
```

args[0]	“10”
args[1]	“20”

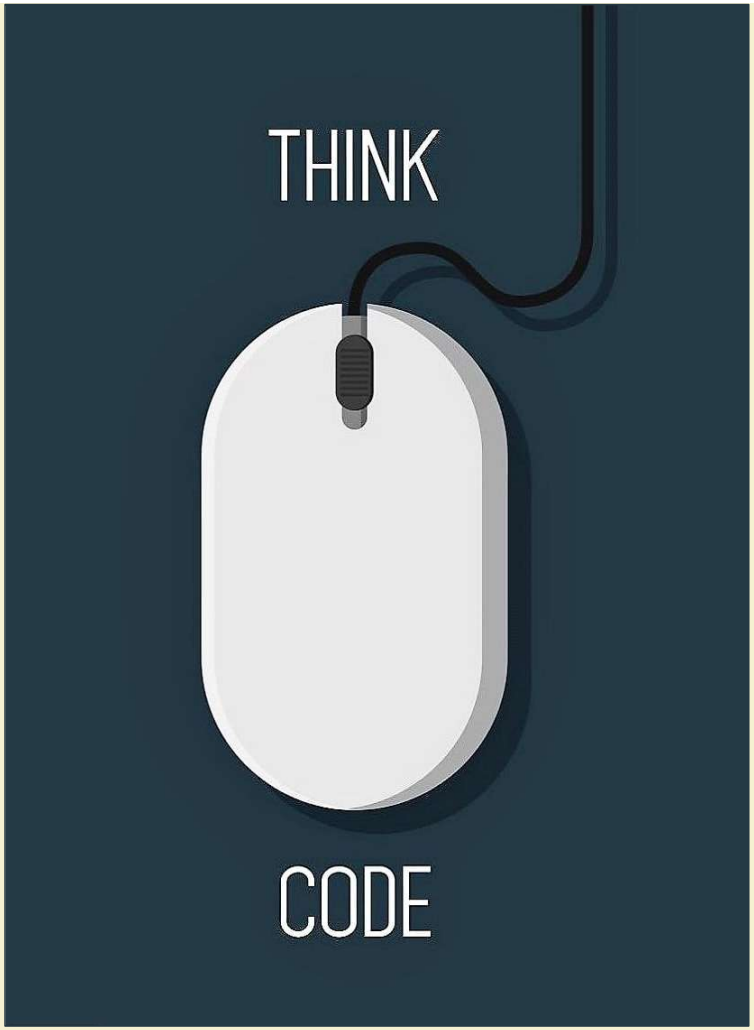
```
public class Saludo{
   public static void main (String args[]){
          int a = Integer.parseInt(args[0]);
          int b = Integer.parseInt(args[1]);
          int suma = a + b;
          System.out.println(”La suma es " + suma);
     }
  }
```

Integer.parseInt
Float.parseFloat
Double.parseDouble

¿Preguntas?

03

Unidad Didáctica 3: Programación Orientada a Objetos (POO)



Módulo: Programación

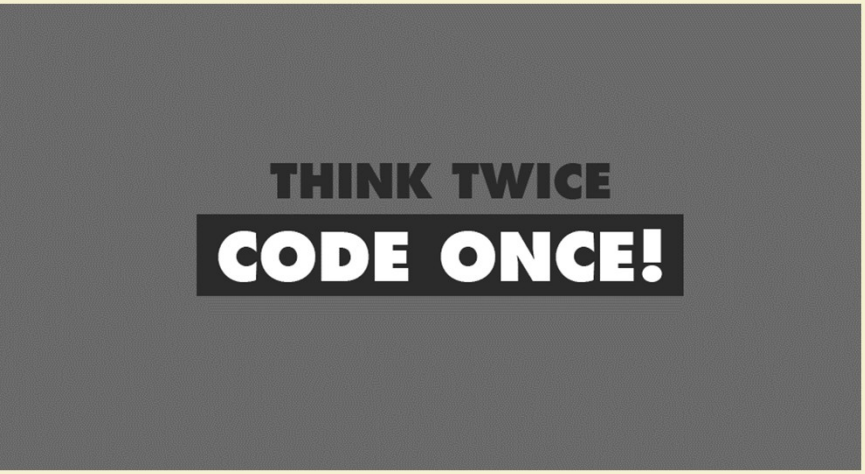
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

Índice



La Programación Orientada a Objetos

¿Programación Orientada a Objetos?

Según Grady Booch(creador de la empresa Rational Software), la POO se define como:

"Un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unidas mediante relaciones de herencia."

Tres conceptos importantes:

- Utiliza objetos, no algoritmos (secuencia de pasos lógicos que permiten solucionar un problema), como bloques básicos lógicos de construcción.
- Las clases definen la información y comportamiento de los objetos, mientras que éstos son instancias concretas de una clase.
- Existe una relación de herencia entre las clases.

- Si falta alguno de estos tres elementos, no es un programa orientado a objetos.

Objeto

El objeto es el concepto fundamental de la POO.

Un objeto en Java no es ni más ni menos que un objeto en el mundo real.

Una tienda online podría tener objetos como: carrito de la compra, cliente, producto...

Los objetos tienen propiedades/campos a los que llamamos atributos, que nos dan información sobre el estado de los mismos.

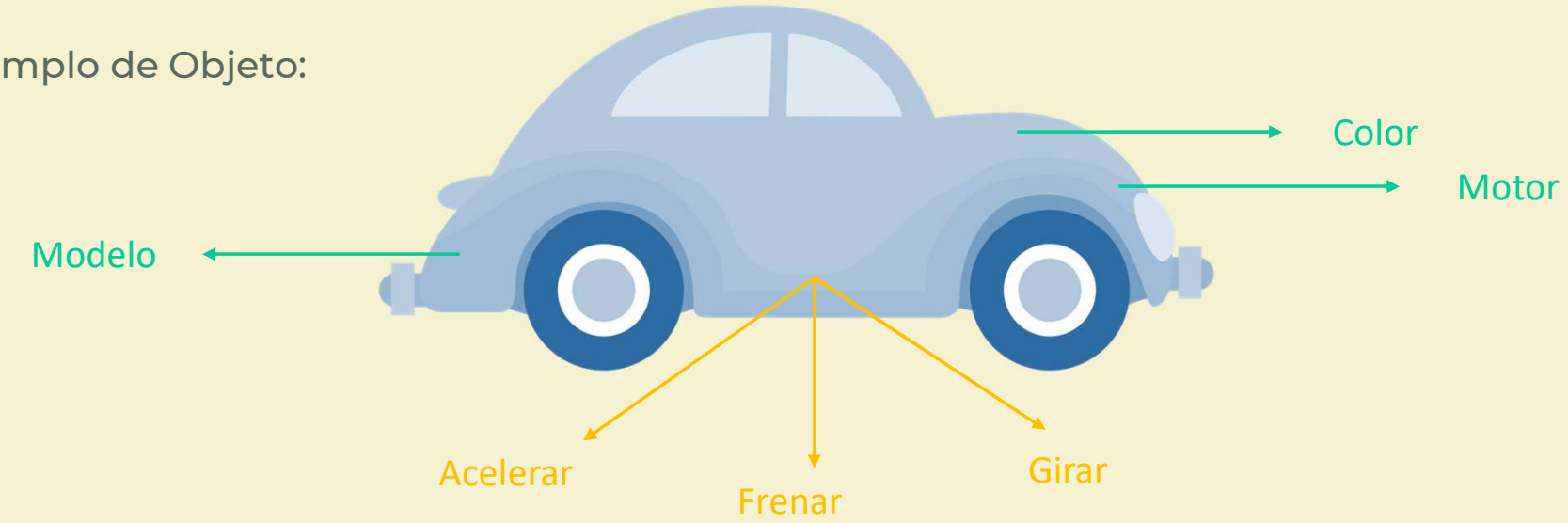
Los objetos también tienen funciones/métodos que nos dan la posibilidad de que se puedan realizar acciones.

- Atributos: información del objeto que define su estado
- Métodos: Comportamiento del objeto = Cambios de Estado

Un ejemplo de objeto es una persona. Una persona tiene muchos atributos, muchas características distintas que pueden ser guardadas en variables. Además de tener ciertas cualidades, una persona también puede realizar tareas (comer, dormir, trabajar...). Ahí entran los métodos o funciones.

Objeto

- Ejemplo de Objeto:



- Los objetos tienen estado y comportamiento.

Objeto

- Estado:

El objeto tiene campos/atributos que lo conforman. Son aquellos que nos dan una idea de las características del objeto. Si seguimos con el ejemplo del coche tenemos: color, modelo, motor, rueda.

- Comportamiento:

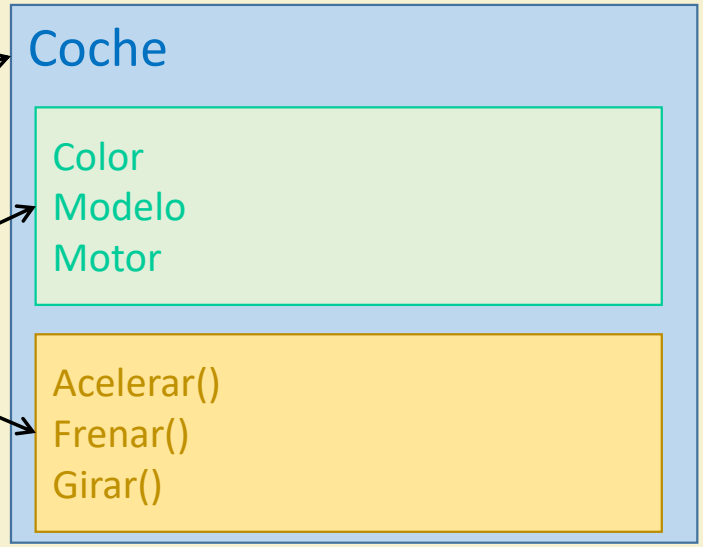
El objeto tiene métodos que nos dan la idea de lo que puede hacer. Ir coche puede arrancar, frenar, acelerar, girar, entre otras cosas.

Clase

Podemos decir que una Clase la definición de la "forma" del objeto. En otras palabras, la Clase como sería el molde, los planos, la definición de un tipo concreto de objetos.

Por ejemplo: Clase Coche

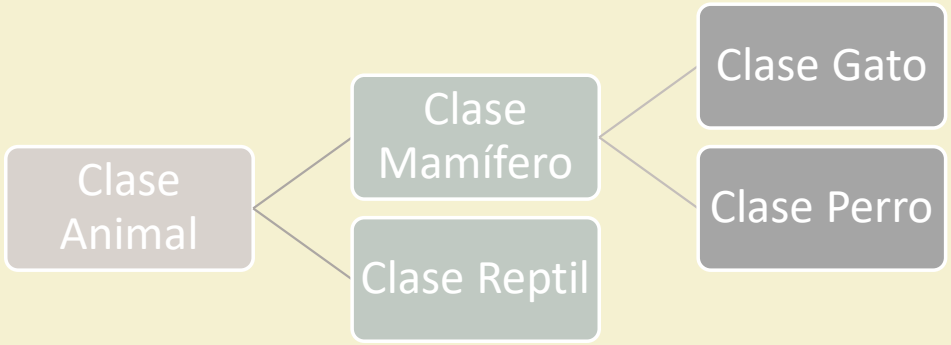
- Nombre de la Clase
- Atributos: definición de los datos
- Métodos: def. del comportamiento



- Los objetos son por tanto instancias, es decir, representaciones concretas de una Clase.

Herencia

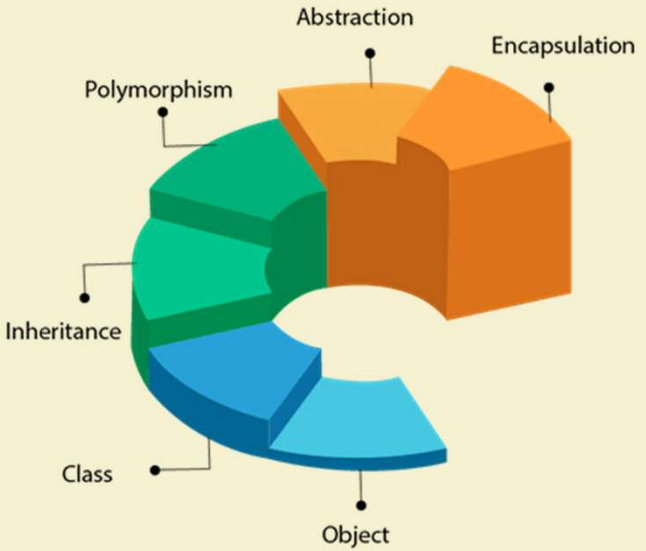
- Los sistemas orientados a objetos permiten definir clases que tienen características de otras clases.



- Un Perro es un determinado tipo de Mamífero, y hereda de ella todas las características de los mamíferos.
- A su vez, Mamífero es un determinado tipo de Animal, y hereda de ella las características de los animales.
- Animal es la Clase Padre o Superclase de Mamífero y Reptil. A su vez, Mamífero es la Superclase de Perro y Gato.

Programación OO y Lenguaje OO

- La Programación Orientada a Objetos es un paradigma, una “filosofía”, un modelo de programación.



- Un Lenguaje Orientado a Objetos es un lenguaje de programación que permite el diseño de aplicaciones orientadas a objetos.

</> Trabajando con Clases

Entonces... ¿Qué es una clase?

- Una clase es un tipo de datos que define la estructura (atributos y métodos) de los objeto de dicho tipo.
- Una clase es la definición de todos los elementos de que está hecho un objeto.
- Una clase es una construcción que se utiliza como un modelo (o plantilla) para crear objetos de ese tipo.
- Una clase es el conjunto de definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.

Ejemplos (Notación UML)

Clase Pokémon

Pokémon

- nombre: String
- tipo: String
- subtipo: String
- salvaje: Boolean

+ atacar()
+ defender()

Nombre de la clase

Atributos: definición de los datos

Métodos: definición del comportamiento

Referencia (variable)

Objetos de la clase Pokémon

p1:Pokémon

- nombre: “Pikachu”
- tipo: “Eléctrico”
- subtipo: null
- salvaje: false

p2:Pokémon

- nombre: “Togepi”
- tipo: “Hada”
- subtipo: null
- salvaje: false

p3:Pokémon

- nombre: “Woobat”
- tipo: “Psíquico”
- subtipo: “Volador”
- salvaje: true

Ejemplos (Notación UML)

Clases

Futbolista

- id: Integer
- nombre: String
- apellido: String
- edad: Integer
- dorsal: Integer
- posición: String
- equipo: String

+ jugarPartido()
+ entrenar()

Bombilla

- marca: String
- encendida: Boolean
- potencia: Integer

+ encender()
+ apagar()

Objetos

b1:Bombilla

- marca: ”Philips”
- encendida: true
- potencia: 3

b2:Bombilla

- marca: ”Siemens”
- encendida: false
- potencia: 7

f1:Futbolista

- id: 2345
- nombre: “Joaquín”
- apellido: “Sánchez”
- edad: 40
- dorsal: 17
- posición: “centrocampista”
- equipo: “Real Betis Balompié”

Declaración … ¿Cómo vamos a implementarlas?

Modificadores de acceso/comportamiento

Palabra reservada que denota la declaración de una clase.

Nombre de la clase

Herencia (UD4)

Interfaces (al final de esta UD)

```
[modificador] class NombreClase [extends ClasePadre] [implements <Lista Interfaces>]{

// Declaración de atributos
        [modificador] tipo1 atributo1;
        …
        [modificador] tipoN atributoN;


// Declaración y definición de métodos
        [modificador] tipo1 método1(<lista de parámetros>){
                //Código del método1
        }
        …
        [modificador] tipoM métodoM(<lista de parámetros>){
                //Código del métodoM
        }
}
```

Declaración de atributos

Declaración de métodos

Declaración

- Cada clase que queramos definir debe estar en un fichero de texto con el mismo nombre que la clase que vamos a crear y su extensión será .java
- **Modificadores:**
  - Acceso:
    - public: accesible desde cualquier otra clase, es decir, podemos hacer uso de ella en otra clase. Es el más común.
    - Si no hay modificador: accesible sólo desde clases de su mismo paquete.
  - Comportamiento:
    - abstract: clase abstracta → hemos omitido algún método y no hemos implementado su código.
    - final: clase que no puede ser derivada, es decir, que no puede tener clases “hijas”.

Ejemplo:

```
public class Bombilla {

    private Boolean encendida;
    private Integer potencia;
    private String marca;

    public Bombilla(String m) {
        marca = m;
    }

    public void encender() {
        encendida = true;
        potencia = 10;
    }

    public void apagar() {
        encendida = false;
        potencia = 0;
    }

    public void aumentarPotencia() {
        potencia++;
    }

    public void disminuirPotencia() {
        potencia--;
    }

    public void imprimir() {
        System.out.println("Bombilla " + marca
    + ": Enc=" +
                encendida + " Pot=" + potencia);
    }
} // End class
```

Bombilla
- marca: String - encendida: Boolean - potencia: Integer
+ Bombilla (m: String) + encender() + apagar() + aumentarPotencia() +disminuirPotencia() + imprimir()

Objetos

- Podemos interpretar que una clase es el plano que describe cómo es un objeto de dicha clase.
- Por tanto, a partir de la clase podemos fabricar objetos.
- A ese objeto construido se le denomina instancia, y al proceso de construir un objeto se le llama instanciación.
- Así pues, un OBJETO es la representación de una entidad física o conceptual modelada mediante una clase con:
    - Una estructura de datos concreta (valores de los atributos).
    - Un comportamiento (métodos que actúan sobre los valores que en un instante posee el objeto).
    - Una identidad propia (sus atributos tienen unos valores que son independiente del resto de los otros objetos).

Objetos

- La sintaxis de declaración de un objeto es:

<Interfaz o Clase> referencia = new ConstructorDeClase();

- <Interfaz o Clase> es el tipo de la referencia que define el uso (métodos disponibles) del objeto.
- referencia es el identificador con el que podremos usar (referenciar) al objeto.
- ConstructorDeClase() es el método de la clase que crea el objeto concreto, dando valores a los atributos y controlando que estos valores sean correctos.
- new es el operador que reserva memoria para guardar el objeto y lo graba físicamente en dicha memoria.
- = (asignación) hace que la referencia apunte al objeto creado para poder referenciarlo.

Ejemplo

Pokémon
- nombre: String - tipo: String - subtipo: String - salvaje: Boolean
+ Pokemon (n: String, t: String, st: String, s: Boolean) + atacar() + defender()

Bombilla
- marca: String - encendida: Boolean - potencia: Integer
+ Bombilla (m: String) + encender() + apagar() + aumentarPotencia() +disminuirPotencia() + imprimir()

p1:Pokémon
- nombre: “Pikachu” - tipo: “Eléctrico” - subtipo: null - salvaje: false

b1:Bombilla
- marca: ”Philips” - encendida: true - potencia: 3

b2:Bombilla
- marca: ”Siemens” - encendida: false - potencia: 7

```
Bombilla b1 = new Bombilla(“Philips”);
Bombilla b2 = new Bombilla(“Siemens”);
Pokemon p1 = new Pokemon (“Pikachu”, ”eléctrico”, null, false);
```

Atributos

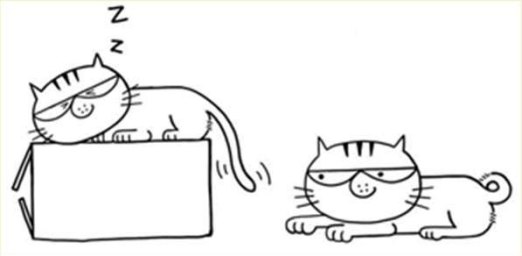
- Los atributos almacenan la información de los objetos. Los valores que toman los atributos en cada instante del programa determinan el estado del objeto.
- Los atributos se declaran como las variables o referencias en un programa.
- Tipos:
- Atributos de clase (o estáticos):
    - Común a todas las instancias de una clase.
    - Sólo se inicializan una vez.
- Atributos de instancia:
    - Determinan el estado de los objetos.
    - Cada objeto (instancia independiente) reserva memoria para almacenar los valores de sus atributos.

Atributos

- Sintaxis: [modificadores] Tipo identificador [=inicialización];
- Tipo: Puede ser un tipo primitivo, interfaz o clase.
- identificador: nombre del atributo.
- Inicialización: Valor inicial (sólo para finales y estáticos).
- Modificadores de Acceso:
  - public: acceso desde cualquier lugar.
  - private: acceso sólo para los métodos de su misma clase.
  - En blanco: acceso sólo para los métodos de su misma clase y clases de su mismo paquete.
  - protected: acceso sólo para los métodos de su misma clase, sus derivadas (subclases) y clases de su mismo paquete.
- Modificadores de Comportamiento:
  - static: atributo de clase (compartido por todos los objetos).
  - final: atributo constante.

Atributos

Modificador/ Acceso	La misma clase	Otras clases de su paquete	Subclase de otro paquete	Otras clases en otros paquetes
Public	X	X	X	X
Protected	X	X	X	
En blanco	X	X		
Private	X			



Ejemplos:

```java
public class Producto {
    public static final float IVA = 0.21F;
    private String marca;
    public String descripcion;
    protected float precio;
//Aquí irán los métodos

}
```

```java
public class Bombilla {
    private boolean encendida;
    private int potencia;
    private String marca;
    private static final int MAX_POT = 10;
//Aquí irán los métodos

}
```

Atributos: ¿Cómo los uso?

- Hay dos formas de acceder o usar los atributos de una clase:
- 1ª Forma: Desde fuera de la clase que los contiene (por ejemplo una aplicación):
  - Es necesario acceder a ellos a través de una referencia (tipo clase o interfaz) utilizando el operador "." (punto). Si es estático, puede usarse el nombre de la clase.

```
public class Main {

   public static void main( String [ ] args) {
      Bombilla b = new Bombilla("Fujitsu");
      b.encendida = true; //No aconsejable
      int pot = b.potencia; //ERROR --> Privado
      int max1 = Bombilla.MAX_POT; // Atributo de clase (static)
      int max2 = b.MAX_POT; // Otra forma de hacer lo mismo
      Bombilla.MAX_POT = 20; //ERROR --> final
   }
}
```

Atributos: ¿Cómo los uso?

- 2ª Forma: Desde dentro de la misma clase que los contiene (métodos). No es necesario ninguna referencia, simplemente el nombre del atributo.

```
public class Bombilla {
    private boolean encendida;
    private int potencia;
    private String marca;
    private static final int MAX_POT = 10;

    public void encender() {

        encendida = true;

        potencia = 10;
    }
}
```

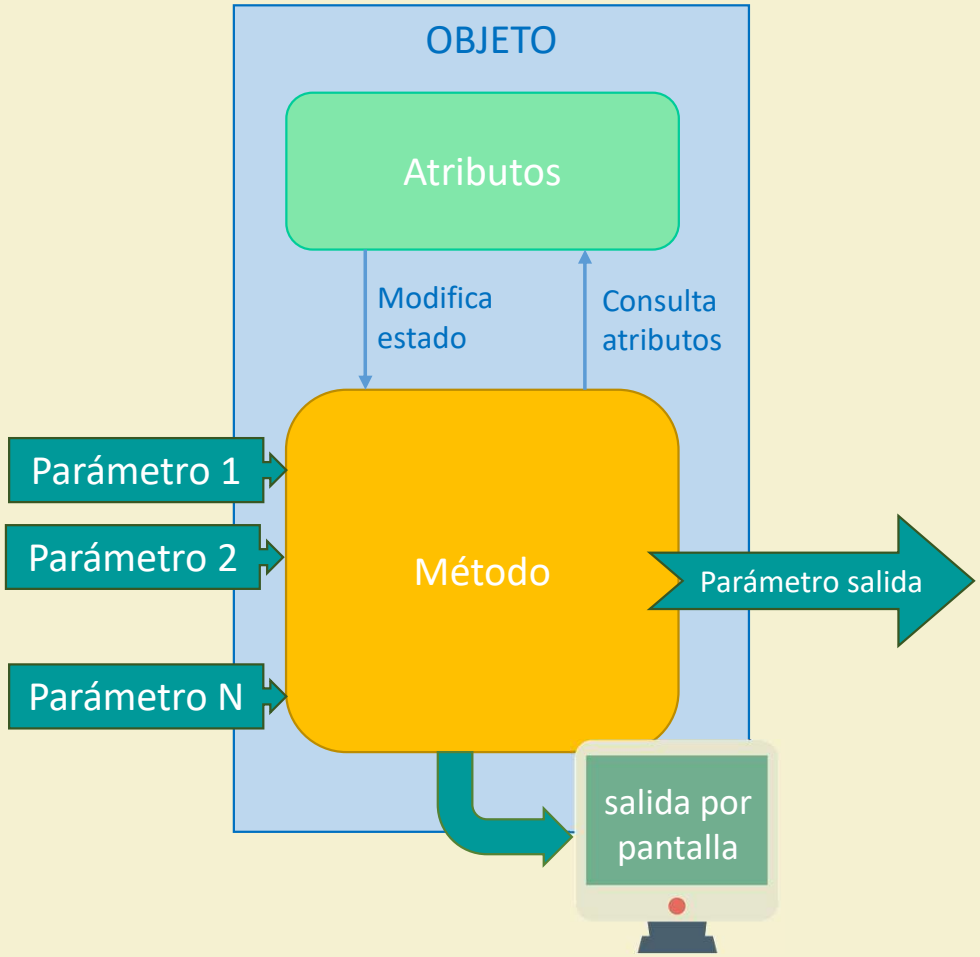
El acceso al atributo no necesita referencia

Atributos: Normas

- Los atributos deben ser siempre privados (excepto cuando se especifique lo contrario)
- Deben cambiar su valor mediante los métodos constructores y los métodos modificadores establecidos en el diseño, nunca directamente.
- Igualmente, deben ser consultados mediante los métodos observadores.
- Desde métodos de la misma clase, no tienen por qué aplicarse estas normas.

Métodos

- Son los encargados de realizar operaciones sobre los objetos de la clase, creándolos, consultándolos y cambiando su valor.
- Son los que contienen el código que se ejecuta para responder al comportamiento (funcionalidad).
- A partir de un serie de parámetros y de los valores que los atributos tengan en el momento de la ejecución, puede generar un resultado (parámetro de salida), un posible cambio de estado del objeto y/o una salida por pantalla.



Métodos

- Sintaxis: [modificadores] Tipo identificador(<lista de parámetros>){

  <Código del método>

  }

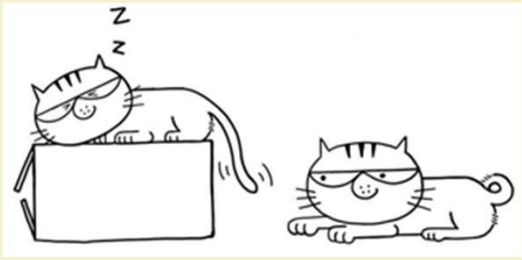
- [modificadores]: tienen el mismo sentido que con los atributos.
- Tipo: tipo del valor o referencia que devuelve el método. Puede ser un tipo primitivo, interfaz o clase.
- Identificador: nombre del método.
- (<lista de parámetros>): parámetros formales.
- {<Código>}: Código Java que se ejecuta cuando se invoca al método.

Métodos

Modificador/ Acceso	La misma clase	Otras clases de su paquete	Subclase de otro paquete	Otras clases en otros paquetes
Public	X	X	X	X
Protected	X	X	X	
En blanco	X	X		
Private	X			

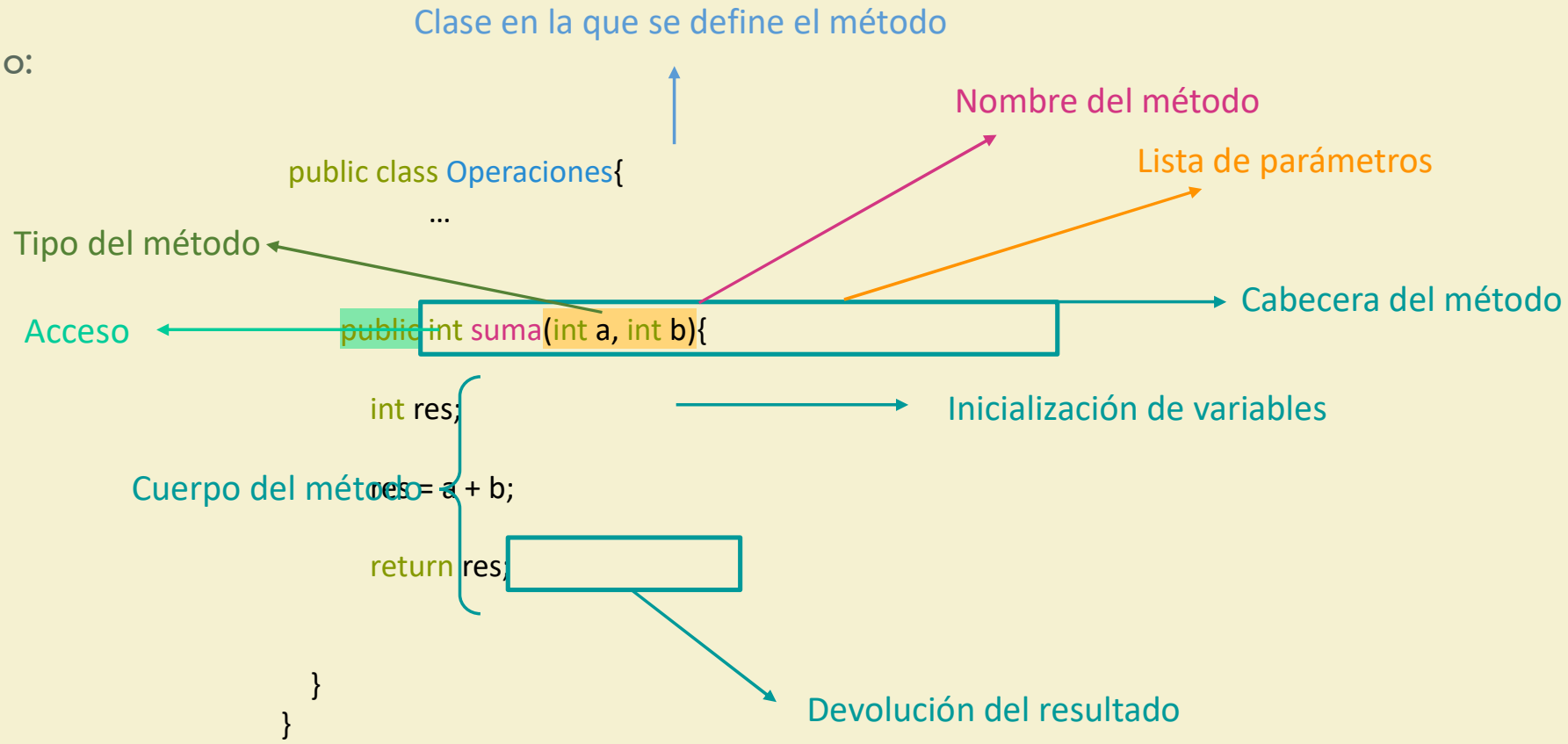
Modificadores:

- **Acceso:**
  - public: acceso desde cualquier lugar.
  - private: acceso sólo para los métodos de su misma clase.
  - En blanco: acceso sólo para los métodos de su misma clase y clases de su mismo paquete.
  - protected: acceso sólo para los métodos de su misma clase, sus derivadas (subclases) y clases de su mismo paquete.
- **Comportamiento:**
  - abstract: método abstracto (algún método no implementado).
  - final: método no redefinible en una subclase.
  - static: método de clase.



Métodos

- Ejemplo:



Métodos: Trabajando con Parámetros

- Los parámetros son argumentos que le pasamos al método para que éste haga su cometido.
- Definición (por ejemplo en la clase Operaciones):

```
public int suma(int a, int b){
    int res;
    res = a + b;
    return res;
}
```

- Llamada (por ejemplo desde la clase Main):

```
int x=3, y=6, s;
Operaciones o = new Operaciones();
s = o.suma(x, y);
System.out.println("La suma de 10 y 30 es" , o.suma(10,30));
```

Métodos: Trabajando con Parámetros

- Podemos usar los parámetros de dos formas: pasándolos por valor o por referencia.

POR VALOR:

- Si el parámetro es de tipo primitivo, se pasa una copia del valor.
- Al tratarse de una copia, los cambios que el método haga sobre los parámetros formales no serán permanentes fuera del propio método, conservando los parámetros reales su valor inicial.

POR REFERENCIA:

- Si el parámetro es un Objeto, se pasa el propio objeto, no una copia.
- Al no tratarse de una copia, los cambios que el método haga sobre los parámetros formales (objeto) serán permanentes fuera del propio método, cambiando los parámetros reales su valor inicial.

Métodos: Parámetros – Ejemplo: Paso por valor

```
int x=3, y=6, s;
Operaciones o = new Operaciones();
s = o.suma(x, y);
```

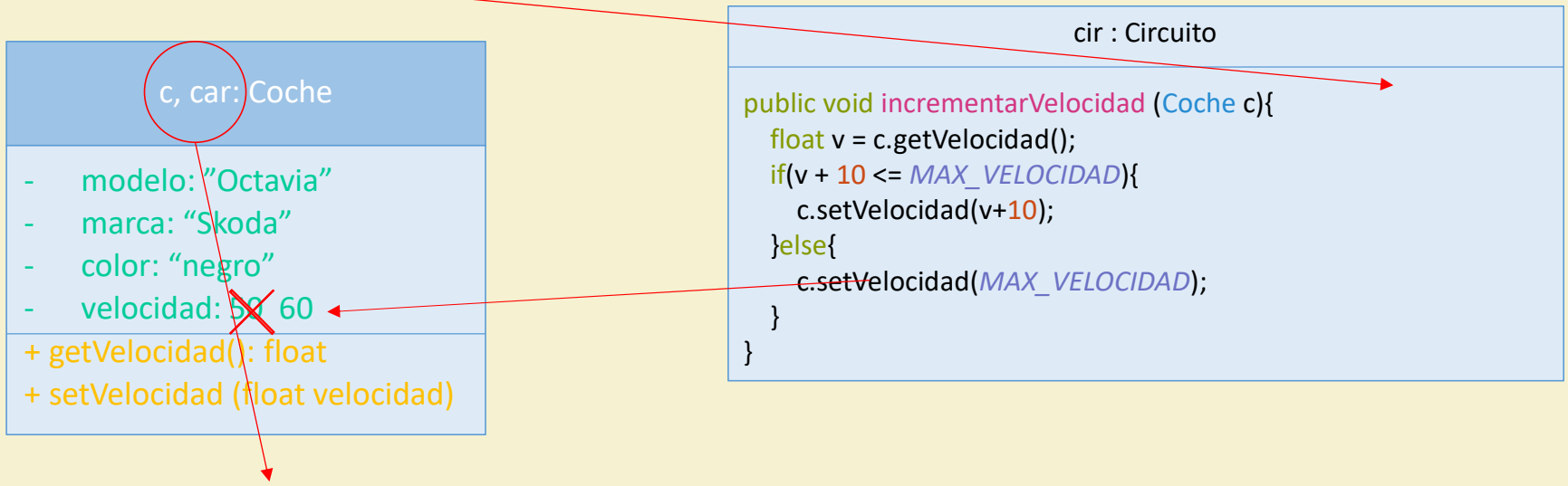
o: Operaciones

```
public int suma(int a, int b){

        int res;
        res = a + b;
        return res;

}
```

Tras la ejecución, x sigue valiendo 3 e y es 6, mientras que s toma el valor de lo que devuelve el método, es decir 9.

Métodos: Parámetros – Ejemplo: Paso por referencia

```
Coche car = new Coche("Octavia","Skoda", "Negro",50);
Circuito cir = new Circuito("Mónaco");
cir.incrementarVelocidad(car);
```



Las referencias car y c apuntan al mismo objeto, por lo que al cambiar la velocidad de c, cambia la de car.

Métodos: Devolviendo el resultado (return)

- Si el método no devuelve ningún valor como resultado, el método se declarará de tipo void.

```
public void incrementarVelocidad (Coche c){
    float v = c.getVelocidad();
    if(v + 10 <= MAX_VELOCIDAD){
        c.setVelocidad(v+10);
    }else{
        c.setVelocidad(MAX_VELOCIDAD);
    }
}
```

- Si el método devuelve algún valor u objeto como resultado, el método se declarará del tipo correspondiente al valor o referencia que devuelva y la última instrucción en ejecutarse ha de ser return seguida de la expresión que contenga el valor a devolver.

```
public int suma(int a, int b){
        int res;
        res = a + b;
        return res;
}
```

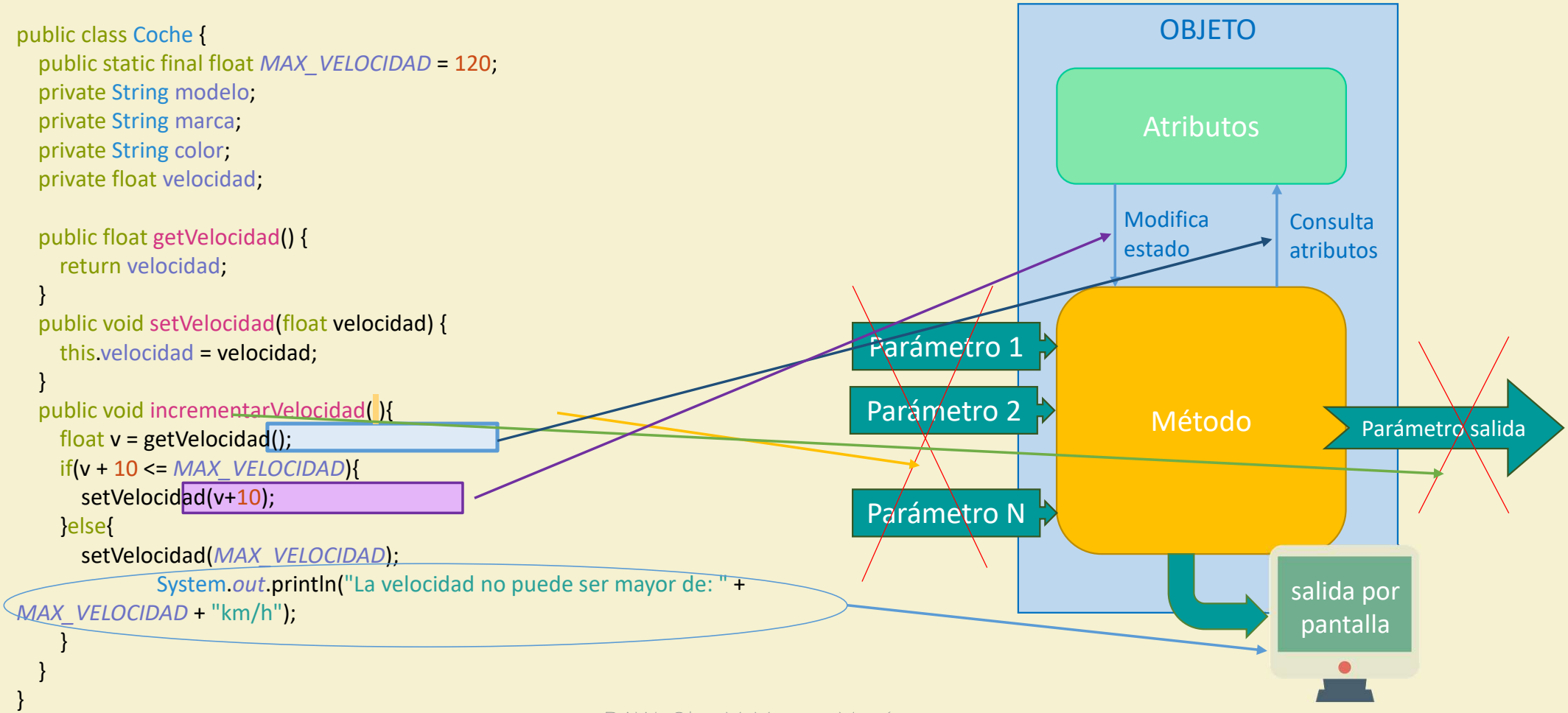
Métodos: Uso de atributos

- Los métodos pueden usar los atributos de su clase libremente como si se tratara de variables locales al propio método.
- Para ello, simplemente deben usar el identificador del atributo, sea para consultar su valor o para modificarlo.
- En caso de cambiar el valor de un atributo, este cambio será permanente incluso cuando el método haya acabado de ejecutarse.

```
public class Coche {
   public static final float MAX_VELOCIDAD = 120;
   private String modelo;
   private String marca;
   private String color;
   private float velocidad;

   public float getVelocidad() {
      return velocidad;
   }
   public void setVelocidad(float velocidad) {
      this.velocidad = velocidad;
   }
   public void incrementarVelocidad(){
      float v = getVelocidad();
      if(v + 10 <= MAX_VELOCIDAD){
         setVelocidad(v+10);
      }else{
         setVelocidad(MAX_VELOCIDAD);
                  System.out.println("La velocidad no puede ser mayor de: " +
MAX_VELOCIDAD + "km/h");
      }
   }
}
```

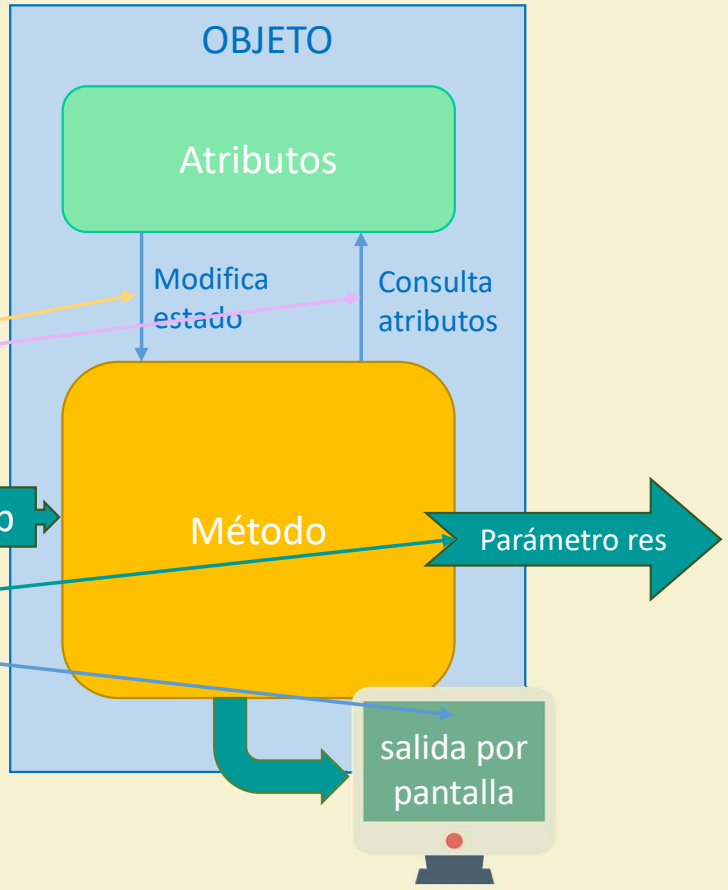
Métodos: Uso de atributos – Ejemplo:

```
public class Coche {
   public static final float MAX_VELOCIDAD = 120;
   private String modelo;
   private String marca;
   private String color;
   private float velocidad;

   public float getVelocidad() {
      return velocidad;
   }
   public void setVelocidad(float velocidad) {
      this.velocidad = velocidad;
   }
   public void incrementarVelocidad( ){
      float v = getVelocidad();
      if(v + 10 <= MAX_VELOCIDAD){
         setVelocidad(v+10);
      }else{
         setVelocidad(MAX_VELOCIDAD);
                System.out.println("La velocidad no puede ser mayor de: " +
MAX_VELOCIDAD + "km/h");
      }
   }
}
```



Métodos: Uso de atributos – Ejemplo:

```
public class Producto {
   public static final double IVA = 0.21;
   private String marca;
   public String descripcion;
   protected double precio;

   public boolean setPrecio(double p){
      boolean res = false;
      if (p <= IVA){
         precio = precio *(1+IVA);
         res = true;
      }
      else
         System.out.println("No se puede aumentar más el precio");
      return res;
   }

}
```



Métodos: ¿Cómo los llamamos o usamos?

- El acceso a los métodos es similar al de los atributos:

1. Desde fuera de la clase que los contiene (por ejemplo una aplicación):

- Es necesario acceder a ellos a través de una referencia (objeto) de la clase o interfaz (dependiendo de si el método está en la interfaz o no) utilizando el operador "."(punto). Si es estático, puede usarse el nombre de la clase.

2. Desde dentro de la misma clase que los contiene (otros métodos). En este caso no es necesario ninguna referencia, simplemente el nombre del método.

Métodos: ¿Cómo los llamamos o usamos?

- Ejemplos:

```
public static void main(String[] args) {
        int x=3, y=6, s;
        Operaciones o = new Operaciones();
        s = o.suma(x, y);
        System.out.println("La suma de 10 y 30 es" , o.suma(10,30));
}
```

CON REFERENCIA

```
public class Coche {
   public static final float MAX_VELOCIDAD = 120;
   private String modelo;
   private String marca;
   private String color;
   private float velocidad;

   public float getVelocidad() {
      return velocidad;
   }
   public void setVelocidad(float velocidad) {
      this.velocidad = velocidad;
   }
   public void incrementarVelocidad(){
      float v = getVelocidad();
      if(v + 10 <= MAX_VELOCIDAD){
         setVelocidad(v+10);
      }else{
         setVelocidad(MAX_VELOCIDAD);
         System.out.println("La velocidad no puede ser mayor de: " +
MAX_VELOCIDAD + "km/h");
      }
   }
}
```

SIN REFERENCIA

Métodos de Clase

- Son los declarados con el modificador **static**.
- No necesitan un objeto de la clase para ser invocados. Pueden ser llamados tanto usando el nombre de la clases como cualquiera de sus instancias (referencias a objetos).
- No pueden acceder a los miembros (atributos o métodos) no estáticos de la clase.
- Sus funciones principales son:
  - Modificar la información que es común para los objetos de la clase, es decir, los atributos static de la clase (Ojo, no los finales).
  - El tratamiento de otros objetos. Por ejemplo, el método main que inicia una aplicación.
  - Definir una clase como "librería de funciones". Algunas clases se diseñan sólo con métodos estáticos. Por ejemplo, Math, System o Arrays (Los veremos en UD siguientes) . Tienen un constructor privado vacío.

Métodos de Clase

```
public class ClaseStatic {
    private int atributoObjeto;
    private static int atributoClase;
    public void metodoObjeto (){
        atributoObjeto++; //OK
        atributoClase++;  //OK
    }
    public static void metodoClase(){
        atributoObjeto++;  // DA ERROR, NO ES ESTÁTICO
        atributoClase++;  //OK
        metodoObjeto();  //DA ERROR, NO ES ESTÁTICO
    }

}
```

```
public class MainClaseStatic {

    public static void main(String [] args){
        ClaseStatic e1 = new ClaseStatic();
        e1.metodoObjeto();
        e1.metodoClase();
        ClaseStatic.metodoClase();
        ClaseStatic.metodoObjeto(); // DA ERROR, NO ESTÁTICO
    }
}
```

Métodos de Clase

- Hay 3 tipos de métodos de clase:
  - Constructores (Constructors): Construyen objetos e inicializan su estado, es decir, asignan valores iniciales a los atributos.
  - Observadores (Getters): Consultan el estado del objeto sin cambiarlo, es decir, realizan acciones que no cambian los valores de sus atributos.
  - Modificadores (Setters): Modifican el estado del objeto, es decir, cambian los valores de sus atributos.

Métodos de Clase: Constructores

- Un constructor es un método perteneciente a la clase que se encarga de crear objetos dando valores a los atributos de éste: reserva memoria, aloja el objeto ella e inicializa sus valores.
- Cuando creamos un objeto de una clase determinada, éste se crea ejecutando el código del método constructor que hayamos utilizado, de forma que además de asignar valores a los atributos, podemos controlar que esos valores son correctos.
- Si el constructor no asigna de manera explícita valores a los atributos, éstos toman los siguientes valores por defecto según el tipo:

Tipo del atributo	Inicializado por defecto a:
boolean	false
char	“\u0000”
entero (byte, short, int, long)	0
coma flotante	+ ó +0.0d
Referencia de objeto	null

Métodos de Clase: Constructores

- **Declaración de un método constructor:**
  - Los constructores siempre tienen el nombre de la clase.
  - No tiene tipo de vuelta, es decir, no devuelven nada.
  - Siempre son públicos.
  - Sólo pueden usarse para crear objetos. No se llaman como los métodos normales.
  - No se escriben en la interfaz.
- Constructor por defecto: **Si la clase no contiene ningún constructor explicito (escrito en la clase), el compilador proporciona uno por defecto sin código, es decir, que asignará los valores iniciales de la tabla anterior. Si se declara cualquier otro constructor, el constructor por defecto se anula.**
- Constructor sin parámetros: **Sustituye al constructor por defecto, dando siempre los mismos valores iniciales a los atributos, que serán los valores iniciales que se especifiquen.**
- Constructores con parámetros: **Constructor al que se le pasa parámetros de inicialización de los atributos.**

Métodos de Clase: Constructores – Ejemplo:

```
public class Bombilla {
   public static final int MAX_POT = 10;
   private boolean encendida;
   private int potencia;
   private String marca;

   public Bombilla(){
      marca = "desconocida";
   }
   public Bombilla(String m) {
      marca = m;
   }
   public Bombilla(String m, int p) {
      marca = m;
      if(p <= MAX_POT) {
         potencia = p;
      } else {
         potencia = MAX_POT;
      }
      encendida = true;
   }
//Más métodos
```

```
// Llamadas a la clase bombilla:
Bombilla b1 = new Bombilla();
Bombilla b2 = new Bombilla("Balay");
Bombilla b3 = new Bombilla("Bosh", 7);
```

Métodos de Clase: Constructores

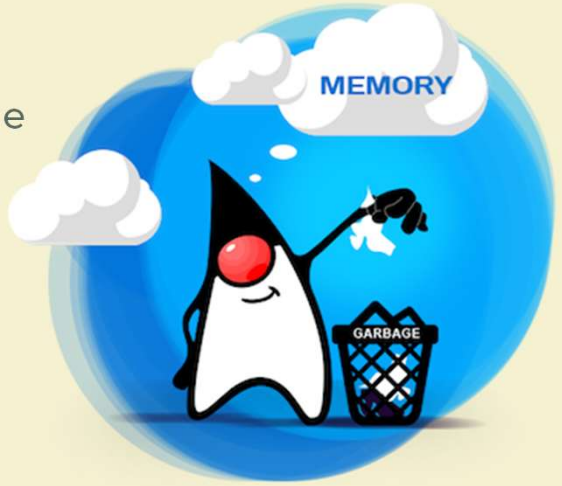
- Como vimos anteriormente, la sintaxis de declaración de un objeto es:

  <Interfaz o Clase> referencia = new ConstructorDeClase();

- Dependiendo de los parámetros reales que se utilicen, se llamará a un constructor o a otro:

```
// Llamadas a la clase bombilla:
Bombilla b1 = new Bombilla();
Bombilla b2 = new Bombilla("Balay");
Bombilla b3 = new Bombilla("Bosh", 7);
```

- Gestión automática de memoria: La gestión de memoria en JAVA se realiza de manera automática por el Recolector de Basura (Garbage Collector). Cuando un objeto deja de ser utilizado en nuestro programa (no tiene ninguna referencia que apunta a él), el recolector de basura se encarga de liberar la memoria que ocupaba sin que el programador tenga que intervenir.



Métodos de Clase: Observadores

- Los métodos observadores (también conocidos como getters por su forma de nombrarse) son aquellos que consultan el estado de un objeto (valor de sus atributos) pero no lo modifican, es decir no cambian los valores de los atributos.
- Observadores Estándar (Métodos get o getters):
  - Además de los métodos requeridos por el diseño de la clase (Interfaz), normalmente se suele declarar e incluir en la interfaz un método observador por cada atributo cuya única función es devolver el valor de éste para poder consultarlo, ya que éste será normalmente privado.
  - El método observador de un determinado atributo devuelve un valor del el mismo tipo que dicho atributo. Si el atributo es de cualquier tipo distinto a boolean, el método observador se nombra con el prefijo get seguido del nombre del atributo. Si el atributo es booleano, se nombra el prefijo is seguido del nombre del atributo.

Métodos de Clase: Observadores

- Caso genérico: Modificador de un atributo no booleano “atrib1” y otro booleano “atrib2”:

```
public class MiClase{

    Tipo atrib1;   // Tipo distinto a boolean
    boolean atrib2;


  public Tipo getAtrib1(){  // Es del mismo tipo del atributo y se nombra
                // con el prefijo get y el nombre del atributo
    return atrib1;   // Devuelve el valor del atributo
  }

  public boolean isAtrib2(){  // Es de tipo booleano y se nombra con el
                // prefijo is y el nombre del atributo
    return atrib2;   // Devuelve el valor del atributo
  }


}
```

Métodos de Clase: Observadores - Ejemplo

```
public class Coche {
   public static final float MAX_VELOCIDAD = 120;
   private String modelo;
   private String marca;
   private String color;
   private float velocidad;

   public String getModelo() {
      return modelo;
   }
   public String getMarca() {
      return marca;
   }
   public String getColor() {
      return color;
   }
   public float getVelocidad() {
      return velocidad;
   }
// Más métodos...
```

Métodos de Clase: Modificadores

- Son aquellos que modifican el estado de los objetos cambiando los valores de los atributos. Es la herramienta para controlar el cambio de los objetos, ya que pueden testar el cambio de valores antes de realizarlo.
- Modificadores Estándar (Métodos set o setters):
  - Además de los métodos requeridos por el diseño de la clase (Interfaz), normalmente se suele declarar un método modificador por cada atributo cuya única función es modificar el valor de éste, ya que el atributo será normalmente privado, y opcionalmente comprobar su validez.
  - El método modificador de un determinado atributo devuelve normalmente void, toma como parámetro un valor del el mismo tipo que dicho atributo y se nombra con el prefijo set seguido del nombre del atributo. En el cuerpo del método, se asigna el valor del parámetro al atributo, aunque en ocasiones se añade más código para comprobar la validez de la asignación.

Métodos de Clase: Modificadores

- Caso genérico: Modificador del atributo “atrib1”:

```
public class MiClase{

        Tipo atrib1;   // Tipo distinto a boolean
        boolean atrib2;

                        ...

        public void setAtrib1(Tipo v){  // Normalmente devuelve void, se nombra
                                                    // con el prefijo set y el nombre del atributo,
          atrib1 = v;        // toma como parámetro un valor del mismo tipo
                                                    // que el atributo y lo asigna al atributo.
        }
      //Más métodos
```

Métodos de Clase: Modificadores

```
public class Coche {
   public static final float MAX_VELOCIDAD = 120;
   private String modelo;
   private String marca;
   private String color;
   private float velocidad;

   public String getModelo() {
      return modelo;
   }
   public String getMarca() {
      return marca;
   }
   public String getColor() {
      return color;
   }
   public float getVelocidad() {
      return velocidad;
   }
// Más métodos...
```

```
   public void setModelo(String modelo) {
      this.modelo = modelo;
   }

   public void setMarca(String marca) {
      this.marca = marca;
   }

   public void setColor(String color) {
      this.color = color;
   }

   public void setVelocidad(float velocidad) {
      this.velocidad = velocidad;
   }
```

Métodos de Clase: Sobrecarga

- Java permite declarar en el mismo ámbito (clase y/o interfaz) métodos con el mismo nombre, siembre que los parámetros sean de distinto tipo o el número de parámetros sea distinto.
- No se admiten sobrecargas donde sólo cambie el tipo de vuelta de la función miembro.

```
public class Bombilla {
...
public void aumentarPotencia(int i){
   if (potencia + i <= MAX_POT)
      potencia = potencia + i;
   else
      System.out.println("No se puede aumentar más la potencia");
}

public void aumentarPotencia() {
   potencia++;
}
...
```

Métodos de Clase: el uso de this

- En todos los métodos existe por defecto una referencia al objeto concreto que ejecuta ese método.
- Esa referencia es la palabra reservada this.
- Entre otras utilidades, la referencia this sirve para:

1. Resolver ambigüedades cuando existen atributos con el mismo identificador que alguna variable local o parámetro.
2. Cuando necesitamos hacer uso del objeto completo.
3. Llamada desde un constructor a otro constructor de la misma clase.

```
public void setMarca(String marca) {
   this.marca = marca;
}
```

```
public boolean igual(Bombilla l) {
   if (this.equals(l))
      return true;
   else
      return false;
}
```

```
public Bombilla(String m) {
   marca = m;
}
public Bombilla(String m, int p) {
   this(m);
   if(p <= MAX_POT) {
      potencia = p;
   } else {
      potencia = MAX_POT;
   }
   encendida = true;
}
```

Métodos de Clase: el uso de this

- En todos los métodos existe por defecto una referencia al objeto concreto que ejecuta ese método.
- Esa referencia es la palabra reservada this.
- Su utilidad:
  - Resolver ambigüedades cuando existen atributos con el mismo identificador que alguna variable local o parámetro.
  - Hacer uso del objeto completo.

```
public void setMarca(String marca) {
   this.marca = marca;
}
```

```
public boolean igual(Bombilla l) {
   if (this.equals(l))
      return true;
   else
      return false;
}
```

Encapsulación

- Con el objetivo de evitar situaciones de incoherencia o estados incorrectos, los atributos de un objeto sólo deben ser modificados desde dentro del propio objeto.
- En este sentido, un objeto debe estar encapsulado respecto al resto del sistema, es decir, el resto del programa debe interactuar con el objeto únicamente a través del conjunto de métodos que definen los servicios proporcionados por el objeto, nunca cambiado los valores de sus atributos de manera directa.
- Estos métodos definen la interfaz entre dicho objeto y el programa que los usa.
- ¿Cómo se consigue?
  - Todos los atributos, excepto las constantes estáticas, deben declararse con visibilidad privada.
  - En caso de que sea necesario acceder a dichos atributos, la modificación o consulta de los mismos se hará mediante métodos públicos definidos en la interfaz correspondiente → Métodos de Servicio.
  - Aquellos métodos auxiliares que sirvan para ayudar a los métodos de servicio a hacer su cometido se declararán con visibilidad privados → Métodos de Soporte.

Encapsulación – Ejemplo:

```
public class Entrenador {
   private String nombre;
   private int pokemons; //número de pokémon que tiene el entrenador
   public String getNombre() {
      return nombre;
   }
   public void setNombre(String nombre) {
      this.nombre = nombre;
   }
   public int getPokemons() {
      return pokemons;
   }
   public void setPokemons(int pokemons) {
      if(pokemons >=0)
         this.pokemons = pokemons;
   }
   public Entrenador(String nombre, int pokemons){
      this.nombre = nombre;
      setPokemons(pokemons);
   }
}
```

</> Trabajando con Interfaces

Interfaces

- Ya hemos usado el término interfaz para referirnos al conjunto de métodos públicos mediante los cuales podemos interactuar con un objeto. Ahora vamos a aclarar el concepto.
- Las interfaces son elementos de programación que se utilizan para definir el comportamiento (funcionalidad) de los objetos.
- La interfaz es la “cara pública” de un objeto.
- Por ejemplo, en un coche (que sería un objeto), la interfaz sería el volante, los pedales y la palanca de cambio que me permiten una serie de acciones (funcionalidad) como dirigir la dirección del coche, acelerar, frenar y cambiar de marcha. Si sabemos usar la interfaz, da igual el coche concreto que sea, todos se conducen igual.

Interfaces

Interfaces Java:

- Una interfaz Java es un conjunto de constantes y métodos abstractos.
- Un método abstracto es un método que no tiene implementación, es decir, no existe un cuerpo de código que definido para el método.
- Por tanto, en las interfaces sólo se describe la signatura (prototipos) de los métodos. La implementación se realizará en las clases.
- Cuando se define una interfaz, se está definiendo un nuevo tipo de datos, ya que se puede declarar una variable de tipo interfaz.

Declaración … ¿Cómo vamos a implementarlas?

Modificador de acceso

Palabra reservada que denota la declaración de una interface

Nombre de la interfaz

Herencia (UD4)

Otras Interfaces

```
[modificador] interface NombreInterface [extends <Lista Interfaces>]{

// Declaración de atributos
        static final tipoC1 constante1 = valor1;
        static final tipoC2 constante2 = valor2;
        ...
        static final tipoCM constanteM = valorM;

// Declaración y definición de métodos
        [modificador] tipo1 método1(<lista de parámetros>);
        [modificador] tipo2 método2(<lista de parámetros>);
        ...
        [modificador] tipoM métodoM(<lista de parámetros>);

}
```

Declaración de constantes

Declaración de métodos

Interfaces : Definición

- Al igual que las clases, cada interfaz se define en un fichero de texto con el mismo nombre que la clase y extensión *.java*.
- Las interfaces sólo admite el modificador de acceso public. Si la interfaz no es public, tendrá visibilidad de paquete (visibilidad por defecto)
- Los métodos declarados en una interfaz son siempre public y abstract de manera implícita.
- Las interfaces sólo describen la signatura de los métodos, es decir, el tipo de vuelta y la lista de parámetros.
- Ejemplo (Series.java):

```
public interface Series {
   int getSiguiente(); //Devuelve el siguiente número de la serie
   void reiniciar(); //Reinicia
}
```

Interfaces: Implementación

- Una clase implementa una interfaz proporcionando implementación para cada uno de los métodos abstractos definidos en la interfaz.
- Para que una clase implemente una interfaz, es necesario usar en la cabecera de la clase la palabra reservada implements seguida de la interfaz o interfaces que la clase implemente.
- Ejemplos:

```
public class Lampara implements Interruptor, Bombilla { . . . }
public class Combate implements Entrenador, Pokemon { . . . }
```

La clase Lampara implementa las interfaces Interruptor y Bombilla

La clase Combate implementa las interfaces Entrenador y Pokemon

Interfaces: Implementación

A tener en cuenta:

- Las clases pueden implementar interfaces, aunque puede haber clases que no implementan a ninguna interfaz.
- Una interfaz puede ser implementada por varias clases. Cuando una clase implementa una interfaz, se ve obligada a implementar todos los métodos definidos en la misma.
- La clase puede contener más métodos de los que tenga las interfaces que implementa.
- Cuando una clase implementa una interfaz, implementa a todas sus interfaces padres.
- Cuando una interfaz define una constante, ésta puede usarse en las clases que implementen dicha interfaz sin necesidad de anteponer el nombre de la interfaz al de la constante.

Interfaces – Ejemplo:

```
public interface Series {

   int getSiguiente();
//Devuelve el siguiente número de la
serie
   void reiniciar(); //Reinicia la serie
}
```

```
public class Fibonacci implements Series {

   int iniciar;
   int valor;
   int anterior;

   public Fibonacci() { iniciar = 0;  valor = 1; anterior = 0;}

   public int getSiguiente() {

      int aux = valor;
      valor += anterior;
      anterior = aux;
      iniciar ++;
      return valor;
   }

   public void reiniciar() { valor = 1; anterior = 0; iniciar = 0; }

   //Añadiendo un método que no está definido en Series
   int getAnterior() { return anterior;}
}
```

Interfaces: Utilización

- Al igual que ocurre con las clases, cuando definimos un interfaz estamos definiendo un nuevo tipo de datos que podemos usar para declarar variables (referencias) de ese tipo:

```
Interruptor i1 = new Lampara("Philips");
Interruptor i2 = new Lampara("Bosh");
Interruptor i3 = i2; // misma lámpara que la 2, pero con otro nombre
```

- Una vez creado un objeto, podemos actuar sobre él a través de la referencia que lo apunta utilizando SÓLO los métodos definidos en la interfaz. Para utiliza una referencia de tipo interfaz, simplemente hay que hacer uso del operador “punto” para acceder al método que se quiera ejecutar.

```
i1.encender();
i1.apagar();
i2.disminuirPotencia(); //ERROR: Interruptor no tiene este método
i1.encendida=true;  //ERROR: desde la interfaz no se puede acceder
                    //a los atributos, aunque éstos sean públicos
```

Interfaces vs Clases en la creación de Objetos

- Como ya hemos visto, las referencias pueden ser declaradas de tipo Interfaz o Clase. Sin embargo, no es aconsejable declararlas de tipo Clase por varias razones:
  - Una interfaz puede ser implementada por cualquier número de clases, permitiendo a cada clase compartir el interfaz de programación sin tener que ser consciente de la implementación que hagan las otras clases que implementen el interfaz.
  - El uso de interfaces facilita el encapsulamiento de los objetos.
  - Las clases no permiten herencia múltiple. (UD4)
  - El uso de interfaces en los parámetros de los métodos, permite que éste pueda ser ejecutado con independencia de la clase del objeto, siempre que dicha clase implemente la interfaz.

Alias

- Un objeto puede ser referenciado por varias referencias, incluso de tipos diferentes.
- La asignación de variables de tipos primitivos implica un copia del valor, siendo en este caso variables independientes.

```
int a = 15;
int b = a;
a = 20;
```

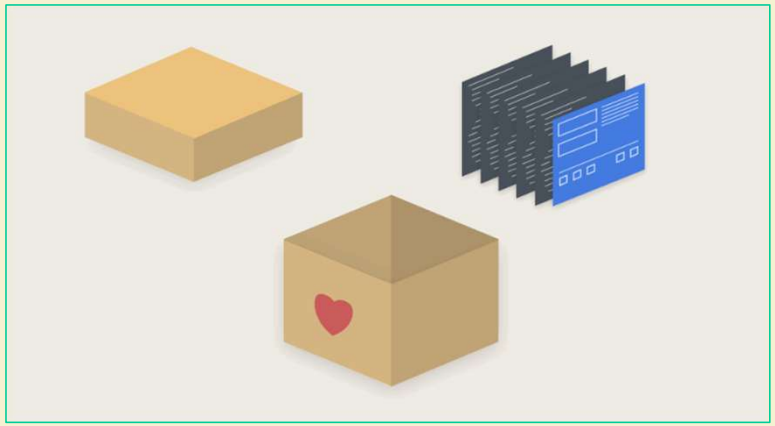
b es una copia de a, pero si cambiamos el valor de a, el de b no varía

- La asignación de referencias (variables de tipo clase o interfaz) no implica copia, sino que cuando una referencia refOrigen se asigna a otra refDestino, ésta es un alias de refOrigen, y ambas referencian al mismo objeto.

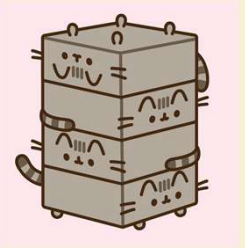
```
Coche c = new Coche("207", "Peugeot");
Vehiculo v1 = (Vehiculo)c;
v1.incrementarVelocidad();
```

v1 es un alias de c , si cambia el valor de v1 también cambia el de c, son el mismo objeto.

</> Paquetes o packages



Paquetes



- Cuando hacemos programas de cierta entidad, el número de clases e interfaces tiene a crecer bastante. Incluir todas éstas en el mismo directorio complica la organización de las mismas. Por ello se recomienda hacer grupos de clases e interfaces, de forma que todas las clases e interfaces que traten de un determinado tema o estén relacionadas entre sí vayan juntas.
- La palabra reservada package permite agrupar clases e interfaces en paquetes.
- Cada clase o interfaz puede ser incluida en un paquete escribiendo al principio del fichero .java la siguiente directiva:

```
package nombrePaquete;
```

- Los nombres de los paquetes son normalmente palabras separadas por puntos y coincide con el nombre del directorio (carpeta) en el que se almacena la clase (.java o .class). Por ejemplo, la clase IO usada en las actividades prácticas y de desarrollo para el manejo de la entrada estándar de texto se encuentra en el paquete poo.io. El programador de esta clase escribió al principio del fichero IO.java la instrucción “package edi.io;” y además guardó el fichero en un subdirectorio llamado “io” dentro del directorio “edi”.

Paquetes

- Por otra parte, los paquetes se cargan con la palabra reservada import, especificando el nombre del paquete como ruta y nombre de clase o interfaz. También se pueden cargar todo el contenido de un paquete utilizando un asterisco. Por ejemplo, si en una clase o interfaz voy a usar la clase IO y varios componentes del paquete java.util, escribiremos lo siguiente antes de la declaración de clase o interfaz.

```
import edi.io.IO;  //importa la clase IO del paquete edi.io
import java.util.*; //importa todas las clases e interfaces del paquete java.util
```

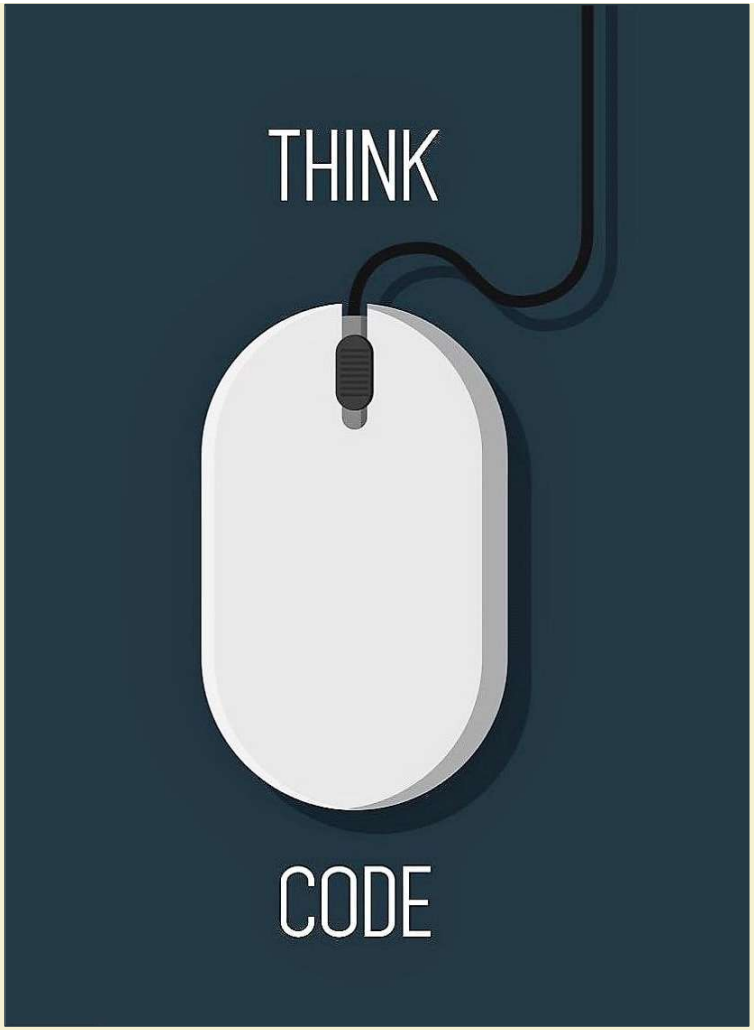
- Si un fichero fuente Java no contiene ningún package, se coloca en el paquete por defecto sin nombre, es decir en el mismo directorio que el fichero fuente.

Paquetes incluidos en Java

- El JDK proporciona una serie de paquetes que incluyen utilidades y herramientas, de las cuales destacan los siguientes:
- java.lang: Es el paquete por defecto y no es necesario importarlo, ya que se importa de manera implícita en todos los ficheros java. Incluye las clases del lenguaje Java propiamente dicho: Object, Exception, System, Integer, Float, Math, String, etc.
- java.util: Incluye una serie clases e interfaces de utilidad como: Date (fecha), Random (números aleatorios), Comparator (Comparadores), List / ArrayList / LinkedList (Listas), etc.
- java.awt: El paquete Abstract Windowing Toolkit (awt) contiene clases para generar widgets (botones, combos, etc) y componentes GUI (Interfaz Gráfico de Usuario), como Incluye las clases Button, Checkbox, Menu, Panel, TextArea, etc.
- java.io: El paquete de entrada/salida contiene las clases de acceso a ficheros: FileInputStream y FileOutputStream.

¿Preguntas?

04 Unidad Didáctica 4: Entrada-Salida y Clases Envoltura



Módulo: Programación

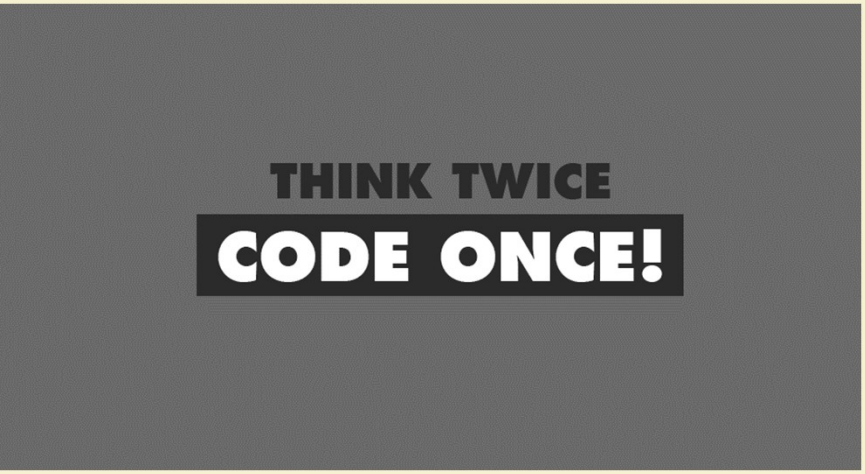
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

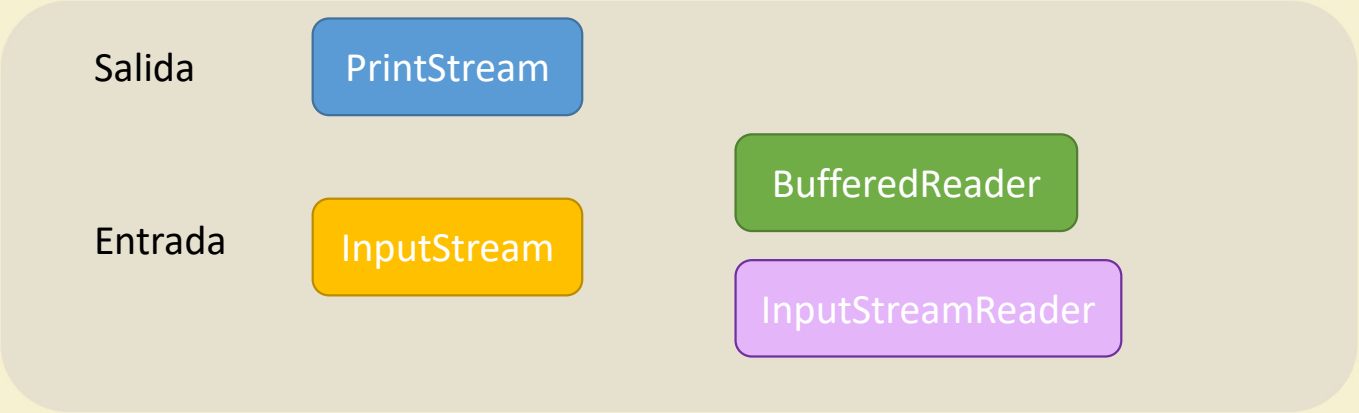
En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

Índice



Entrada y Salida

- Una de las operaciones más habituales que tiene que realizar un programa Java es intercambiar datos con el exterior. Para ello, el paquete java.io incluye una serie de clases que permiten gestionar la entrada y salida de datos en un programa, independientemente de los dispositivos utilizados para el envío/recepción de los datos. El esquema muestra cuáles son las principales clases de este paquete y la operación para la que son utilizadas:



</> Salida de Datos

Salida de datos

- El envío de datos al exterior se gestiona a través de la clase PrintStream, utilizándose un objeto de la misma para acceder al dispositivo de salida. Posteriormente, con los métodos proporcionados por esta clase, podemos enviar la información al exterior.
- El proceso de envío de datos a la salida debe realizarse siguiendo dos pasos.
- 1. Obtención del objeto PrintStream. Se debe crear un objeto PrintStream asociado al dispositivo de salida, la forma de hacerlo dependerá del dispositivo en cuestión. La clase System proporciona el atributo estático out que contiene una referencia al objeto PrintStream asociado a la salida estándar, representada por la consola.
- 2. Envío de datos al stream. La clase PrintStream dispone de los métodos print(String cadena) y println(String cadena) para enviar una cadena de caracteres al dispositivo de salida, diferenciándose uno de otro en que el segundo añade un salto de línea al final de la cadena. Esto explica que para enviar un mensaje a la consola se utilice la expresión: `System.out.println("texto de salida");`
- Ambos métodos se encuentran sobrecargados, es decir, existen varias versiones de los mismos para los distintos tipos soportados por el lenguaje. por el lenguaje. En general, para enviar datos desde un programa Java al exterior habrá que utilizar la expresión:

```
objeto printstream.println(dato);
objeto printstream.print(dato);
```

Salida con formato

- La clase PrintStream proporciona dos métodos de escritura que permiten aplicar un formato a la cadena de caracteres que se va a enviar a la salida. Se trata de los métodos printf() y format(); ambos realizan la misma función y tienen exactamente el mismo formato:

format (String formato, Object... datos) printf (String formato, Object... datos)

- El argumento formato consiste en una cadena de caracteres con las opciones de formato que van a ser aplicadas sobre los datos a imprimir.
- Por otro lado, datos representa la información que va a ser enviada a la salida y sobre la que se va a aplicar el formato, siendo el número de estos datos variable. La sintaxis *Object …datos* indica que se trata de un número variable de argumentos, en este caso puede tratarse de cualquier número de objetos Java.
- A modo de ejemplo, dadas las siguientes instrucciones:

```
double cuad=Math.PI*Math.PI;
System.out.printf("El cuadrado de %1$.4f es %2$.2f",Math.PI, cuad);
```

Salida con formato: Sintaxis

- La cadena de formato puede estar formada por un texto fijo, que será mostrado tal cual, más una serie de especificadoras de formato que determinan la forma en que serán formateados los datos. En el ejemplo anterior, las expresiones %1$.4f y %2$.2f representan los especificaciones de formato para los valores Pi y cuadrado de Pi, respectivamente.
- Los especificadores de formato para números y cadenas deben ajustarse a la sintaxis:

%[posición_argumento$][indicador][mínimo][num_decimales] conversión

- Posición_argumento. Representa la posición del argumento sobre el que se va a aplicar el formato. El primer argumento ocupa la posición 1. Su uso es opcional.
- Indicador. Consiste en un conjunto de caracteres que determina el formato de salida. Su uso es opcional. Entre los caracteres utilizados cabe destacar: "-"., el resultado aparecerá alineado a la izquierda; y con '+', el resultado incluirá siempre el signo (sólo para argumentos numéricos).
- Mínimo. Representa el número mínimo de caracteres que serán presentados. Su uso también es opcional.
- num_decimales. Número de decimales que serán presentados, por lo que solamente es aplicable con datos de tipo float o double. Obsérvese que este valor debe venir precedido por un punto. Su uso es opcional.
- Conversión. Consiste en un carácter que indica cómo tiene que ser formateado el argumento.

Salida con formato: Sintaxis - Conversión

La tabla de la figura contiene algunos de los caracteres de conversión más utilizados:

Carácter	Función
's', 'S'	Si el argumento es null se formateará como "null". En cualquier otro caso se obtendrá argumento.toString()
'c', 'C'	El resultado será un carácter unicode
'd'	El argumento se formateará como un entero en notación decimal
'x', 'X'	El argumento se formateará como un entero en notación hexadecimal
'e', 'E'	El argumento se formateará como un número decimal en notación científica
'f'	El argumento se formateará como un número decimal

Salida con formato: Sintaxis - Ejemplo

- Por ejemplo, dadas las siguientes instrucciones:

```
double cuad=Math.PI*Math.PI;
System.out.printf("El cuadrado de %1$.4f es %2$.2f",Math.PI, cuad);
```

- La salida producida por pantalla será la siguiente:

El cuadrado de 3,1416 es 9,87

Salida con formato: Sintaxis - Fechas

- Sintaxis: %[posición_argumento$][indicador][mínimo]conversión
- El significado de los distintos elementos es el indicado anteriormente. En este caso, el elemento conversión está formado por una secuencia de dos caracteres. El primer carácter es 't' o 'T', siendo el segundo carácter el que indica cómo tiene que ser formateado el argumento.

Carácter	Función
'H'	Hora del día, formateada como un número de dos dígitos comprendido entre 00 y 23
'I'	Hora del día, formateada como un número de dos dígitos comprendido entre 01 y 12
'M'	Minutos de la hora actual, formateados como un número de dos dígitos comprendido entre 00 y 59
'S'	Segundos de la hora actual, formateados como un número de dos dígitos comprendido entre 00 y 59
'B'	Nombre completo del mes
'b'	Nombre abreviado del mes
'A'	Nombre completo del día de la semana
'a'	Nombre abreviado del día de la semana
'y'	Últimos dos dígitos del año
'e'	Día del mes formateado como un número comprendido entre 1 y 31

Salida con formato: Sintaxis - Ejemplo

- Por ejemplo, dadas las siguientes instrucciones:

```
Calendar c = Calendar.getInstance();

System.out.printf("%1$tH:%1$tM:%1$tS---%1$td de %1$tB" + " del %1$ty", c);
```

- La salida producida por pantalla será la siguiente (correspondiendo a la fecha actual):

```
12:56:43---05 de septiembre del 22
```

</> Entrada de Datos

Entrada de Datos

- La lectura de datos del exterior se gestiona a través de la clase InputStream. Un objeto InputStream está asociado a un dispositivo de entrada, caso de la entrada estándar (el teclado) podemos acceder al mismo a través del atributo estático in de la clase System.
- Sin embargo, el método read() proporcionado por la clase InputStream para la lectura de los datos, no nos ofrece la misma potencia que print o println para la escritura. La llamada a read() devuelve el último carácter introducido a través de dispositivo, esto significa que para leer una cadena completa sería necesario hacerlo carácter a carácter, lo que haría bastante ineficiente el código.
- Por ello, para realizar la lectura de cadenas de caracteres desde el exterior es preferible utilizar otra de las clases del paquete java.io: la clase BufferedReader.

Entrada de Datos

- La lectura de datos mediante BufferedReader requiere seguir los siguientes pasos en el programa:
- 1. Crear objeto InputStreamReader. Este objeto permite convertir los bytes recuperados del stream de entrada en caracteres. Para crear un objeto de esta clase, es necesario indicar el objeto InputStream de entrada, si la entrada es el teclado este objeto lo tenemos referenciado en el atributo estático in de la clase System:

```
InputStreamReader rd = new InputStreamReader(System.in);
```

- 2. Crear objeto BufferedReader. A partir del objeto anterior se puede construir un BufferedReader que permita realizar la lectura de cadenas:

```
BufferedReader bf =new BufferedReader(rd);
```

- 3. Invocar al método readLine(). El método readLine() de BufferedReader devuelve todos los caracteres introducidos hasta el salto de línea, si lo utilizamos para leer una cadena de caracteres desde el teclado devolverá los caracteres introducidos desde el principio de la línea hasta la pulsación de la tecla "enter":

```
String s= bf.readLine();
```

Entrada de Datos - Ejemplo

- El ejemplo crea un programa que solicita por teclado la introducción de una cadena de texto para, posteriormente, mostrarla por consola:

```
import java.io.*;
public class Entrada {
   public static void main(String[] args) throws IOException {
      String cad;
      InputStreamReader ir = new InputStreamReader(System.in);
      BufferedReader bf = new BufferedReader(ir);
      System.out.println("Introduzca su nombre:");
      cad = bf.readLine();
      System.out.println("Te llamas: " + cad);
   }
}
```

- Hay que mencionar un punto importante a tener en cuenta cuando se utilizan ciertos métodos de determinadas clases, se trata del hecho de que al invocar a estos métodos el programa puede lanzar una excepción. Aunque más adelante se tratará en profundidad el tema de las excepciones, cuando la llamada a un método de un objeto puede lanzar una excepción el programa que utiliza ese método está obligado a capturarla o a propagarla.
- Éste es el caso del método readLine() de BufferedReader, cuya llamada; puede lanzar la excepción IOException. En este ejemplo, se ha optado por relanzar la excepción, incluyendo la expresión throws IOException en la cabecera del método main().
- Cuando se utiliza readLine() para leer datos numéricos, hay que tener en cuenta que el método devuelve los caracteres introducidos como tipo String, por lo que deberemos recurrir a los métodos de las clases de envoltorio (se comentan más adelante, los métodos estáticos parseXxx(String) se utilizan para convertir el dato a número y poder operar con él.

Entrada de Datos - Ejemplo

- El siguiente código corresponde a un programa que realiza el cálculo del factorial de un número, incluyendo la lectura de dicho número por teclado a través del método readLine():

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

public class Factorial {
   public static void main(String[] args) throws IOException {
      String cad;
      long result = 1;
      BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));
      System.out.println("Introduzca el número");
      cad = bf.readLine(); //Obtiene el número en formato cadena
      long num=Long.parseLong(cad); //Convierte la cadena a número
      for(int i=1; i<=num; i++){
         result*=i;
      }
      System.out.println ( "El factorial de "+num+ " es "+result);
   }
}
```

</> Clases Envoltura

Clases De Envoltorio o Clases Envoltura

- Como vimos en la UD2, para cada uno de los tipos de datos básicos, Java proporciona una clase que lo representa. A estas clases se las conoce como clases de envoltorio, y sirven para dos propósitos principales:
- Encapsular un dato básico en un objeto, es decir, proporcionar un mecanismo para "envolver" valores primitivos en un Objeto para que los primitivos puedan ser incluidos en actividades reservadas para los objetos, como ser añadido a un vector o devuelto desde un método.
- Proporcionar un conjunto de funciones útiles para los primitivos. La mayoría de estas funciones están relacionadas con varias conversiones: convirtiendo primitivos a String y viceversa y convirtiendo primitivos y objetos String en diferentes bases, tales como, binario, octal y hexadecimal.

Clases De Envoltorio o Clases Envoltura

- Hay una clase de envoltorio para cada primitivo en Java. Por ejemplo, la clase de envoltura para int es Integer, la clase Float es de float y así todas. Recuerda que los nombres de los datos primitivos es simplemente el nombre en minúscula, a excepción de int que es Integer y char que es Character. La siguiente tabla nos muestra la lista de clases de envoltura en la API de Java.

Tipo	Clase de envoltura	Argumentos de conversión
boolean	Boolean	boolean o String
char	Character	char
byte	Byte	byte o String
short	Short	short o String
int	Integer	int o String
long	Long	long o String
float	Float	float, double o String
double	Double	double o String

Clases De Envoltorio

- Antes todas las clases envoltorio excepto Character proveían dos constructores: uno que tomaba un dato primitivo y otro que tomaba una representación String del tipo que estaba siendo construido.
- Pero en las últimas versiones ya no hace falta llamar a new Clase(), y basta con asignar el valor a la variable automáticamente, si usamos el constructor el IDE nos avisa de que está desactualizado y ha sido marcado para su "destrucción". Por ejemplo:

```
Integer i1 = new Integer( value: 42);
Float f1 = new Float( value: 3.14f);
Character c1 = new Character( value: 'c');
```



```
Integer i1 = 42;
Float f1 = 3.14f;
Character c1 = 'c';
```

Clases De Envoltorio: valueOf()

- El método estático valueOf() es suministrado en la mayoría de las clases de envoltorio para darle la capacidad de crear objetos envoltorio. También toma una cadena como representación del tipo de Java como primer argumento. Este método toma un argumento adicional, int radix, que indica la base (por ejemplo, binario, octal, o hexadecimal) del primer argumento suministrado, por ejemplo:

```
Integer i2 = Integer.valueOf("101011", 2);
// convierte 101011 a 43 y asigna el valor 43 al objeto Integer i2
Float f2 = Float.valueOf("3.14f");
// asigna 3.14 al objeto Float f2
```

Clases De Envoltorio: Herramientas de Conversión

- Como dijimos antes, la segunda gran función de un envoltorio es la conversión. Los siguientes métodos son los más usados.
- Los métodos xxxValue(). Se utilizan cuando se necesita convertir el valor de un envoltorio numérico a primitivo. Todos los métodos de esta familia no tienen argumentos. Hay 36 métodos. Cada una de las seis clases envoltorio, tiene seis métodos, así que cualquier número envoltorio puede ser convertido a cualquier tipo primitivo, por ejemplo:

```
Integer i2 = 42;  // Crea un nuevo objeto envoltorio
byte b = i2.byteValue();  // convierte el valor de i2 a byte
short s = i2.shortValue();// otro método de xxxValue para números enteros
double d = i2.doubleValue();// otro más

Float f2 = 3.14f; // crea un nuevo objeto envoltorio
short s = f2.shortValue(); // convierte el valor de f2 a short
System.out.println(s); // el resultado es 3 (truncado, no redondeado)
```

Clases De Envoltorio: Herramientas de Conversión

- Los seis métodos parseXxx() (uno por cada tipo de envoltorio) son muy parecidos a los de valueOf(). Los dos toman una cadena como argumenta arrojando NumberFormatException (comúnmente denominado NFE) si el argumento cadena no esta apropiadamente formateado, y puede convertir objetos de cadena a diferentes bases (radix), cuando el dato primitivo es cualquiera de los cuatro tipos de números enteros. La diferencia entre ambos métodos es:
- parseXxx() devuelve el primitivo.
- valueOf() devuelve el recién creado objeto envoltorio del tipo invocado en el método.

```
double d4 = Double.parseDouble("3.14");// convierte un String a un primitivo
System.out.println("d4 = " + d4);// el resultado es d4 = 3.14
Double d5 = Double.valueOf("3.14"); // crea un objeto Double
System.out.println(d5 instanceof Double); // el resultado es "true"

long L2 = Long.parseLong("101010", 2);  // binary String a long
System.out.println("L2 = " + L2); // resultado es: L2 = 42
Long L3 = Long.valueOf("101010", 2); // binary String a Long objecto
System.out.println("L3 value = " + L3);  // resultado es: L3 value = 42
```

Clases De Envoltorio: Herramientas de Conversión

- toString(): La clase alfa, o sea, la primera clase de Java, Object, tiene un método toString().Así que todas las clases de Java tienen también este método. La idea de toString() es permitirte conseguir una representación mas significativa del objeto. Por ejemplo, si tienes una colección de varios tipos de objetos, puedes hacer un bucle a través de la colección e imprimir algunas de las representaciones más significativas de cada objeto usando toString(), que está presente en todas las clases. Hablaremos más de toString() en UD7, ahora vamos a centrarnos en como funciona toString() en las clases envoltorio, que como ya sabemos, están marcadas como clases finales (ya que son inmutables). Todas las clases envoltorio devuelven una cadena con el valor del objeto primitivo del objeto, por ejemplo:

```
Double d = 3.14;
System.out.println("d = "+ d.toString() ); // result is d = 3.14
```

- Todas las clases envoltorio suministran un método sobrecargado, que toma un número primitivo apropiado (Double.toString() toma tipo de un un double, Long.toString() toma un long, etc...) y naturalmente devuelve una cadena.

```
String d = Double.toString(3.14); // d = "3.14"
```

- Para terminar, los enteros y los long suminstran un tercer método, es estático. Su primer argumento es el dato primitivo, y el segundo argumento es el radix. El radix le dice al método como coger el primer argumento, lo que significa que si el radix es 10 (base 10, por defecto), lo convertirá a la base sumnistrada, y devolverá una cadena, por ejemplo:

```
String s = "hex = "+ Long.toString(254,16); // s = "hex = fe"
```

Clases De Envoltorio: Herramientas de Conversión

- toXxxString() (Binario, Hexadecimal, Octal): Las clases envoltorio Long e Integer te permiten convertir números en base 10 a otras bases. Estas conversiones toman un int o un long y devuelven una cadena que representa la conversión del número. Por ejemplo:

```
String s3 = Integer.toHexString(254);  // convert 254 to hex
System.out.println ("254 is " + s3);  // result: "254 is fe"
String s4 = Long.toOctalString(254); // convert 254 to octal
System.out.print("254(oct) ="+ s4); // result: "254(oct) =376"
```

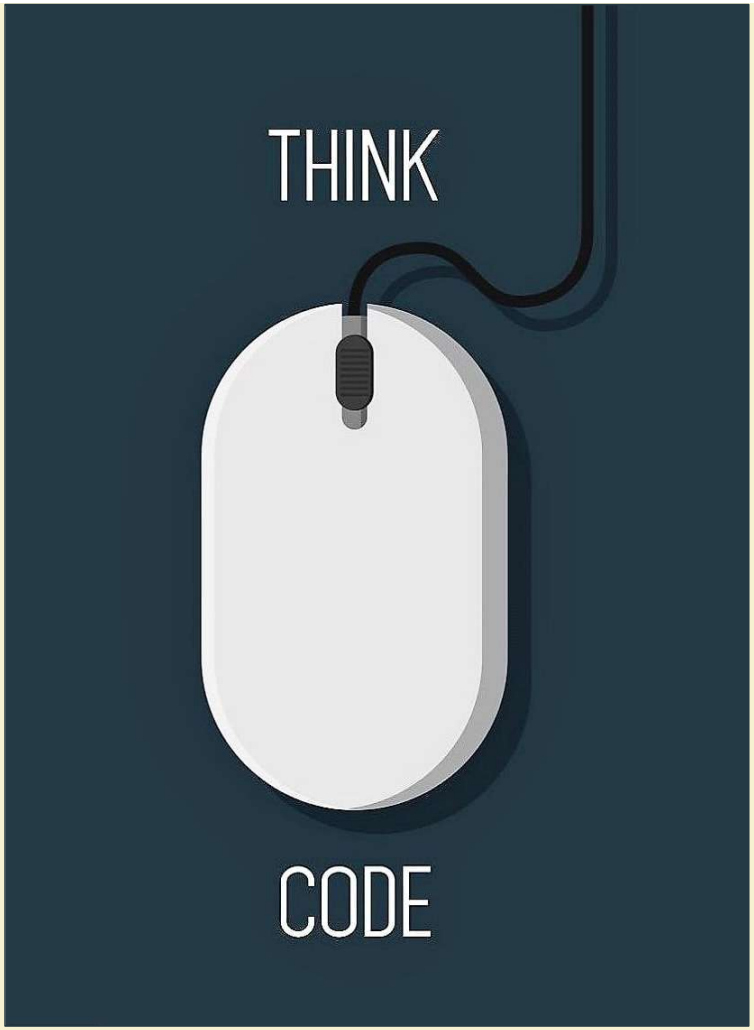
Clases De Envoltorio: Resumen

- En esencia, los métodos para la conversión de envoltorios son:
- Primitive xxxValue(), convierte un envoltorio a primitivo.
- Primitive parseXxx(String), convierte una cadena a un primitivo.
- Envoltorio valueOf(String), convierte cadena a envoltorio.
- String toString() convierte un envoltorio a cadena.

¿Preguntas?

05

Unidad Didáctica 5: Herencia y Polimorfismo



Módulo: Programación

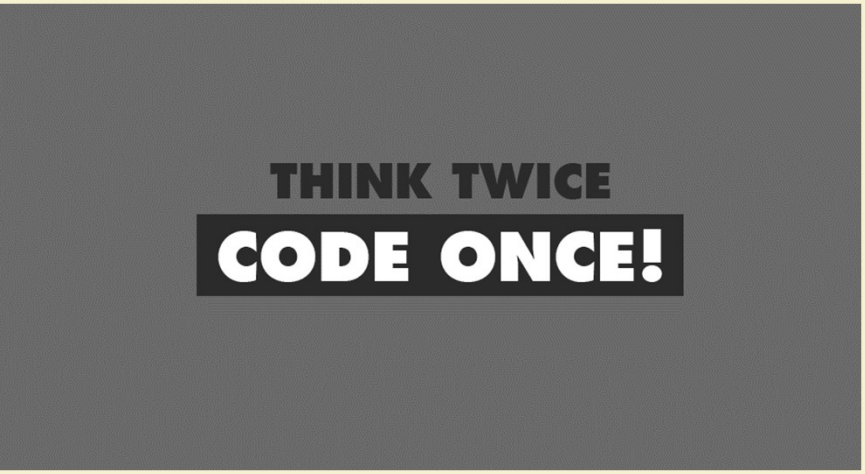
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

Índice



Adaptación

- Entre los aspectos aportados por la POO destaca la reutilización del código para crear nuevas clases adaptando otras ya existentes y probadas, lo que obtenemos una mayor seguridad.
- Hay dos técnicas de adaptación:
- La composición (delegación): consiste en construir una clase que está compuesta por otras.
- La herencia (extensión) de clases: nos permite reutilizar lo que ya está escrito y estructurar el código de un modo más cercano a las estructuras mentales del programador. La herencia está soportada por dos aspectos: el enlace dinámico y el polimorfismo.

Composición

- La composición es el concepto más sencillo, consiste en utilizar objetos de clases ya definidas como atributos de otra clase. La composición consiste en utilizar objetos de clases ya definidas como atributos de otra clase, es decir, componer una clase utilizando otras ya existentes.
- Por ejemplo, una Persona es un objeto que contiene en su interior otros objetos como la fecha de nacimiento (de tipo Fecha) o su dirección (de tipo DirecciónPostal).

```
public class Fecha {
   private int dia, mes, año;
}

public class Direccion {
   private String calle;
   private int numero;
   private String localidad;
   private String provincia;
   private String cp;
}

public class Persona {
   private String nombre;
   private String dni;
   private Fecha fechaNacimiento;
   private Direccion direccion;

}
```

Herencia: ¿Qué es?

- La herencia es uno de los mecanismos de la POO por medio del cual una clase se deriva de otra de manera que extiende su funcionalidad. Así, dada una clase existente, llamada superclase o clase padre, se crea una nueva clase, llamada subclase o clase hija, que extiende a la anterior y hereda de ella todos sus métodos y atributos.
- La herencia es una funcionalidad potente de un lenguaje orientado a objetos, pero a veces no se utiliza como se debe. Se pueden contemplar dos categorías de relaciones entre dos clases. Podemos tener la relación «es un tipo de» y la relación « se trata de».
- La relación de herencia debe utilizarse cuando es posible aplicar la relación «es un tipo de» entre dos clases.

Herencia - Ejemplo

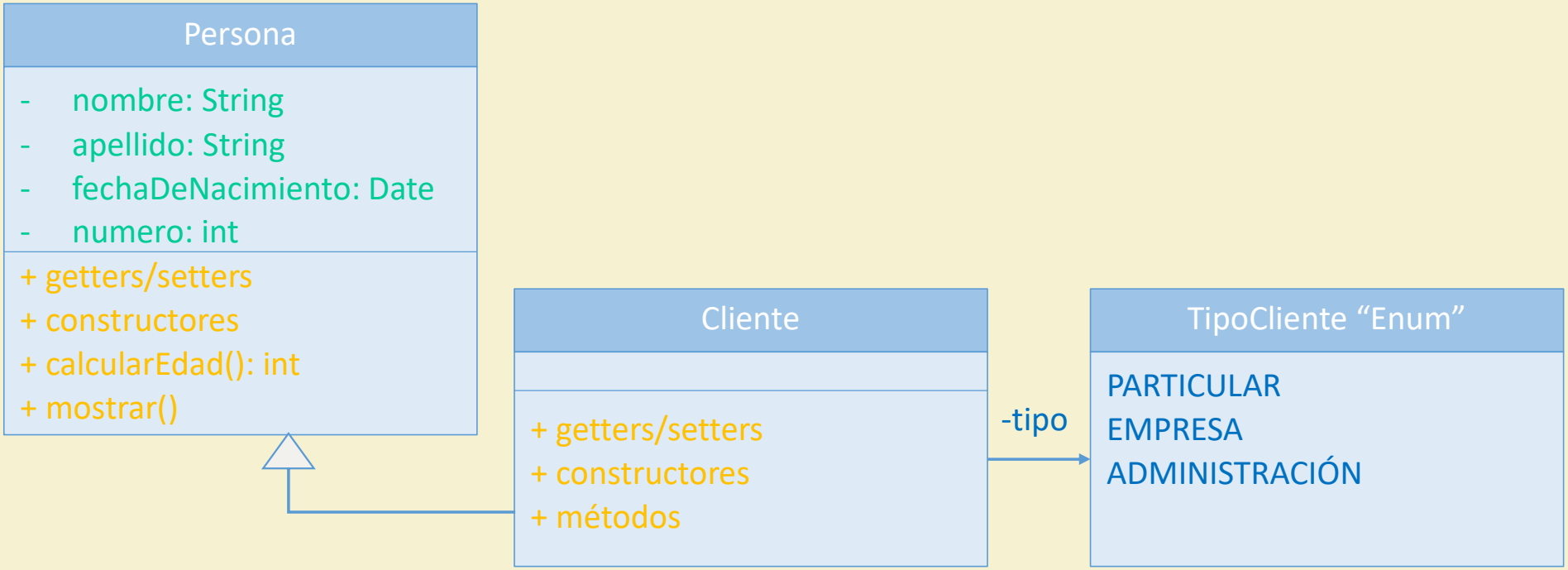
- Veamos un ejemplo con tres clases: Persona Cliente y Pedido.
- Probamos la relación «es un tipo de» para cada una de las clases.
  - Un pedido es un tipo de cliente.
  - Un pedido es un tipo de persona.
  - Un cliente es un tipo de pedido.
  - Un cliente es un tipo de persona.
  - Una persona es un tipo de cliente.
  - Una persona es un tipo de pedido.
- De entre todos estos intentos, solo uno nos resulta lógico: un cliente es un tipo de persona. Por lo tanto, podemos considerar una relación de herencia entre estas dos clases.

Herencia

- Sintaxis: public class ClaseHija extends ClasePadre implements <lista interfaces> {...}
- La puesta en práctica es muy sencilla a nivel del código, ya que en la declaración de la clase basta con especificar la palabra clave extends seguida del nombre de la clase que se desea heredar. Al no aceptar Java la herencia múltiple, solo podemos especificar un único nombre de clase básica. En el interior de esta nueva clase, podemos:
  - Utilizar los campos heredados de la clase básica (con la condición, por supuesto, de que su visibilidad lo permita, es decir, que no deben ser privados).
  - Añadir nuevos campos.
  - Enmascarar un campo heredado declarándolo con el mismo nombre que el usado en la clase base. Se debe utilizar esta técnica con moderación.
  - Usar un método heredado siempre que su visibilidad lo permita.
  - Sustituir un método heredado al declararlo idéntico (misma firma).
  - Sobrecargar un método heredado creándolo con una firma diferente.
  - Añadir un nuevo método.
  - Añadir uno o varios constructores.

Herencia - Ejemplo

- A continuación, presentamos el ejemplo de la creación de la clase Cliente; la siguiente podría ser una representación UML simplificada:



Herencia - Ejemplo

- Esta clase hereda de la clase Persona, a la cual se añade el campo tipo y los métodos de acceso correspondientes.

```
public class Cliente extends Persona{

    private TipoCliente tipo;

    public TipoCliente getTipo(){
        return tipo;
    }

    public void setTipo(TipoCliente t){
        tipo = t;
    }
}
```

```
Cliente c =new Cliente();
c.setApellido("ENI");
c.setNombre("");
c.setFechaDeNacimiento(LocalDate.of(1981,05,15));
c.setType(TipoCliente.EMPRESA);
c.mostrar();
```

- Ya se puede utilizar la clase, la cual presenta todas las funcionalidades definidas en la clase Cliente más las heredadas de la clase Persona.

Herencia - Ejemplo

- Ejemplo: Construcción de la clase semáforo extendiendo (heredando) la clase Bombilla.

```
public class Semaforo extends Bombilla {
   private int color;
   public static final int ROJO = 1;
   public static final int AMBAR = 2;
   public static final int VERDE = 3;

   public Semaforo (String m){
      //Atributos heredados, hay que cambiar el acceso a public o protected
      marca = m;
      potencia = 10;
      encendida = true;
      color = ROJO;
   }
   //Todos los métodos públicos de bombilla se heredan y es como si estuviesen aquí

   public void setColor(int c) {
      if(c==ROJO || c==AMBAR || c==VERDE){
         color=c;
      }
   }
}
```

Constructores: método super

- En ocasiones se necesita invocar a algunos de los constructores de la clase padre desde un método constructor de la clase hija.
- En este caso se usa el método super(<argumentos>), que invocará aquel constructor de la clase padre cuyo tipo y número de parámetros coincida con los argumentos que se pasan como parámetros.
- Ejemplo: Constructor de la clase Semáforo:

```
public Semaforo (String m){
   super(m);//Llama al constructor de Bombilla
   color = ROJO;
}
```

Constructores: método super

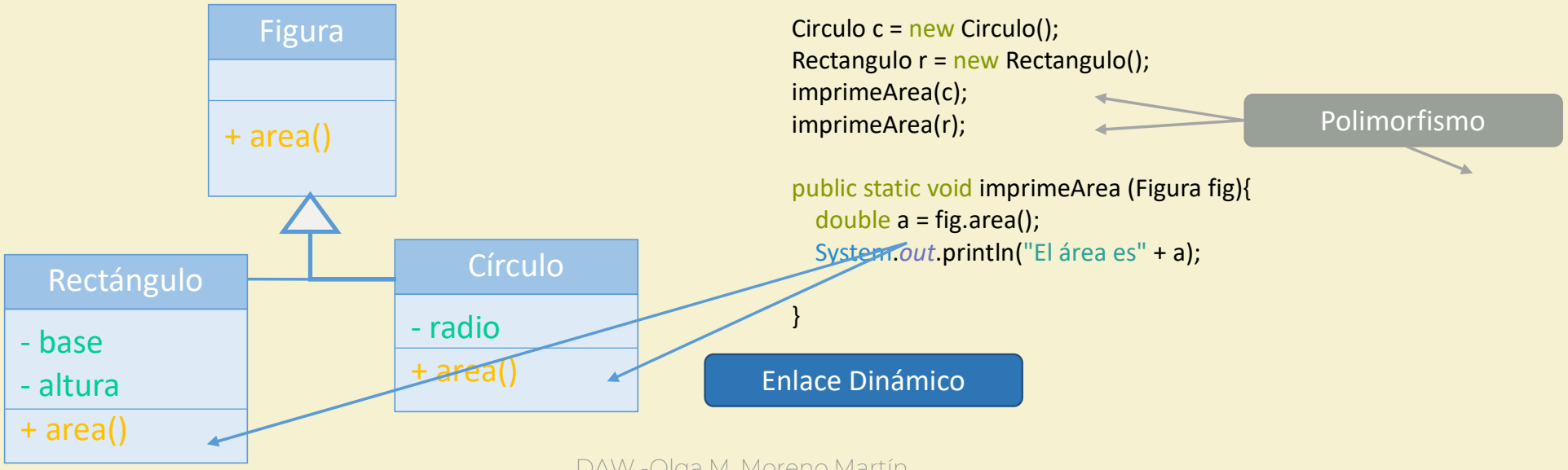
- A tener en cuenta:
- El uso del método super es similar al uso del método this, pero en vez de llamar a al constructor correspondiente de la misma clase, lo hace para la clase padre.
- Sólo se puede llamar una vez al método super.
- Si se llama al método super, la llamada debe ser la primera instrucción del constructor de la clase hija.
- Si no se llama al método super, se llama implícitamente al constructor por defecto de la clase Padre.
- Si no se llama a un constructor de la clase padre y no existe un constructor por defecto en la clase padre el compilador da un error.

Herencia: Visibilidad

- La clase hija hereda todos los miembros de una clase padre, pero dependiendo de los modificadores de visibilidad de estos miembros en la clase padre se puede acceder a ellos directamente o no desde la clase hija.
- public: Todo miembro public es accesible de forma directa desde cualquier clase, tanto clases que declaren objetos de ella como clases que hereden de ella.
- private: Sólo se puede acceder a los miembros private desde la propia clase donde están definidos. Por tanto desde las clases derivadas no se puede acceder a estos miembros.
- protected: Se puede acceder a los miembros protected sólo desde los miembros de la propia clase, desde todas las clases que hereden de ella y desde todas las clases que compartan paquete. Se puede decir que protected es como un private que se puede heredar.

Herencia: Polimorfismo

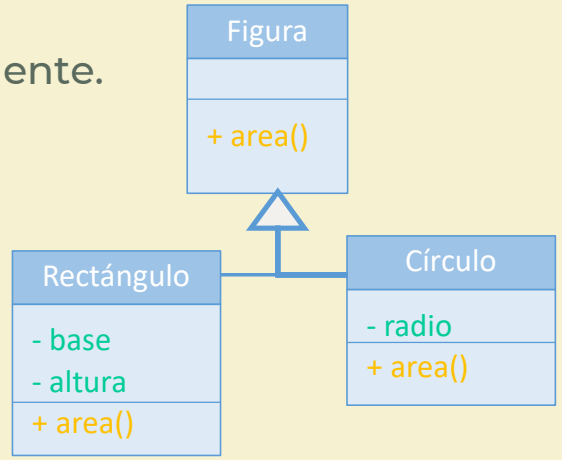
- El Polimorfismo es la propiedad mediante la cual un objeto puede adquirir formas distintas.
- El Enlace Dinámico es el mecanismo mediante el cual la máquina virtual decide en tiempo de ejecución cuál de las implementaciones de un método se ejecutará dependiendo del tipo del objeto (clase padre o clase hija) que invoca el método.



Herencia: Polimorfismo – Compatibilidad de Tipos

- La conversión hacia clases superiores en la jerarquía se hace implícitamente.
- Casting automático (implícito):

```
Circulo c = new Circulo();
Figura a = c;
```



- Conversión hacia las subclases ha de ser explícita Casting explícito:

```
Figura a = new Figura();
Circulo c = Circulo(a);
```

- No es posible convertir una referencia de tipo clase en otra si no existe una relación de herencia entre ambas:

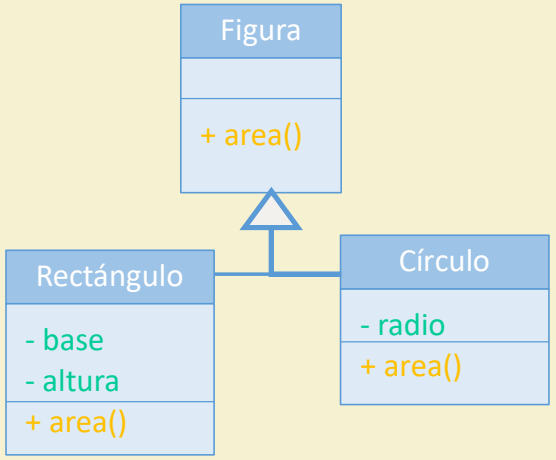
```
Circulo c = new Circulo();
Rectangulo d = (Rectangulo) c;
```

Error de Compilación

Herencia: Polimorfismo – Compatibilidad de Tipos

- Para comprobar la clase de un objeto dado, usamos el operador instanceof

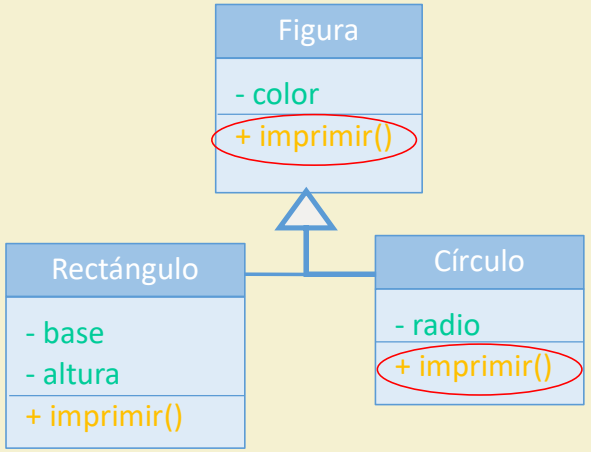
```
public static void imprimeAtributos (Figura fig){

   if(fig instanceof Rectangulo){

      Rectangulo r = (Rectangulo) fig;

      System.out.println("Base:" + r.getBase());
      System.out.println("Altura: "+ r.getAltura());

   }else if(fig instanceof Circulo){

      Circulo r = (Circulo) fig;
      System.out.println("Radio: "+ c.getRadio());

   }else{

      System.out.println("La figura no es un rectángulo ni un círculo");
   }
}
```



Herencia: Polimorfismo – Redefinir Métodos

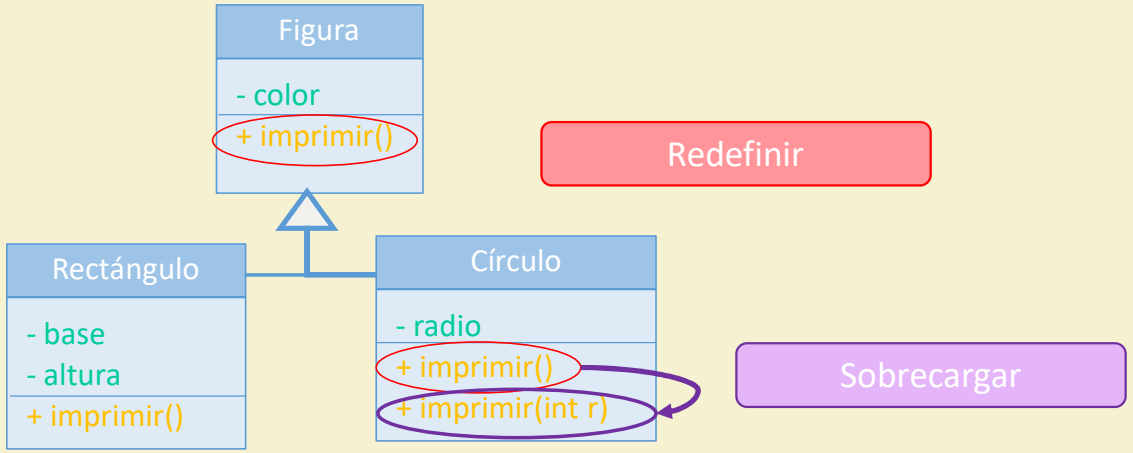
- Una subclase puede modificar los métodos que ha heredado de la clase padre creando un método con el mismo nombre, tipo de retorno y parámetros. A esto se le denomina Redefinición del Método.

```
public class Figura {
   private String color;
   public void imprimir(){
      System.out.println("Figura de color " + color);
   }
}

public class Circulo extends Figura {
   public void imprimir() {
      System.out.println("Figura de color" + color);
      System.out.println("Circulo de radio" + radio);
   }
}
```



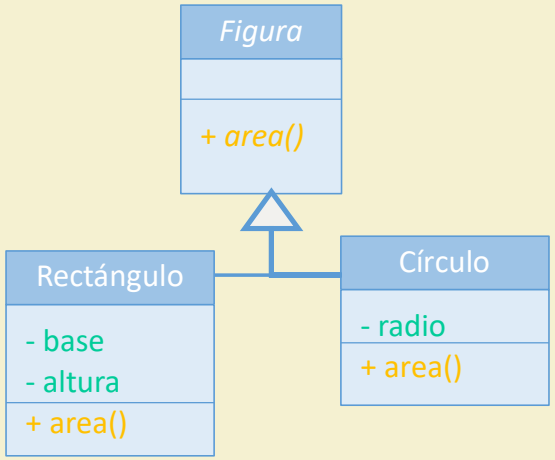
Herencia: Polimorfismo – Redefinir Métodos

- Error común: Confusión entre Redefinición y Sobrecarga.
- Redefinición: La clase hija define exactamente el mismo método, con el mismo tipo de vuelta, el mismo nombre y los mismos parámetros.
- Sobrecarga: Posibilidad de que coexistan métodos con el mismo nombre pero distintos parámetros).



Herencia: Clases Abstractas

- Una Clase Abstracta es una clase cuya definición es incompleta debido a que uno o varios de sus métodos no tiene código.
- Las clases abstractas se usan cuando queremos implementar la Herencia Forzada, es decir, cuando queremos forzar a las clases hijas a implementar un determinado método.



En UML, las clases y métodos abstractos se indican en cursiva

```
public abstract class Figura {
   private String color;
   …
public abstract double area();
}
public class Circulo extends Figura {
   …
public void area(){
   return 3.1415*radio*radio;
}
```

Herencia: Clases Abstractas

- Consideraciones:
- La superclase abstracta determina la signatura de los métodos que las subclases deben implementar.
- Cualquier clase que contenga uno o varios métodos abstractos es una clase abstracta
- No se pueden instanciar objetos de clases abstractas.
- No se pueden declarar constructores abstractos ni métodos estáticos abstractos.
- Cualquier subclase de una superclase abstracta debe implementar todos los métodos abstractos de la superclase o ser declarada también como abstracta.

Herencia: Clases finales

- Una clase final es una clase que no puede ser derivada.
- Para declarar una clase como final, simplemente usaremos el modificador final en la cabecera de la declaración de la clase.

```
public final class ClaseNoDeribable{...}
```

- Un método final es aquel que no se puede redefinir en clases derivadas, manteniendo siempre la implementación dada en la clase que lo define.
- Para declarar una clase como final, simplemente usaremos el modificador final en la cabecera de la declaración de la clase.

```
public class ClaseEjemplo {
   public final void metodoNoRedefinible() {}
      . . .
//Elcódigo aquí insertado será siempre común para todas las subclases.
```

Herencia: Clases Selladas

- Las clases selladas son clases finales, para las que es posible autorizar explícitamente clases hija. Esta es una característica muy interesante cuando desea controlar las posibles herencias para una clase determinada.
- La palabra clave sealed permite definir la clase sellada y permits permite definir las clases que pueden heredar de la clase sellada.
- Cronológicamente en el desarrollo, el desarrollador escribirá su código y, al final, por ejemplo para proteger las operaciones sensibles, colocará estas palabras clave para sellar la operación.
- Las clases autorizadas a heredar de una clase hija, deben indicar ellas mismas en su firma si son definitivas. (final), selladas (sealed) o no selladas (non-sealed) :

Herencia: Clases Selladas - Ejemplo

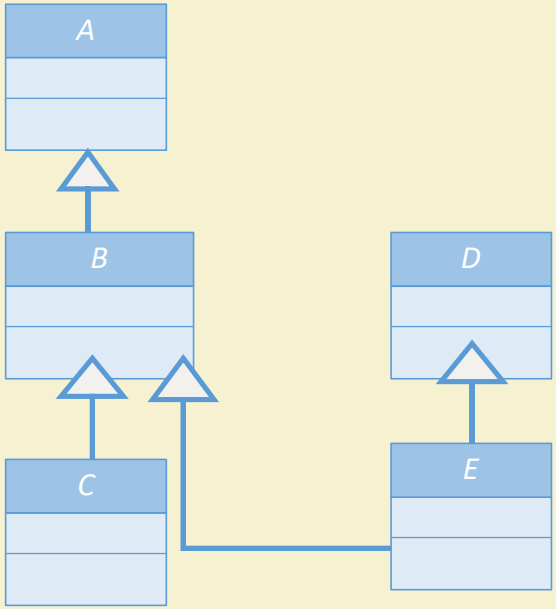
```
// La declaración de una clase sellada, tiene la siguiente forma:
    public sealed class Persona permits Cliente, Proveedor{
   ...
 }

//La declaración de una clase final no permite su redefinición en una subclase
 public final class Cliente extends Persona {
...
}
//Una clase no sellada (non-sealed) recupera el comportamiento clásico de una clase
estándar.

      public non-sealed class Proveedor extends Persona {
...
}
```

Herencia: Interfaces

- El concepto de herencia puede aplicarse también a la interfaces, además de a las clases.
- Podemos derivar por tanto una interfaz de otra interfaz padre ya definida.
- Cuando derivamos una interfaz, la interfaz hija hereda todos los métodos abstractos y constantes declaradas en la interfaz o interfaces padre.
- Una clase que implemente una interfaz, tiene que implementar todos los métodos de ésta, tanto los propios como los heredados.
- Cuando una clase implementa una interfaz, implementa a todas su interfaces padre.

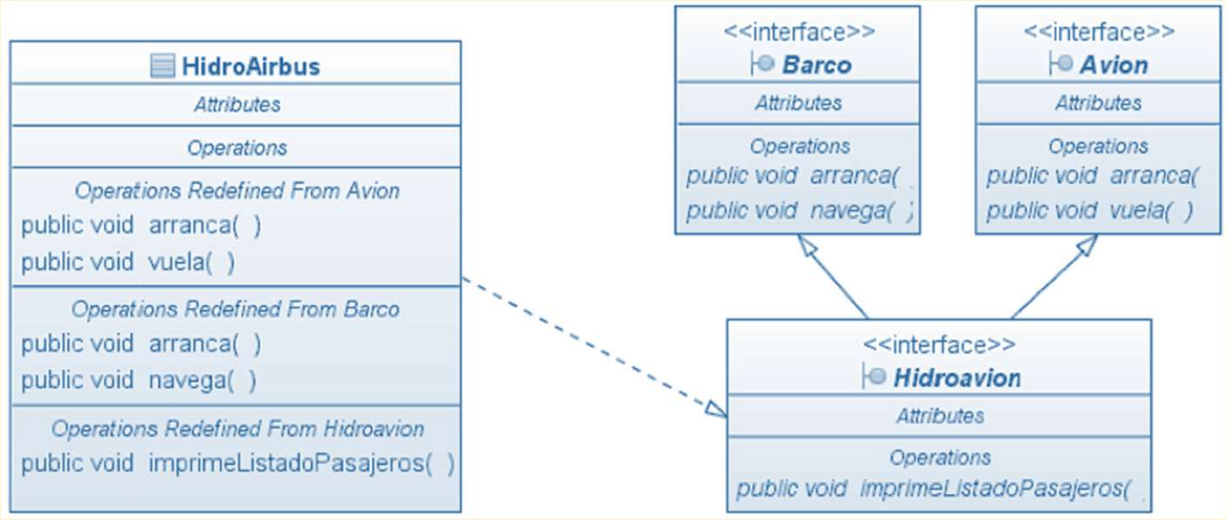
Herencia: Interfaces

- **Sintaxis public interface InterHija extends InterPadre1, InterPadre2 , … { … }**
- **Ejemplo:**
    - interface A
    - interface B extends A
    - interface C extends B
    - interface D
    - interface E extends B,D



Herencia: Interfaces – Herencia Múltiple

- Un diferencia importante entre herencia de clases y herencia de interfaces es que en el caso de las interfaces se permite la herencia múltiple.
- Es decir, una interfaz puede heredar de tantas interfaces padre sea necesario.



Herencia: La Clase Object

- Todas las clases en Java derivan (heredan) por defecto de una misma clase llamada Object.
- Esta clase contienen el conjunto de métodos y atributos que debe contener cualquier objeto en Java.
- De todas ellas sólo veremos las más importantes para el desarrollo de la asignatura, que son:
  - public boolean equals (Object o)
  - public int hashCode()
  - public String toString()
  - protected Object clone()

Herencia: La Clase Object - equals

- Cabecera del método: public boolean equals (Object o)
- Compara dos objetos, devolviendo true si son el mismo (identidad ==).
- Se puede redefinir en cualquier clase para que devuelva true si son objetos distintos pero con los idénticos valores en sus atributos (igualdad).
- Ejemplo: En la clase Bombilla:

```
public boolean equals(Object obj) {
   Bombilla b = (Bombilla)obj;
   return (marca.equals(b.marca)) && (encendida == b.encendida) && (potencia == b.potencia);
}

//Ejemplo de uso:

if(b1.equals(b2)){…}
```

Herencia: La Clase Object - hashCode

- Cabecera del método: public int hashCode()
- Cuando se redefine el método equals, normalmente es necesario redefinir también el método hashCode, sobre todo cuando se va a hacer uso de *Colecciones del JCF de tipo HashMap y HashSet*.
- Este método devuelve un número entero que identifica al objeto inequívocamente.
- Ejemplo en la clase Bombilla:

```
public int hashCode() {
   return (marca.hashCode() + potencia);
}
```

Herencia: La Clase Object - toString

- Cabecera del método: public String toString ( )
- Devuelve una representación del objeto como una cadena de caracteres (String). Si no se redefine, devuelve por defecto la cadena:

Nombredelaclase@númerodeidentificación (donde númerodeidentificación es el código hash).

- Se invoca automáticamente cuando se quiere visualizar el contenido de un objeto y la referencia a ese objeto forma parte de una operación con el operador +, donde algún operando es una cadena de texto.
- Para visualizar los atributos hay que redefinir este método.
- Ejemplo: En la clase Bombilla:

```
public String toString(){
   return "Datos de la bombilla: " + "\n Marca: "+ marca +
"\n Encendida: "+ encendida + "\n Potencia: " + potencia + "\n";
}
```

Herencia: La Clase Object - clone

- Cabecera del método: protected Object clone()
- Crea y devuelve una copia del objeto que lo invoque.
- Para poder clonar un objeto de una clase es necesario realizar previamente las siguientes operaciones.
- Implementar en la clase la interfaz Cloneable
- Controlar las posibles excepciones (*CloneNotSupportedException*)
- Redefinir en la clase el método clone ( ) y declararlo público.
- Ejemplo: En la clase Bombilla:

```
public class Bombilla implements Cloneable{ …

    public Object clone() throws CloneNotSupportedException {
       return super.clone();
    }
```

Herencia: Identidad e igualdad

- Igualdad:
  - Dos objetos son iguales cuando tienen los mismos valores en los atributos.
  - Para comparar si dos objetos son iguales, hay que usar el método equals.

```
Bombilla b1 = new Bombilla(”Siemens");
Bombilla b2 = new Bombilla("Siemens");
if (b1.equals(b2)) {
   System.out.println("Los objetos son iguales");
}
```

- Identidad:
  - Dos objetos son idénticos cuando son el mismo objeto, es decir, cuando una referencia es un alias de la otra.
  - Para comparar si dos objetos son idénticos, usamos el operador “==“.

```
Bombilla b1 = new Bombilla("Phillips");
Bombilla b3 = b1;
if (b1==b3){
   System.out.println("Son el mismo objeto");
}
```

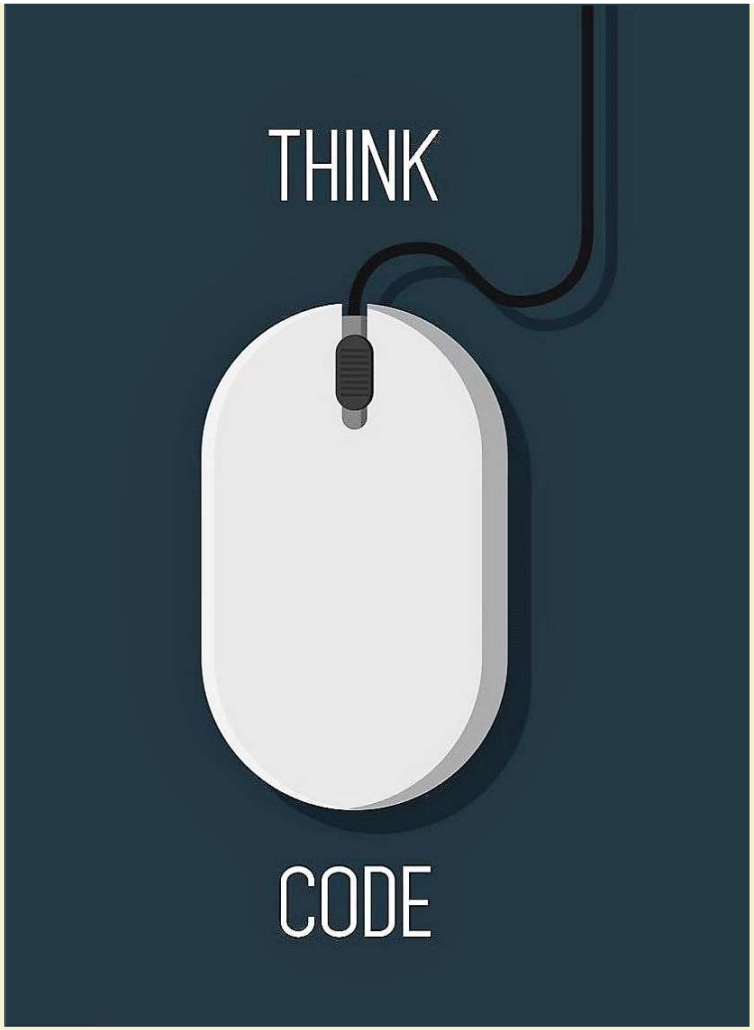
Herencia: Copia de Objetos y Tipos Primitivos

- Como ya hemos visto anteriormente, la asignación directa de referencias con el operador “=” (obj1=obj2) hace que la referencia de la izquierda sea un alias de la referencia de la derecha. Es decir, hacemos que dos referencias apunten al mismo objeto sin realizar ningún tipo de copia → IDENTIDAD.
- El método clone() (obj1 = (Clase)obj2.clone()) es aplicable sólo a objetos y se usa cuando queremos copiar el contenido de un objeto (obj2) en otro distinto (obj1) →IGUALDAD.
- El operador “=” con tipos básicos copia el contenido de un tipo básico en el otro, lo que es equivalente a clone() para objetos.

¿Preguntas?

06

Unidad Didáctica 6: Trabajando con vectores La Clase Array



Módulo: Programación

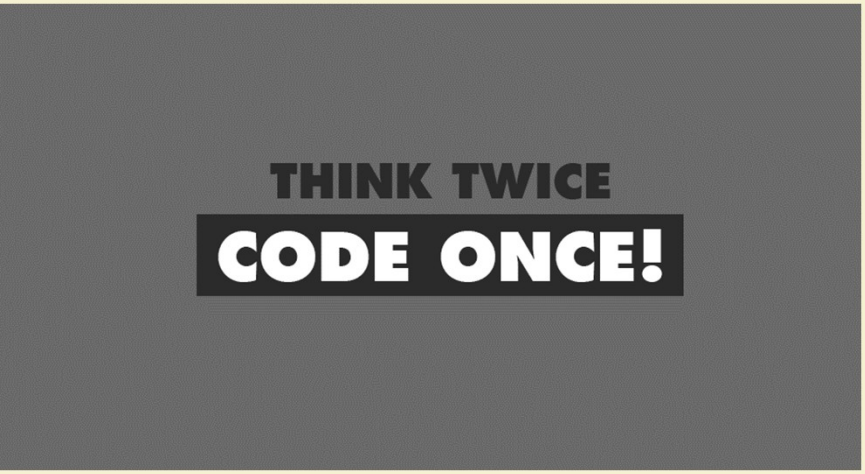
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

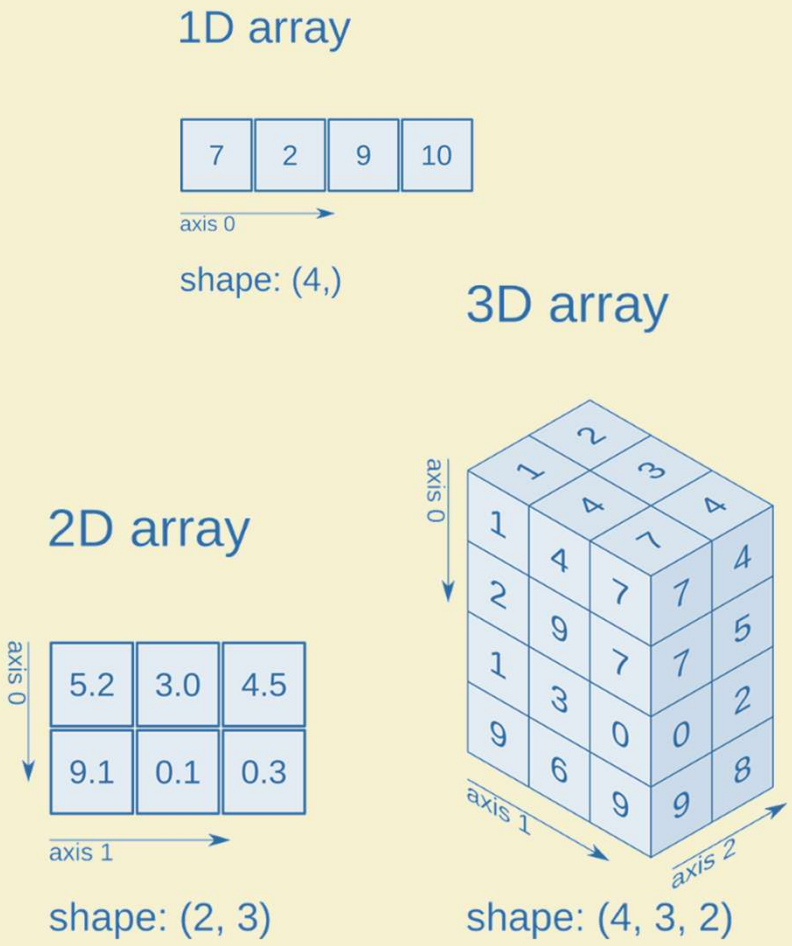
En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

Índice



El concepto de Array o Vector

- Un vector es una estructura secuencial que almacena un conjunto de datos, todos del mismo tipo y que se distinguen por la posición (índice) que ocupan en la misma.
- Una tabla, un vector o un array en Java, es un objeto que representa una colección indexada de elementos. Dichos elementos pueden ser valores u objetos (referencias).
- Un array puede contener referencias a otros arrays, ya que éstos son objetos, constituyendo así estructuras matriciales de múltiples dimensiones.



Arrays: Declaración

- Sintaxis:

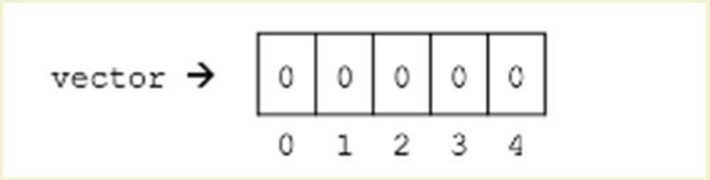
```
Tipo [ ] nombre;
nombre = new Tipo [d];

Tipo [ ]  nombre = new Tipo[d];

Tipo [ ] nombre = {valor1, valor2, ... valorN};
```

- La dimensión(d) de la tabla puede se un literal, una constante, una variable o en general una expresión de tipo entero. Ejemplo:

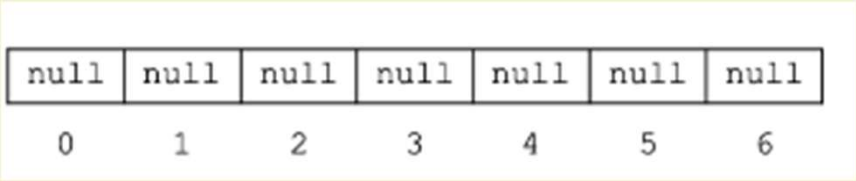
```
int tam = 9;
int [] array1 = new int[tam];
int [] array2 = new int[10];
int [] array3 = new int[Bombilla.MAX_POT];
```

Arrays: Declaración

```
int [] array = new int[5];
```




```
Bombilla [] luces = new Bombilla[7];
```




```
double [] v_reales = {1.3, 5.6, 0.5};
```

Arrays: Acceso

- Todos los arrays tienen por defecto un atributo público denominado length, que almacena el tamaño de la tabla.
- Por ejemplo, el número de elemento de una array a sería: a.length
- Observa que no tiene paréntesis, ya que se trata de un atributo, no de un método.
- Así, los arrays en Java se indexan desde 0 y hasta length-1.
- El acceso se hace mediante el operador corchetes:

Variable = ref[índice]; donde índice ∈ [0, length-1]

Arrays: Accesos - Ejemplo

```
//Dias de la semana
String[] weekDays = new String[7];
weekDays[0] = "Monday ";
weekDays[1] = "Tuesday ";
weekDays[2] = "Wednesday ";
weekDays[3] = "Thursday ";
weekDays[4] = "Friday ";
weekDays[5] = "Saturday ";
weekDays[6] = "Sunday ";

//Números primos
int[] primes = {2, 3, 5, 7, 11, 13, 17};

for (int i=0; i<7; i++){
  System.out.println(weekDays[i]);
  System.out.println(primes[i]);
}
```

Arrays: Accesos – For extendido

- Otra sintaxis del bucle for permite ejecutar un bloque de código para cada elemento contenido en un array o en una instancia de clase al implementar la interfaz Iterable. Es el bucle foreach o for extendido. La sintaxis general de esta instrucción es la siguiente:

```
for(tipo elementoDelArray : array){
   instrucción 1;
   ...
   instrucción n;
}
```

No hay noción de contador en esta estructura, ya que efectúa por sí misma las iteraciones en todos los elementos presentes en el array o la colección.

La variable declarada en la estructura (llamada elementoDelArray en el ejemplo) sirve para extraer uno a uno los elementos del array o de la colección para que el bloque de código pueda manipularlos. Por supuesto, hace falta que el tipo de la variable sea compatible con el tipo de los elementos almacenados en el array o la colección. Se debe declarar la variable obligatoriamente en la estructura for y no en el exterior. Solo se podrá utilizar en el interior de la estructura. No obstante, no tenemos que preocuparnos del número de elementos porque la estructura es capaz de gestionar por sí misma el desplazamiento en el array o la colección.

Arrays: Accesos – For extendido: Ejemplo

- Comparación entre for clásico y for extendido:

```
String [ ] array  = {  "rojo" "verde""azul" "blanco"};

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

for (String s: array) {
   System.out.println(s);
}
```

Arrays multidimensionales

- Un array multidimensional no es más que un array de arrays, es decir, un vector de objetos donde cada celda contiene una referencia a otro vector.
- Sintaxis:

```
Tipo []...[] ref = new Tipo [Dim1][Dim2]...[DimN];

//Habrá tantos [] como dimensiones
```

- Veremos las matrices (vectores bidimensionales) y lo vamos a hacer extensible a N dimensiones.

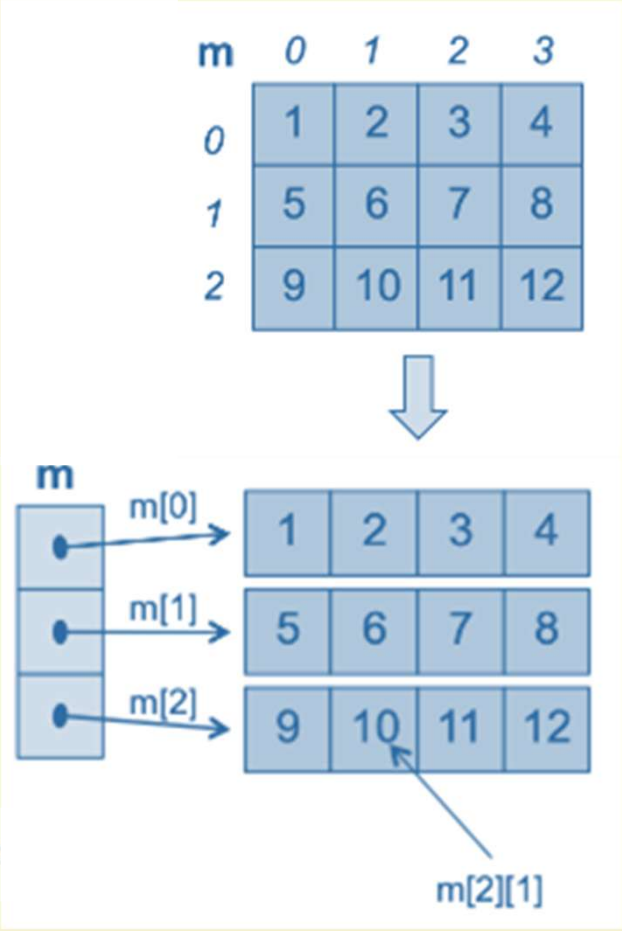
Arrays multidimensionales - Ejemplo

- Matriz 3x4 Tres declaraciones equivalentes:

```
float [][] m = new float[3][4];

float [][] m2 = new float [4][];
m2[0] = new float[4];
m2[1] = new float[4];
m2[2] = new float[4];
m2[3] = new float[4];

float [][] m3 = new float [4][];
for (int i = 0; i< m3.length; i++){
    m[i] = new float [4];
}
```

- Una vez declarada podemos recorrerla:

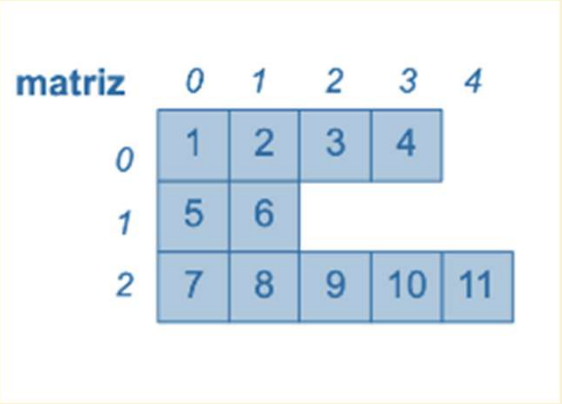
```
for (int i = 0; i< m.length; i++){
   for (int j=0; j < m[i].length; j++){
      m[i][j] = i*4+j+1;//inicializa la matriz
   }
}
```



Arrays multidimensionales Irregulares

- De las anteriores declaraciones podemos intuir que en Java es posible tener “matrices” en las que cada fila tenga un número distinto de columnas:

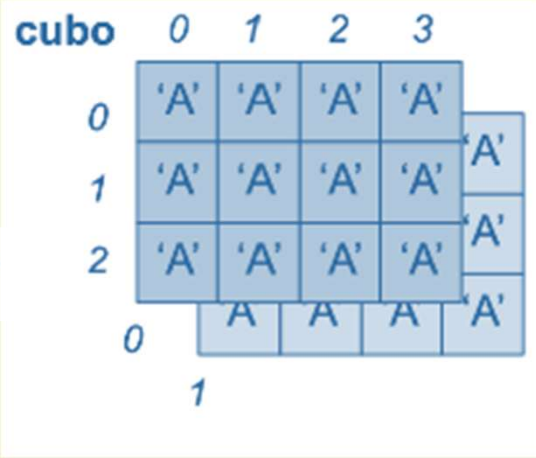
```
float [][] m2 = new float [3][];

m2[0] = new float[4];
m2[1] = new float[2];
m2[2] = new float[5];

int cont =1;
for (int i = 0; i< m2.length; i++){
   for (int j=0; j < m2[i].length; j++){
      m[i][j] = cont;
      cont++;
   }
}
```



Arrays Multidimensionales: Cubo

- El caso de los arrays de más de dos dimensiones es similar. Por ejemplo, la declaración de un array de caracteres de tres dimensiones y su inicialización sería:

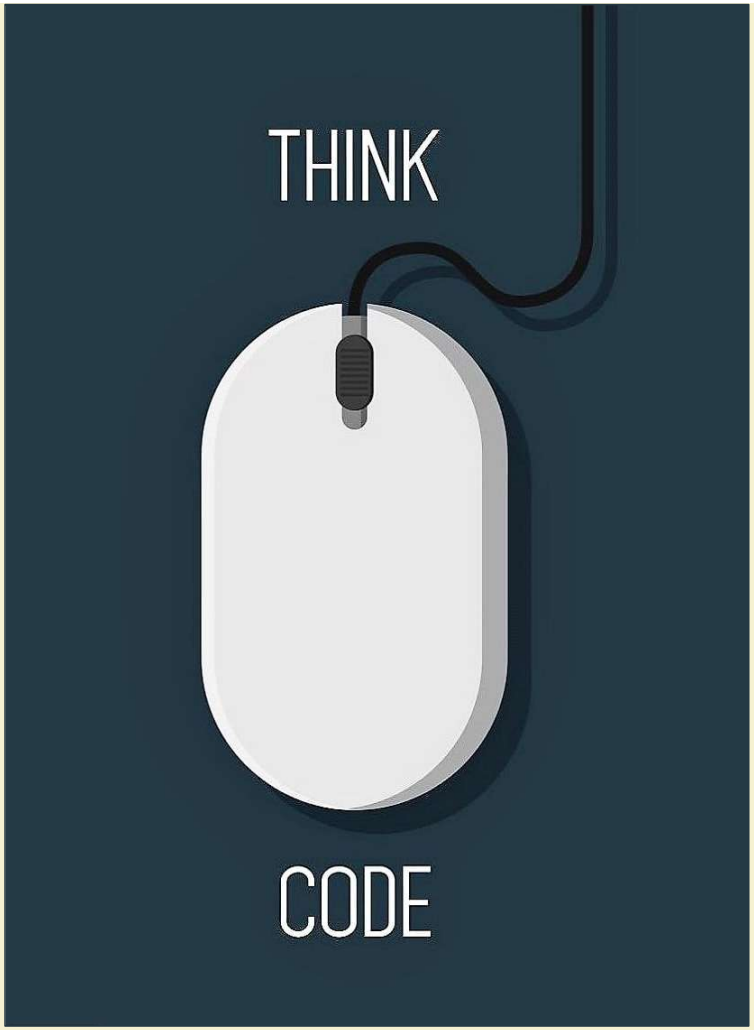
```
char [][][] cubo = new char[3][4][2];

for (int i = 0; i< cubo.length; i++){
   for (int j=0; j < cubo[i].length; j++){
      for (int k=0; cubo[i][j].length; k++){
         cubo[i][j][k] = 'A';
      }
   }
```



¿Preguntas?

07

Unidad Didáctica 7: Trabajando con cadenas La Clase String



Módulo: Programación

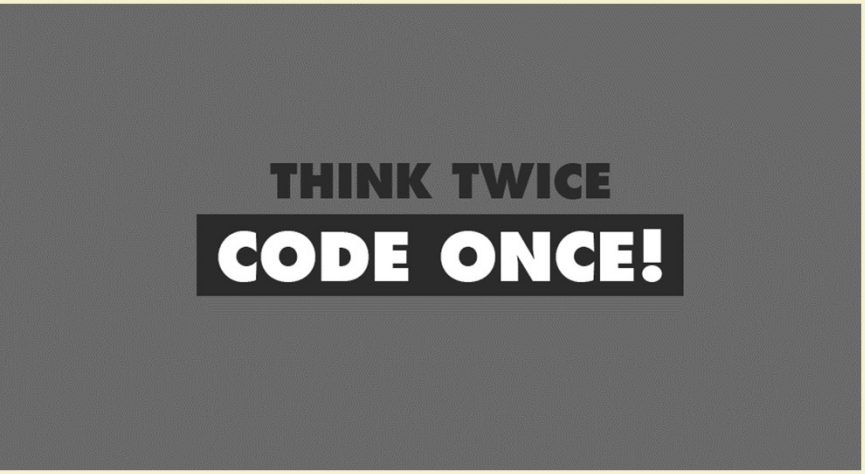
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

Índice



La Clase String

- La clase String es la clase definida en el paquete java.lang para el tratamiento de cadenas de caracteres.
- Es una clase public y final (sus métodos no se pueden redefinir), hereda de la clase Object e implementa las interfaces Comparable y Serializable.
- Los objetos de tipo String son inmutables, es decir constantes, luego una vez creados sólo se pueden leer, nunca modificar. Existe otro tipo de cadenas de la clase StringBuffer que permite ciertas modificaciones sobre los objetos.

String: Crear Objetos

- La clase String dispone de varios constructores, pero por el momento sólo usaremos los dos siguientes:
- String(): Construye la cadena de caracteres vacía:

```
String nuevo = new String();
```

- String(String cad): Construye la cadena de caracteres cuyo valor es pasado como argumento. En otras palabras el String creado es una copia del que se pasa como parámetro.

```
String str = new String("Hello world!");
String copia = new String(str);
```

- String(char [ ] cad): Construye la cadena de caracteres a partir del array (o tabla) de caracteres que recibe como parámetro.

```
char [] datos = {'a', 'b', 'c'};
String s = new String(datos);
```

String: Crear Objetos

- También se pueden crear cadenas sin necesidad de usar los métodos constructores, inicializando directamente:

```
String cad = "Programando..."; // Es como new String("Hello world!");
String vacio = ""; // Es como new String();
String nulo ; //Se inicializa a null, no se crea cadena
```

- También se pueden crear cadenas de una forma implícita o usando el operador concatenación.

```
String cad3 = cad + vacio; //A partir de una expresión
System.out.println("Cadena por pantalla"); //De forma implícita
```

String: Métodos

- La clase String dispone de métodos para examinar caracteres individuales de la secuencia, para buscar y comparar cadenas, para extraer subcadenas, para crear una copia de una cadena con todos los caracteres en mayúsculas o minúsculas, etc. Algunos de estos métodos son los que siguen:

  - Operador +
  - boolean equals(String)
  - boolean equalsIgnoreCase(String)
  - int compareTo(String)
  - int length()
  - char charAt(int pos)
  - char [] toCharArray()
  - String substring(int inicio, int fin)
  - String substring(int inicio)
  - String concat(String)
  - String replace(char caracter1,char caracter2)
  - String toLowerCase()
  - String toUpperCase()
  - boolean startsWith(String)
  - boolean endsWith(String)
  - int indexOf(char)
  - int indexOf(char, int pos)
  - int indexOf(String)
  - int indexOf(String, int pos)

String: Métodos

- Operador +: Concatenación.
- boolean equals(String): Compara el String que invoca con el que pasa como parámetro, devolviendo true si su contenido es igual y false si no lo es. Tiene en cuenta las mayúsculas y minúsculas.
- boolean equalsIgnoreCase(String): Funciona como equals, pero no distingue entre mayúsculas y minúsculas.
- int compareTo(String): Compara el String que invoca con el que pasa como parámetro. Devuelve un entero menor, igual o mayor que cero, según sea su contenido menor, igual o mayor que el que se pasa como parámetro, respectivamente.
- int length(): Devuelve el número de caracteres de la cadena que invoca al método.
- char charAt(int pos): Devuelve el carácter que ocupa la posición pos del objeto que invoca. Comienza en la posición 0.
- char [ ] toCharArray(): Devuelve una referencia a un Array de caracteres (char[]) que contiene los caracteres del String que invoca al método. Es decir, convierte un String en un array de caracteres.

String: Métodos

- String substring(int inicio, int fin): Devuelve la subcadena que está entre las posiciones inicio (incluido) y fin (no incluido) de la cadena que invoca al método. El primer carácter está en la posición 0.
- String substring(int inicio): Devuelve la subcadena que está a partir de inicio (incluido) del String que invoca al método.
- String concat(string): Devuelve una cadena con la concatenación del String que invoca al método y el que pasa como parámetro .
- String replace (char caracter1,char caracter2): Devuelve una nueva cadena donde se han reemplazado todas las ocurrencias del carácter1 por el carácter2 en el String que invoca al método. Si no existiese el carácter1, devuelve el objeto que invoca (no crea ninguno).

String: Métodos

- String toLowerCase(): Devuelve un nuevo String a partir del que invoca donde se han cambiado las letras mayúsculas por minúsculas. Si no hay cambio se devuelve el String que invocó. (no se crea ninguno).
- String toUpperCase(): Igual que el anterior pero cambiando las minúsculas a mayúsculas.
- boolean startsWith(String): Devuelve un booleano indicando si la cadena que llama al método empieza por la subcadena que se da como parámetro.
- boolean endsWith(String): Devuelve un booleano indicando si la cadena que llama al método termina con la subcadena que se da como parámetro.
- int indexOf(char), int indexOf(char, int pos), int indexOf(String), int indexOf(String, int pos): Devuelve la posición de la primera aparición del carácter o cadena que se le pasa como parámetro comenzando la búsqueda a partir de la posición pos si se le pasa por parámetro.

String: Comparación

- Dado que las cadenas de caracteres son objetos, hay que tener en cuenta los conceptos de igualdad e identidad vistos anteriormente.
- Comparación de Igualdad → equals / equalsIgnoreCase

```
if(cad1.equals(cad2))
   System.out.println("cad1 y cad2 son iguales");
```

- Comparación de Identidad → operador =

```
if(cad1==cad2)
   System.out.println("cad1 y cad2 son la misma cadena");
```

- Comparación Relacional Alfabética → compareTo

```
if(cad1.compareTo(cad2) > 0 )
   System.out.println("cad1 > cad2");
else if (cad1.compareTo(cad2) < 0)
   System.out.println("cad1 < cad2");
else
   System.out.println("cad1 igual que cad2");
```

String: Comparación

- Comparación Relacional sin distinción entre mayúsculas y minúsculas → compareTo + toUpperCase/toLowerCase

```
if(cad1.toLowerCase().compareTo(cad2.toLowerCase()) > 0 )
   System.out.println("cad1 > cad2");
else if (cad1.toLowerCase().compareTo(cad2.toLowerCase())< 0)
   System.out.println("cad1 < cad2");
else
   System.out.println("cad1 igual que cad2");
```

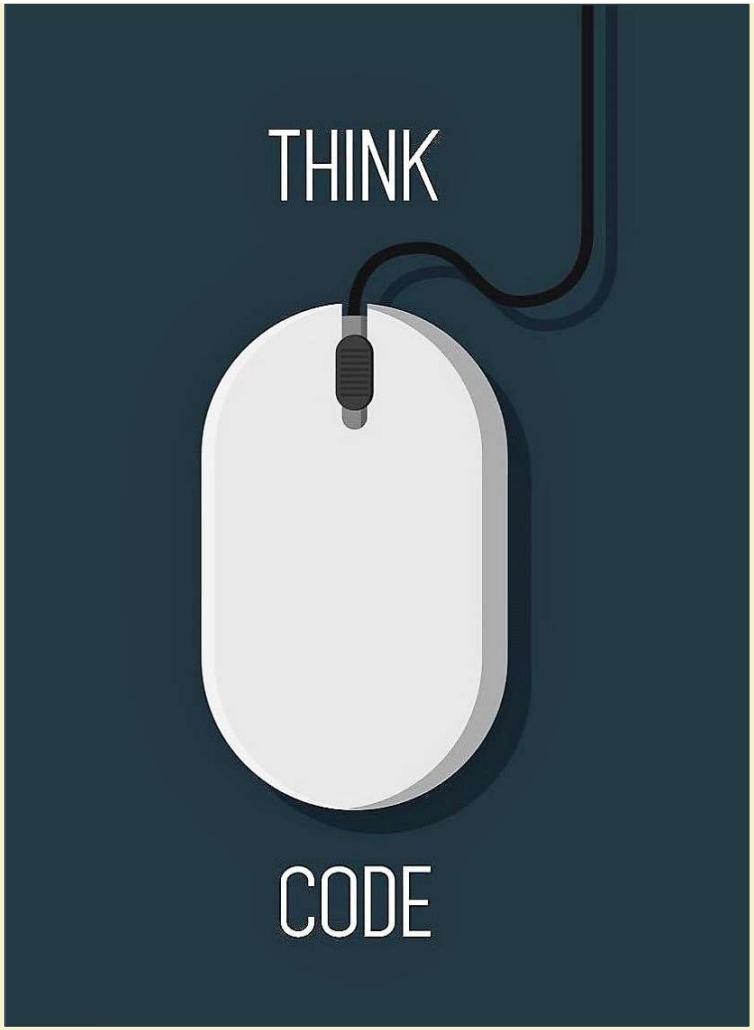
- Comparación Relacional por tamaño → length

```
if(cad1.length() > cad2.length())
   System.out.println("cad1 > cad2");
else if (cad1.length() < cad2.length())
   System.out.println("cad1 < cad2");
else
   System.out.println("cad1 tiene el mismo tamaño que cad2");
```

¿Preguntas?

08

Unidad Didáctica 8: Trabajando con comparadores Las interfaces Comparable y Comparator, y la clase Arrays



Módulo: Programación

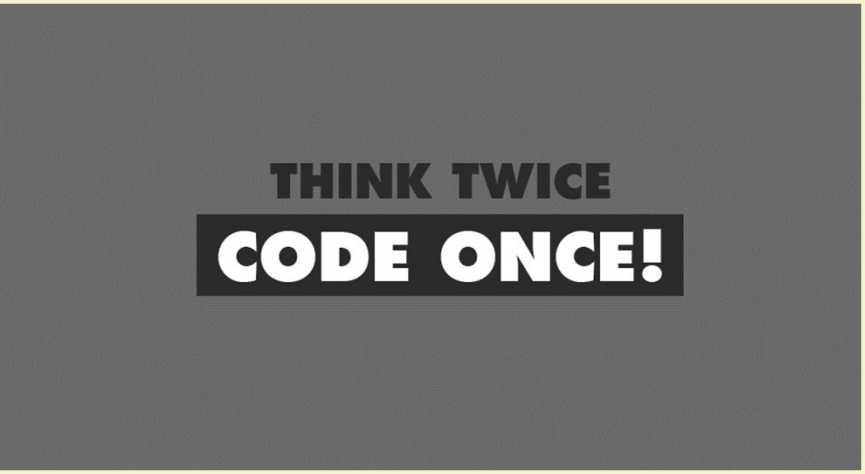
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

Índice



Interfaces Comparable y Comparator

- En ocasiones, cuando se manejan colecciones de datos (en nuestro caso vectores o arrays), es necesario ordenar dichos vectores.
- Si el array contiene elementos de tipo primitivo u objetos de clases estándar (como String), el orden intrínseco está claro. Pero cuando los objetos almacenados en el vector no son de una clase estándar (por ejemplo Bombilla), es necesario establecer un criterio de ordenación, es decir, cuándo un objeto es menor, mayor o igual a otro de su misma clase.
- Para ello se dispone de dos interfaces:
  - java.lang.Comparable : Para establecer el orden natural de una clase
  - java.util.Comparator : Para establecer criterios de ordenación alternativos
- Pertenecen a paquetes diferentes.

Interfaz Comparable

- Se dice que las clases que implementan esta interfaz cuentan con un "orden natural".
- Este orden es total, es decir, que siempre han de poder ordenarse dos objetos cualesquiera del la clase que implementa este interfaz.
- La interfaz Comparable declara el método compareTo():

```
public int compareTo(Object obj);
```

- que compara el objeto implícito (this) con el que se le pasa como parámetro.
- Este método debe devolver un entero negativo, cero o positivo según el objeto implícito (this) sea anterior, igual o posterior al objeto obj, respectivamente.

Interfaz Comparable

- Así, si queremos dotar a una clase de orden natural:
- La clase implementará la interfaz Comparable
- La clase redefinirá el método compareTo para establecer el orden natural de los objetos de la misma.
- El método compareTo() debe ser programado con cuidado, ya que:
  - Es muy conveniente que sea coherente con el método equals()
  - Debe cumplir la propiedad transitiva.(Si X<Y y Y<Z → X<Z)
- Una vez declarada la clase como Comparable y redefinido el método compareTo, podremos ordenar cualquier vector v de objetos de dicha clase usando el método Arrays.sort(v).

Interfaz Comparable - Ejemplo

- Dotar a los objetos de la clase Bombilla de un orden natural alfabético por marca y nivel de potencia, respectivamente. (nótese que es coherente con el equals).

```
public class Bombilla implements Cloneable, Comparable{
...
public int compareTo(Object o) {
   Bombilla b = (Bombilla) o;
   int cmp = this.marca.compareTo(b.marca);
   if(cmp==0){
      cmp = this.potencia-b.potencia;
      if(cmp ==0){
         if(this.encendida == b.encendida){
            cmp =0;
         }else{
            cmp =-1;
         }
      }else{
         cmp = 1;
      }
   }
   return cmp;
}
```

Interfaz Comparable: Ordenación de un array

- El interfaz Empleado es implementado por la clase Persona, que a su vez implementa el interfaz Comparable (redefiniendo compareTo). Así, compareTo() define la ordenación natural de las personas, siendo ésta por apellidos, nombre y edad.

```
//Empleado.java

public interface Empleado{
 public String getNombre();
 public String getApellidos();
 public int getEdad();
 }
```

- No se añade compareTo, ya que está declarado en la interfaz Comparable

Interfaz Comparable: Ordenación de un array

```
//Persona.java
public class Persona implements Comparable, Empleado {
   private String nombre, apellidos;
   private int edad;
   public Persona(String n, String a, int e) {nombre = n; apellidos = a; edad = e;}
   public String getNombre() {return nombre;}
   public String getApellidos() {return apellidos;}
   public int getEdad() {  return edad; }
   public boolean equals(Object o) {
      if (!(o instanceof Persona))
         return false;
      else {
         Persona p = (Persona) o;
         return p.nombre.equals(nombre) && p.apellidos.equals(apellidos) &&
                              p.edad == edad;//son enteros int
      }
   }
```

Interfaz Comparable: Ordenación de un array

```
public int compareTo(Object o) {
   Persona b = (Persona) o;
   int cmp = this.nombre.compareTo(b.nombre);
   if(cmp==0){
         cmp = this.apellidos .compareTo(b.apellidos);
          if(cmp ==0){
             if(this.edad>b.edad)
                cmp=1;
             else if(this.edad<b.edad)
                cmp=-1;
             else
                cmp=0;
          }
   }
   return cmp;

}
```

```
public String toString(){
   return this.getNombre()+ " "+ this.getApellidos() + " ("+
this.getEdad() + ")" + "\n";
}
```

Interfaz Comparable: Ordenación de un array

```
public class TestEmpleado {
   public static void main(String[] args) {
      Empleado [] t = new Empleado[5];
      t[0] = new Persona("Olga", "Moreno Perez", 25);
      t[1] = new Persona("Lola", "Lopez Jimenez", 23);
      t[2] = new Persona(”Antonio", "Lopez Perez", 25);
      t[3] = new Persona("Antonio", ”lopez Perez", 25);
      t[4] = new Persona("Daniela", "Sanchez Olmo", 21);

      //ComparadosEmpleados.MUY IMPORTANTE EL CASTING
      if (((Persona) t[0]).compareTo((t[1])) > 0)
         System.out.println(t[0] + " es mayor que " + t[1]);
      System.out.println("Lista Original:");//Imprime la lista original
      for (int i = 0; i < t.length; i++)
         System.out.println("\t" + t[i]);

//Ordena e imprime la lista según la ordenación natural de Persona
      System.out.println("Ordenación:");
      Arrays.sort(t);//Ordena
      for (int i = 0; i < t.length; i++)
         System.out.println("\t" + t[i]);
   }

}
```

Interfaz Comparator

- Si una clase ya tiene un criterio de ordenación natural (interfaz Comparable) y se desea tener un criterio de ordenación diferente, por ejemplo descendente o dependiente de otros campos, es necesario crear una clase que implemente dicho criterio.
- Esta clase, que se denomina comprador, es independiente de la clase objeto de la ordenación y deberá implementar la interfaz Comparator del paquete java.util. Un comparador es por tanto una clase que define un criterio de ordenación de otras clases.
- La interfaz Comparator declara el método compare en la forma:

```
public int compare(Object obj1, Object obj2);
```

- Así, la clase comparadora deberá implementar la interfaz Comparator, y por tanto el método compare, de la siguiente forma:

```
public class MiComparador implements Comparator{
   public int compare(Object o1, Object o2){
      <implementación del criterio de comparación>
   }
}
```

Interfaz Comparator

- El método compare() devuelve un entero negativo, cero o positivo según su primer argumento sea anterior, igual o posterior al segundo (así asegura un orden ascendente). La implementación debe asegurar que:
- signo(compare(x,y) debe ser igual - signo(compare(y,x)) para todas las x, y. Esto implica que compare(x, y) lanzará una excepción solo si compare(y, x) la lanza.
- La relación es transitiva:

  compare(x,y) > 0 && compare(y,z) > 0 implican compare(x z) > 0.

- compare(x,y)==0 implica signo(compare(x,z)== sgn(compare(y,z)) ∀z.
- Este método lanzará la excepción ClassCastException si el tipo de los argumentos impide la comparación por este Comparator.
- Es importante que compare() sea compatible con el método equals() de los objetos que hay que mantener ordenados. Su implementación debe cumplir unas condiciones similares a las de compareTo().

Interfaz Comparator - Ejemplo

- Usando las la interfaz Empleado y la clase Persona del ejemplo anterior, se añaden dos comparadores para hacer diferentes ordenaciones. En concreto:
- ComparadorIgnoraMaxyMin, cuyo método compare() compara dos Personas con el mismo criterio que compareTo() del ejemplo anterior, sólo que en este caso no se distinguen mayúsculas y minúsculas
- ComparadorEdad, que usa la edad como criterio de comparación entre personas (si son de la misma edad, compara por apellidos y nombre).

Interfaz Comparator - Ejemplo

```
import java.util.Comparator;

public class ComparadorIgnoraMaxyMin implements Comparator {

   public int compare(Object o1, Object o2) {
      Persona p1 = (Persona) o1;
      Persona p2 = (Persona) o2;

      int cmp = p1.getApellidos().compareToIgnoreCase(p2.getApellidos());
      if(cmp==0){
         cmp = p1.getNombre().compareToIgnoreCase(p2.getNombre());
         if(cmp==0){
            cmp= p1.getEdad()-p2.getEdad();
         }
      }
     return cmp;
   }
}
```

Interfaz Comparator - Ejemplo

```
import java.util.Comparator;

public class ComparadorEdad implements Comparator {

   public int compare(Object o1, Object o2) {

      Persona p1 = (Persona) o1;
      Persona p2 = (Persona) o2;
      int cmp = p1.getEdad()-p2.getEdad();
      if(cmp==0){
         cmp = p1.getApellidos().compareToIgnoreCase(p2.getApellidos());
         if(cmp ==0){
            cmp = p1.getNombre().compareToIgnoreCase(p2.getNombre());
         }
      }
      return cmp;
   }
}
```

Interfaz Comparator - Ejemplo

```
//TestEmpleado.java
//…
System.out.println("Ordenando con ComparadorIgnoraMaxyMin: ");
Arrays.sort(t, new ComparadorIgnoraMaxyMin());
for (int i = 0; i < t.length; i++)
   System.out.println("\t" + t[i]);



System.out.println("Ordenando con ComparadorEdad: ");
Arrays.sort(t, new ComparadorEdad());
for (int i = 0; i < t.length; i++)
   System.out.println("\t" + t[i]);
```

Clase Arrays

- La clase Arrays, que pertenece al paquete java.util, contiene una serie de métodos estáticos que permiten el manejo de los objetos de tipo tabla (vectores).
- Así, para llamar a cualquiera de estos métodos basta con usar el nombre de la clase Arrays seguido de un punto y el nombre del método que se quiere invocar junto a los parámetros.

```
Arrays.sort(v); //Ordenar el vector v

pos = Arrays.binarySearch(v,obj); //busca el objeto obj en v

java.util.Arrays.equals(v,w); // v y w iguales?
```

Clase Arrays

- void sort(<vector>): Permite la ordenación de un array unidimensional. Dependiendo del tipo de datos que almacena los criterios serán:
- Si el array es de tipos primitivos los criterios son los habituales de comparación de estos tipos.
- Si el array contiene referencias a objetos de otras clases es necesario que implementen la interfaz Comparable y redefinir el método compareTo.(LO HEMOS HECHO EN EL EJEMPLO CON COMPARABLE).

```
Bombilla[] alumbrado = new Bombilla[3];
alumbrado[0] = new Bombilla("Siemens");
alumbrado[1] = new Bombilla("Fujitsu");
alumbrado[2] = new Bombilla("Philips");
Arrays.sort(alumbrado);
```

Clase Arrays

- int binarySearch(<vector>,<valor>): Busca en un array unidimensional ordenado un determinado elemento (valor u objeto).
- Si encuentra el objeto con ese valor devuelve la posición que ocupa dentro del vector.
- Si no lo encuentra devuelve la posición en la que debería haber estado, precedida de un signo menos (-).
- Como ocurre con el método sort(), los objetos contenidos en la tabla deben tener redefinido el método compareTo().

```
Bombilla b1 = new Bombilla("Siemens");
Bombilla b2 = new Bombilla("Bosh");
b1.encender();
int i,j;
i= Arrays.binarySearch(alumbrado,b1);
j= Arrays.binarySearch(alumbrado,b2);
```

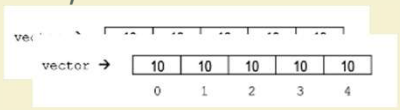
Clase Arrays

- boolean equals(<vector>,<vector>): Comprueba si dos arrays son iguales.
- Devuelve true si los dos vectores tienen la misma longitud, los mismos elementos y en el mismo orden.
- Para que se compare por igualdad, los objetos contenidos en el vector deben tener redefinido el método equals().

```
Bombilla[] alumbrado2 = new Bombilla[3];
alumbrado[0] = new Bombilla("Siemens");
alumbrado[1] = new Bombilla("Fujitsu");
alumbrado[2] = new Bombilla("Philips");

boolean iguales;
iguales = Arrays.equals(alumbrado, alumbrado2); //FALSE
Arrays.sort(alumbrado2);
iguales = Arrays.equals(alumbrado, alumbrado2); //TRUE
```

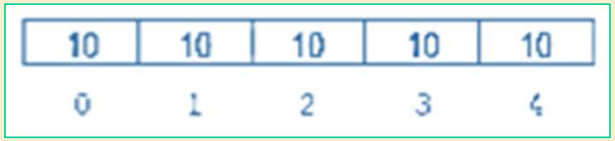
Clase Arrays

- void fill(<vector> , <valor>): Rellena todos los elementos de un vector con el valor que se pasa como argumento.
- Si el valor es una referencia a un objeto, todos los elementos del vector referenciarán a dicho objeto.



- Ejemplos:

```
Bombilla [] alumbrado3= new Bombilla[4];
Bombilla b3 = new Bombilla("Sanyo");
Arrays.fill(alumbrado3,b3);

int[] vector = new int [5];
Arrays.fill(vector,10);
```



Clase Arrays

- String toString(<vector>): Devuelve la representación en forma de cadena de caracteres del vector que se le pasa como parámetro.
- Si el vector contiene objetos, la clase correspondiente deberá tener redefinido el método toString.

```
String[] nombres = {"Lola", "Pepe", "Ana", "Luis"};
float[] notas = {5.6F, 4.2F, 8.7F, 9.8F};
System.out.println(java.util.Arrays.toString(nombres));
System.out.println(java.util.Arrays.toString(notas));
```

Clase Arrays

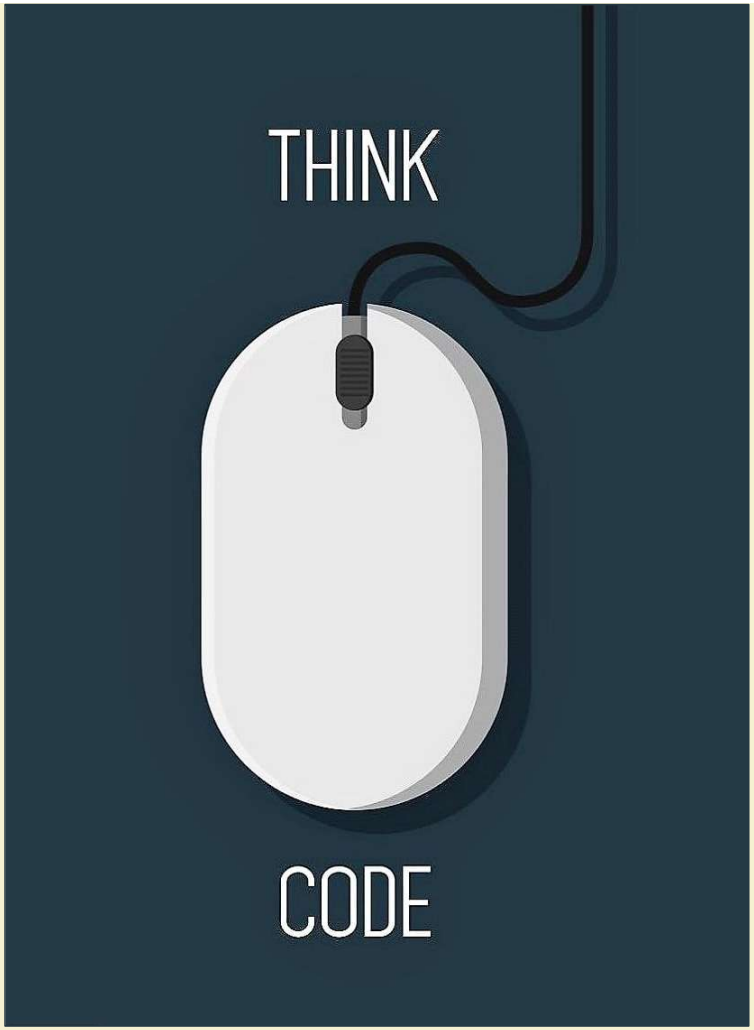
- Sobrecargas:

```
public static void sort(tipoPrimitivo[]);
public static void sort(tipoPrimitivo[],Comparator);
public static void sort(Object[]);
public static void sort(Object[],Comparator);

public static int binarySearch(tipoPrimitivo[],unPrimitivo);
public static int binarySearch(Object[],Object);
public static int binarySearch(Object[],Object, Comparator);

public static void fill(tipoPrimitivo[],unPrimitivo);
public static void fill(tipoprimitivo[],int fromIndex, int toIndex, unPrimitivo);
public static void fill(Object[],Object);
public static void fill(Object[],int fromIndex, int toIndex, Object);
```

¿Preguntas?

09

Unidad Didáctica 9: Excepciones: Gestionando errores



Módulo: Programación

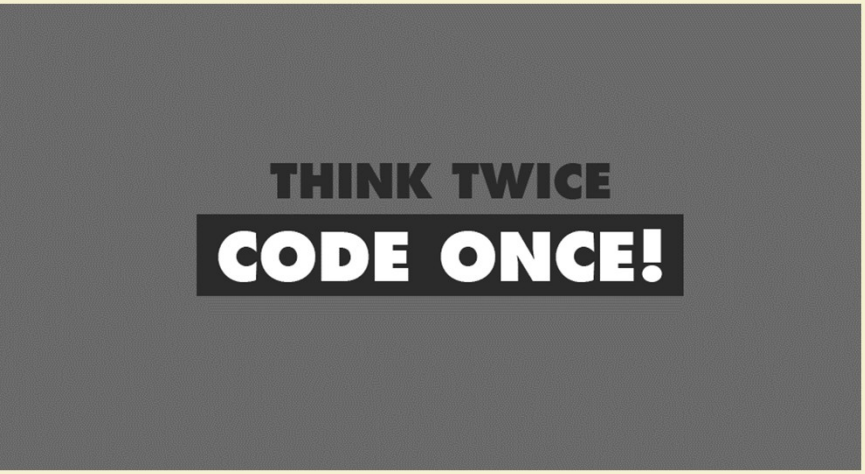
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

Índice



Excepciones: ¿Qué son?

- Una excepción es la representación orientada a objetos de los errores.
- Una excepción es un error provocado por alguna condición inesperada.
- Cuando se produce un error durante la ejecución de un programa, se crea un objeto para representar el error que se acaba de producirse. Este objeto contiene numerosa información relativa al error ocurrido en la aplicación, así como el estado de la aplicación en el momento de la aparición del error. A continuación, se transmite este objeto a la máquina virtual. Esto activa la excepción.
- Entonces, la máquina virtual debe buscar una solución para resolverla. Para ello, explora los diferentes métodos llamados para alcanzar la ubicación donde se produjo el error. En estos distintos métodos, la máquina busca un gestor de excepciones capaz de tratar el problema. La búsqueda empieza con el método en el cual se activó el error y, a continuación, sube hasta el método main de la aplicación, si es necesario. Cuando se localiza un gestor de excepciones adecuado, se le transmite el objeto para que se encargue de su tratamiento. Si la búsqueda no da resultado, la aplicación se detiene.

Excepciones: La clase Exception

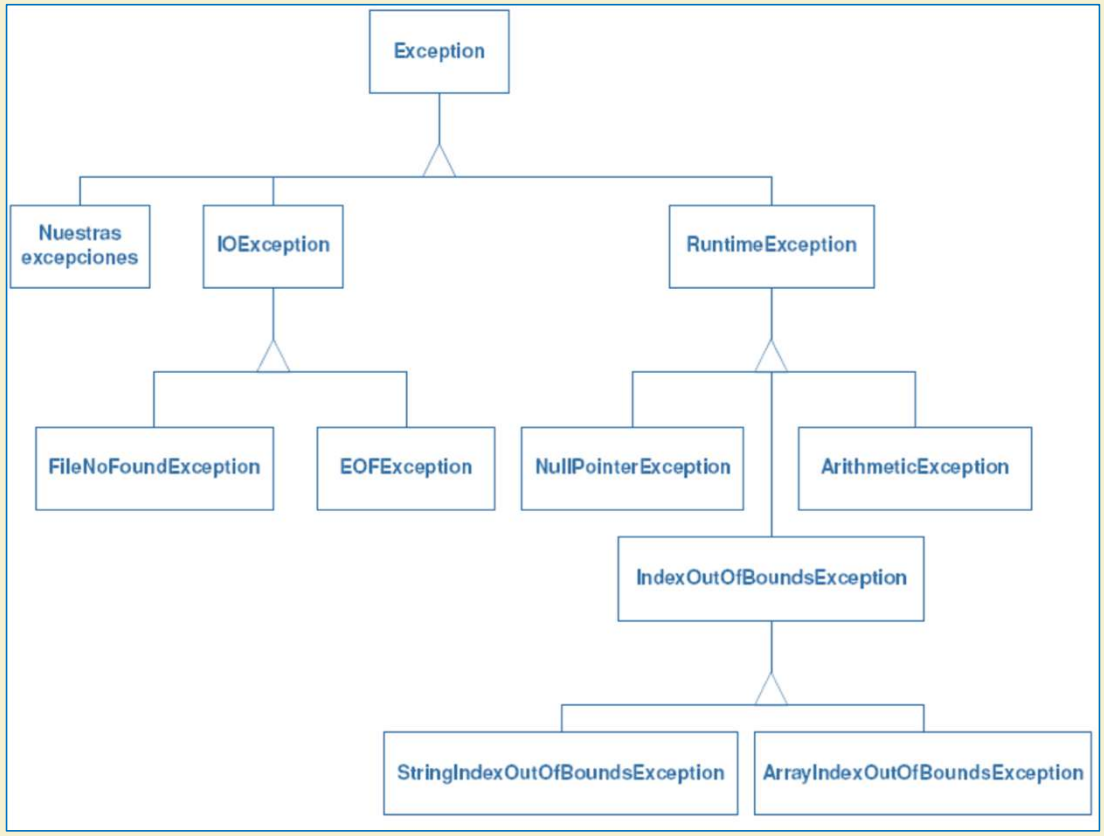
- La clase Exception tiene definidos, entre otros, los siguientes métodos que serán de utilidad para la señalización y tratamiento de excepciones:
- Exception(String m): Aunque la clase tiene un constructor por defecto, este constructor es más interesante, ya que nos permite pasar una cadena de caracteres que se usará como mensaje indicativo de la excepción.
- String getMessage( ): Este método devuelve el mensaje que describe la excepción.
- printStackTrace( ): Este método imprime por pantalla el estado de la pila de ejecución en el momento que se produjo la excepción.

Excepciones: Las subclases de Exception

- Las subclases de Exception, por convenio, suelen tener un nombre que indica su tipo y suelen finalizar con el sufijo Exception.
- En la librería estándar de Java hay definidas multitud de subclases en función del tipo de excepción. Algunos ejemplos de las mismas son los siguientes:
- ClassNotFoundException: Indica que se ha intentado cargar una clase (p.ej. al crear un objeto) y no puede encontrar la definición de la misma.
- IOException: Indican de forma genérica un error en el sistema de Entrada/Salida.
- RunTimeException: Indica de forma genérica algún tipo de excepción en tiempo de ejecución. Algunas de sus subclases:
- AritmeticException: Excepción producida por una operación aritmética errónea (p. ej. Dividir algo entre cero).
- NullPointerException: Excepción producida por hacer uso de una referencia que no apunta a un objeto.

Excepciones: Clasificación

- Las excepciones se suelen clasificar en categorías representadas en el diagrama de clases siguiente:



Excepciones: Clasificación

- Las excepciones en Java se modelan como objetos de alguna subclase de la clase Exception.
- La clase IOException son aquellas excepciones relacionadas con la entrada/ salida de Java.
- La clase RuntimeException son aquellas excepciones que Java lanza automáticamente en tiempo de ejecución y el programador no está obligado a capturarlas, por lo que son un caso especial de excepciones.
- Como ocurre con las clases ordinarias de Java en las que se da polimorfismo, una excepción de una determinada clase también puede ser considerada como una excepción de la clase padre de la clase a la que pertenece dicha excepción. Así, una excepción de la clase AritmeticException se considera también que es de la clase RunTimeException y de la clase Exception.

Excepciones: Tratamiento

- Las excepciones en Java no pueden ser ignoradas. Java obliga a decidir qué hacer con ellas, y el hecho de ignorarlas provoca un error de compilación.
- Ante una excepción se pueden tomar dos acciones:
  - Propagarla: Cuando en un método no se desea o no se puede hacer nada con una excepción, éste puede propagar la excepción para que sea tratada por el método que lo llamó el que la trate. Esta propagación puede ir dándose hasta que la excepción llegue al método main, en cuyo caso se aborta la ejecución.
  - Capturarla: Capturar una excepción consiste en detectarla y ejecutar cierto código (try-catch) para reaccionar ante ella. Veremos en siguientes apartados más detalles sobre este tema.

Excepciones: Propagación

- Para indicar que un método propagará una excepción se añadirá a final de la cabecera de dicho método la palabra clave thows y seguidamente se indicará las clases de excepciones que se propagan, separándolas por comas.
- Por ejemplo, el siguiente método propaga cualquier excepción (ya que todas pertenecen a clases hijas de la clase Exception):

```
public void metodo(int argumento)throws Exception{
    . . .
}
```

```
public void cambiaEdad(int e) throws EdadErroreaException{
    if(e >=0 || e <130)
        this.edad = e;
    else
            throw new EdadErroneaException("edad incorrecta");
}
```

Excepciones: Propagación

- Las excepciones que heredan de la clase RunTimeException son una excepción a la obligación de tratamiento de excepciones.
- No será obligatorio tratar estas, y por defecto todos los métodos las propagan.
- Esta peculiaridad de Java se produce ya que este tipo de excepciones son las excepciones más comunes en los programas y la obligación de tener tratarlas complicaría mucho el código fuente.

```
public double divide(int x, int y){
   return x/y;
}
```

Puede provocar una ArithmeticException si y=0, pero no es necesario propagarla.

Excepciones: Captura

- Cuando llamamos a un método y éste, debido a alguna situación anómala lanza una excepción, deberíamos de detectar y tratar dicha excepción. A este hecho se le llama comúnmente capturar la excepción.
- Para ello, Java nos ofrece la estructura try-catch (intentar-atrapar):
- La parte try contendrá aquellas sentencias que posiblemente puedan provocar una excepción y las sentencias que dependan de las anteriores.
- En la parte catch se especificarán las sentencias que permiten tratar la excepción si ésta se ha producido.
- Este bloque tiene la forma:

```
try{
   //Código del bloque try
}catch(tipoExcepcion){
   //Código del bloque catch
}
```

Excepciones: Captura

- Ejemplo: El siguiente código muestra un método de ejemplo que invoca al método cero (que lanza una excepción) anteriormente descrito y trata una posible excepción que pudiese ser lanzada:

```
public void intenta(int numero){
   int resultado;
   System.out.println("Vamos a intentar la ejecución del método 'cero'");
   try{
      resultado = cero(numero);
      System.out.println("El resultado obtenido es: " + resultado);
   } catch(Exception ex){
      System.out.println("Se ha producido un error: " + ex.getMessage());
   }
   System.out.println("Fin del intento");
}
```

Excepciones: Captura

- La estructura try-catch puede tener tantos bloques catch cómo sea necesario, siempre y cuando el tipo de excepción que se indique en la signatura sea distinto. Cuando se produce una excepción Java irá analizando por orden los bloques catch:

```
public void intenta(int numero){
   int resultado;
   System.out.println("Vamos a intentar la ejecución del método 'cero'");
   try{
      resultado = cero(numero);
      System.out.println("El resultado obtenido es: " + resultado);
   } catch(ArithmeticException ex){
      System.out.println("Excepción aritmética: " + ex.getMessage());
   }catch(Exception ex) {
      System.out.println("Se ha producido un error: " + ex.getMessage());
   }
   System.out.println("Fin del intento");
}
```

Excepciones: Lanzamiento

- Se dice que un método lanza o dispara una excepción cuando éste la eleva explícitamente.
- La palabra reservada throw (lanzar) lanza una excepción a la que sigue un objeto que indica la excepción a lanzar.
- Todo método que lance excepciones a quien los invoque debe indicar en su cabecera que las propaga, ya que si no estaría obligado a tratarlas capturándolas.
- Un ejemplo trivial de un método que lanza una excepción es el siguiente:

```
public int cero(int numero) throws Exception{
   if(numero==0)
      return numero;
   else
      throw new Exception("El parámetro no es cero");
}
```

→ Hay que propagar la excepción

→ Se lanza una excepción genérica, con un mensaje describiendo el error.

Excepciones: Crear excepciones

- El tipo de una excepción viene indicado por la clase a la que pertenece el objeto que la representa.
- Para definir tantos tipos de excepciones como se desee, deberemos crear sus propias clases de excepción.
- Estas clases heredarán de la clase Exception, o de alguna de sus clases hijas, y podrán estar organizadas de forma jerárquica.

```
public class MiExcepcion extends Exception{

   public MiExcepcion(String msj){
      super(msj);
   }
}
```

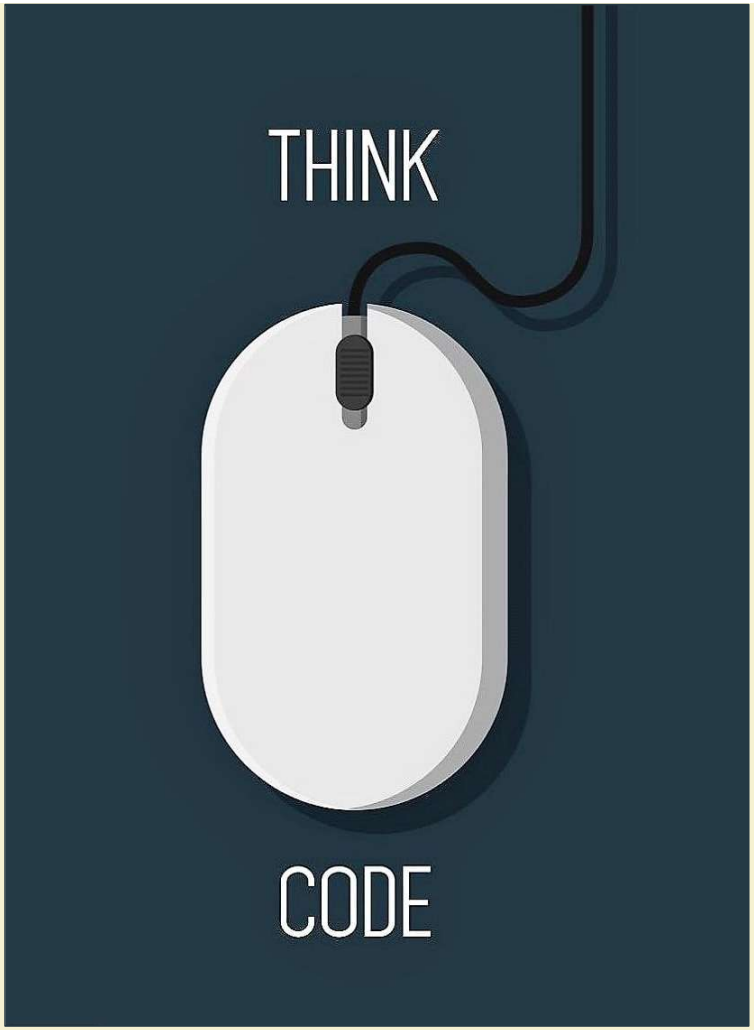
- Para lanzar una excepción del tipo que hemos creado sólo será necesario el siguiente código:

```
throw new MiExcepcion("Mensajito de error");
```

¿Preguntas?

010

Unidad Didáctica 10: Las clases Scanner y Collections



Módulo: Programación

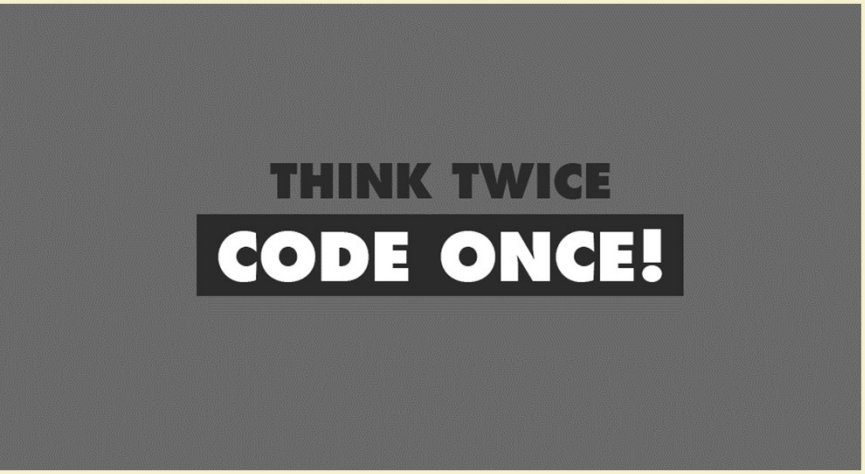
Licencia

Toda la documentación de esta asignatura queda recogida bajo la licencia de Creative Commons

https://creativecommons.org/licenses/by-nc-nd/4.0/

En el caso de incumplimiento o infracción de una licencia Creative Commons, el autor, como con cualquier otra obra y licencia, habrá de recurrir a los tribunales. Cuando se trate de una infracción directa (por un usuario de la licencia Creative Commons), el autor le podrá demandar tanto por infracción de la propiedad intelectual como por incumplimiento contractual (ya que la licencia crea un vínculo directo entre autor y usuario/licenciatario). El derecho moral de integridad recogido por la legislación española queda protegido aunque no aparezca en las licencias Creative Commons. Estas licencias no sustituyen ni reducen los derechos que la ley confiere al autor; por tanto, el autor podría demandar a un usuario que, con cualquier licencia Creative Commons, hubiera modificado o mutilado su obra causando un perjuicio a su reputación o sus intereses. Por descontado, la decisión de cuándo ha habido mutilación y de cuándo la mutilación perjudica la reputación o los intereses del autor quedaría en manos de cCutacia Juez o Tribunal.

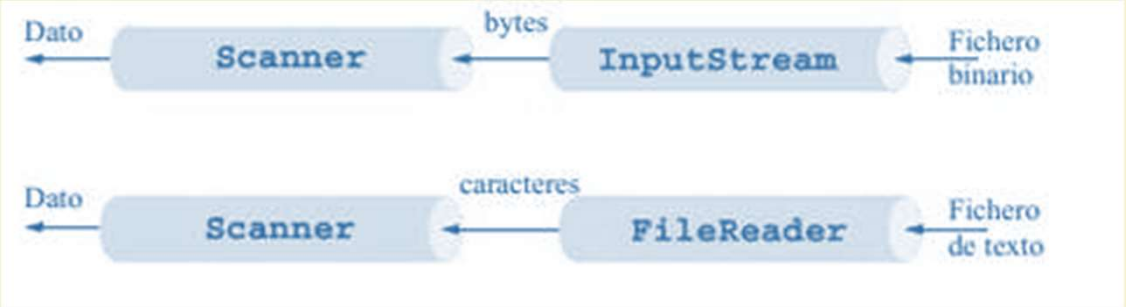
Índice



</> La clase Scanner

La clase Scanner

- La lectura de datos por teclado desde una aplicación Java resulta bastante engorrosa.
- Esta clase (java.util.Scanner) proporciona una serie de métodos para realizar la lectura de datos desde un dispositivo de entrada o fichero, tanto en forma de cadena de caracteres como en cualquier tipo básico.

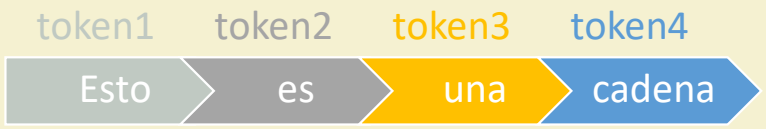


La Clase Scanner: Crear un objeto

- Para tener acceso a estos datos de entrada, primeramente necesitamos crear un objeto scanner asociado al InputStream del dispositivo de entrada. En el caso del teclado se utilizaría la instrucción:

```
Scanner se = new Scanner (System.in);
```

- La cadena de caracteres introducida por teclado hasta la pulsación de la tecla "enter" es dividida por el objeto scanner en un conjunto de bloques de caracteres de longitud variable, denominados tokens. De forma predeterminada, el carácter utilizado como separador de token es el espacio en blanco.



- Utilizando los métodos de la clase Scanner es posible recuperar secuencialmente cada uno de estos tokens, e incluso convertirlos implícitamente a un determinado tipo de datos.

La Clase Scanner: Métodos

- Entre los métodos más destacables de Scanner se encuentran:
- String next(). Devuelve el siguiente token.
- String nextLine(). Devuelve hasta el final de la línea.
- boolean hasNext(). Indica si existe o no un nuevo token para leer.
- Xxx nextXxx(). Devuelve el siguiente token como un tipo básico siendo Xxx el nombre de este tipo básico. Por ejemplo, *nextInt()* para lectura de un entero o *nextFloat()* para la lectura de un float.
- boolean hasNextXxx(). Indica si existe o no un siguiente token del tipo especificado, siendo Xxx el nombre del tipo.
- void useDelimiter(String d). Establece un nuevo delimitador de token.

La Clase Scanner: Ejemplo

- El siguiente programa solicita al usuario su nombre y número de personal, mostrando a continuación dichos datos en pantalla:

```
import java.util.Scanner;

public class MuestraDatos {
   public static void main(String[] args) {
      String nom;
      int cod;
      Scanner sc = new Scanner(System.in);
      System.out.println("Introduzca su nombre:");
      nom = sc.next(); //lee el nombre
      System.out.println("Introduzca su número de personal: ");
      cod = sc.nextInt(); //lee el número de personal
      // como int
      System. out . println (nom + ":" + cod);
   }
}
```

La Clase Scanner: Ejemplo

- Como hemos indicado antes, hay que tener en cuenta que el delimitador de token es el espacio en blanco. En el programa anterior esto significa que si el nombre introducido contiene un espacio, por ejemplo, "Luis Pérez", al ejecutar la instrucción: `nom = sc.next();`
- la cadena recuperada en la variable nom será "Luis", pero como la cadena aún no ha finalizado y hay más tokens para leer la siguiente llamada a next() (en el ejemplo nextInt() intentará devolver el siguiente token (en el ejemplo será la cadena "Pérez"), dado que el contenido de éste no puede ser convertido a int se producirá una excepción de tipo java.util.InputMismatchException.
- Para evitar este problema, deberá asignarse como delimitador de token el salto de línea, de esta forma la primera llamada a next() devolverá todo el texto introducido hasta la pulsación de "enter". El establecimiento de un nuevo delimitador se lleva a cabo mediante el método useDelimiter() comentado anteriormente. En este ejemplo habría que añadir, después de la creación del objeto scanner, la instrucción: `sc.useDelimiter ("\n");`

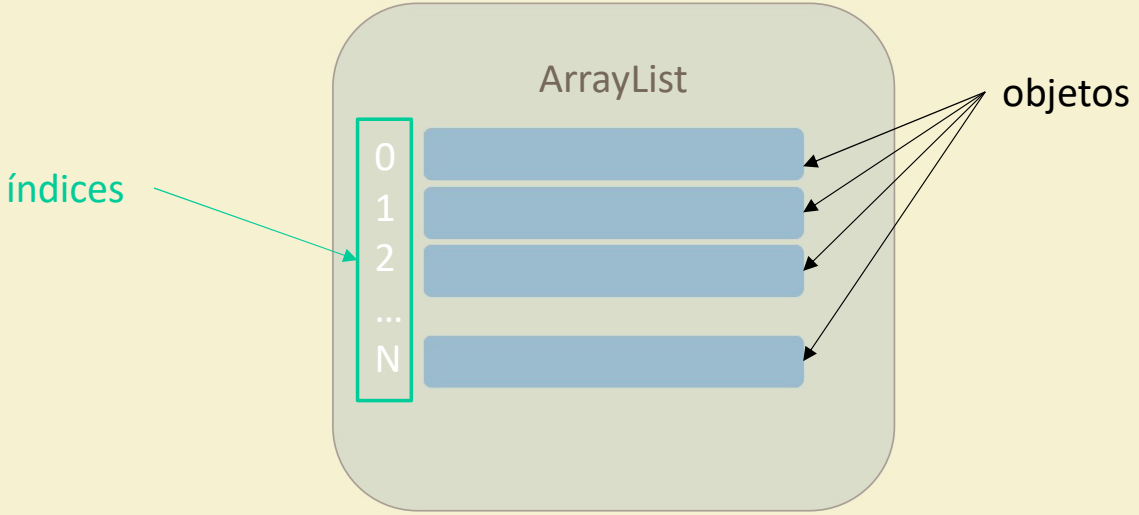
</> Collections

Colecciones o Collections

- Una colección es un objeto que almacena un conjunto de referencias a otros objetos, dicho de otra manera, es una especie de array de objetos.
- Sin embargo, a diferencia de los arrays, las colecciones son dinámicas, en el sentido de que no tienen un tamaño fijo y permiten añadir y eliminar objetos en tiempo de ejecución.
- Java incluye en el paquete java.util un amplio conjunto de clases para la creación y tratamiento de colecciones. Todas ellas proporcionan una serie de métodos para realizar las operaciones básicas sobre una colección, como son:
    - Añadir objetos a la colección.
    - Elimina objetos de la colección.
    - Obtener un objeto de la colección.
    - Localizar un objeto en la colección.
    - Iterar a través de una colección.
- A continuación, vamos a estudiar algunas de las clases de colección más significativas.

La clase ArrayList

- Un ArrayList representa una colección basada en índices, en la que cada objeto de la misma tiene asociado un número (índice) según la posición que ocupa dentro de la colección, siendo 0 la posición del primer elemento



ArrayList vs Array

- Los arrays son como la cuerda que se muestra en la imagen de arriba; Tendrán una longitud fija, no pueden expandirse ni reducirse desde la longitud original.
- Sin embargo, ArrayList puede crecer como, y cuando se requiera para acomodar los elementos que necesita almacenar y cuando se eliminan elementos, puede reducirse a un tamaño más pequeño.
- Podemos decir que ArrayList es como un array dinámico o una matriz de longitud variable.

La clase ArrayList

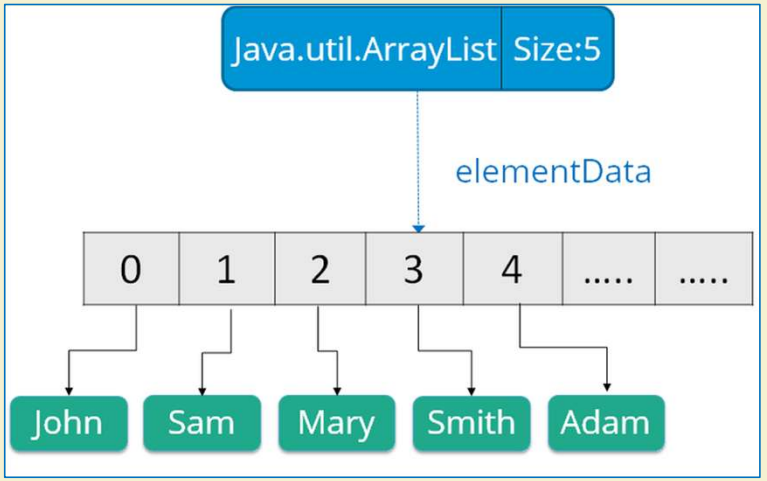
- Para crear un objeto ArrayList utilizamos la sintaxis:

```
ArrayList  variable_objeto = new ArrayList();
```

- Donde variable_objeto es la variable que contendrá la referencia al objeto ArrayList creado. Por ejemplo:

```
ArrayList  v = new ArrayList();
```

- Una vez creado, podemos hacer uso de los métodos de la clase ArrayList para realizar las operaciones habituales con una colección.



La clase ArrayList: Métodos

- Los principales métodos expuestos por esta clase son:
- boolean add(Object o). Añade un nuevo objeto y la colección (su referencia) y lo sitúa al final de la misma, devolviendo el valor true. Los objetos añadidos pueden ser de cualquier tipo no siendo necesario que todos pertenezcan a la misma clase:

```
ArrayList v=new ArrayList();
v.add ("hola"); //añade la cadena en la primera posición del ArrayList
v.add(6); //añade el numero en la segunda posición - autoboxing
```

- En este ejemplo observamos cómo, debido a que el ArrayList no puede almacenar tipos básicos, gracias a la característica del autoboxing es posible añadir directamente el número sin envolverlo previamente en un objeto pues, como ya sabemos, esta operación se lleva a cabo implícitamente.
- void add(int índice, object o). Añade un objeto al ArrayList en la posición especificada por índice, desplazando hacia delante el resto de los elementos de la colección. El índice de la primera posición es 0.

La clase ArrayList: Métodos

- Object get(int indice). Devuelve el objeto que ocupa la posición indicada. Hay que tener en cuenta que el tipo de devolución es Object, por tanto, para guardar la referencia al objeto devuelto en una variable de su tipo será necesario realizar una conversión explícita:

```
ArrayList v = new ArrayList();
v.add("texto"); //Conversión explícita a String
String s = (String) v.get(0);
```

- En el caso de que se almacenen objetos numéricos, es necesario recordar que la llamada a get() devuelve el objeto de envoltorio, y no el número:

```
ArrayList v=new ArrayList();
v.add(6); //autoboxing
Integer i=(Integer)v.get(0);//unboxing
System.out.println(i.intValue());
```

La clase ArrayList: Métodos

- La realización de conversiones cuando se recupera una referencia almacenada en una colección suele generar cierta confusión entre los programadores Java juniors. Algunas veces se tiende a realizar operaciones como ésta:

```
ArrayList v=new ArrayList();
v.add("35");
Integer s=(Integer)v.get(0); //Intenta recuperarlo como objeto numérico
```

- El código anterior compilaría sin ningún problema, sin embargo, al ejecutar la última línea se produciría una excepción ClassCastException, dado que no se puede convertir explícitamente un objeto String (eso es lo que se ha almacenado en el objeto de colección) en un Integer. Únicamente se puede convertir explícitamente el objeto a su tipo original. En el caso de que tengamos que recuperar como número un dato numérico almacenado como de texto en la colección, deberíamos hacerlo del siguiente modo:

```
ArrayList v=new ArrayList();
v.add("35");
String s=(String)v.get (0);//Se recupera en su tipo original
int n=Integer.parseInt(s);//Se convierte la cadena a número entero
```

La clase ArrayList: Métodos

- Object remove(int índice). Elimina de la colección el objeto que ocupa la posición indicada, desplazando hacía atrás los elementos de las posiciones siguientes. Devuelve el objeto eliminado.
- voíd clear(). Elimina todos los elementos de la colección.
- int indexOf(Object o). Localiza en el ArrayList el objeto indicado como parámetro, devolviendo su posición. En caso de que el objeto no se encuentre en la colección, la llamada al método devolverá como resultado el valor -1.

La clase ArrayList: Métodos

- int size(). Devuelve el número de elementos almacenados en la colección. Utilizando este método conjuntamente con get(), se puede recorrer la colección completa. El siguiente método realiza el recorrido de un ArrayList de cadenas para mostrar su contenido en pantalla:

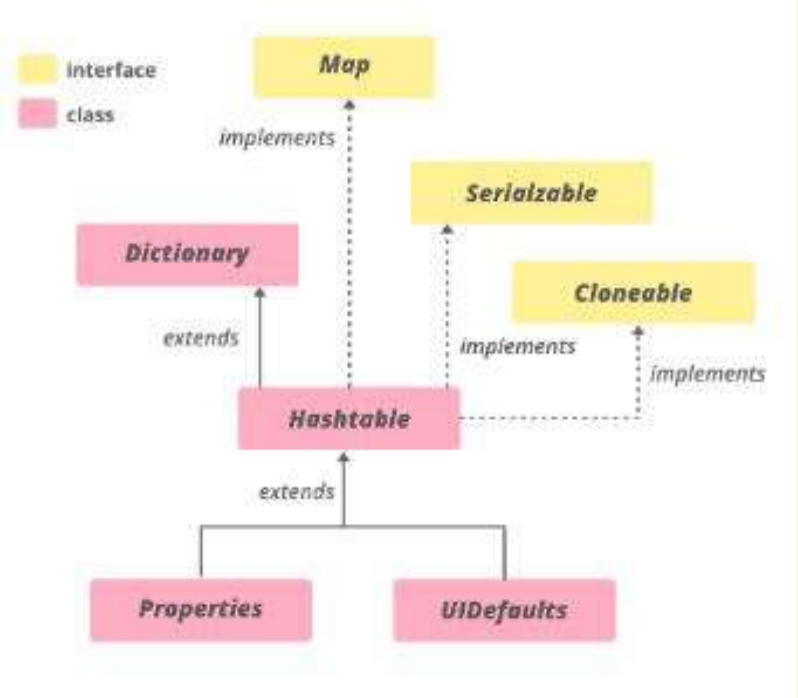
```
public void muestra (ArrayList v){
   for(int i=0;i<v.size();i++){
      System.out.println((String)v.get(i));
   }
}
```

- Como sabemos, se puede utilizar la variante for extendido para simplificar el recorrido de colecciones, así el método anterior quedaría:

```
public void muestra (ArrayList v){
   for(Object obj: v) {
      System.out.println((String)obj);
   }
}
```

La clase Hashtable

- La clase Hashtable representa un tipo de colección basada en claves, donde los objetas almacenados en la misma (valores) no tienen asociado un índice numérico basado en su posición, sino una clave que lo identifica de forma única dentro de la colección. Una clave puede ser cualquier tipo de objeto.
- La utilización de colecciones basadas en claves resulta útil en aquellas aplicaciones en las que se requiera realizar búsquedas de objetos a partir de un dato que lo identifica. Por ejemplo, si se va a gestionar una colección de objetos de tipo "Empleado", puede resultar más práctico almacenarlos en un Hashtable, asociándoles como clave el “dni", que guardarlos en un ArrayList en el que a cada empleado se le asigna un índice según el orden de almacenamiento.



La clase Hashtable: Creación de un objeto

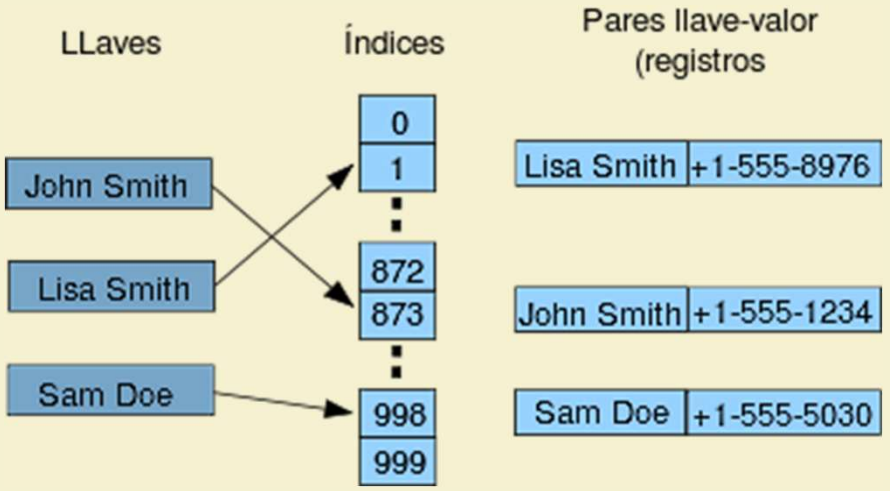
- La creación de un objeto Hashtable se realiza utilizando el constructor sin parámetros de la clase:

```
Hashtable varible_objeto = new Hashtable();
```

- Donde variable_objeto es la variable que contendrá la referencia al objeto Hashtable creado. Por ejemplo:

```
Hashtable ht = new Hashtable();
```

- Una vez creado, podemos hacer uso de los métodos de la clase Hashtable para realizar las operaciones habituales con una colección.



La clase Hashtable: Métodos

- Los principales métodos expuestos por la clase Hashtable para manipular la colección son los siguientes:
- Object put(Object key, Object valor). Añade a la colección el objeto valor, asignándole; la clave especificada por key. En caso de que exista esa clave en la colección, el objeto que tenía asignada esa clave se sustituye por el nuevo objeto valor, devolviendo objeto sustituido. Por ejemplo, el siguiente código:

```
Hashtable hs = new Hashtable();
hs.put ("a21", "pepito");
System.out.println ("Antes se llamaba "+ hs.put ("a21", "luis"));
```

- Mostrará en pantalla: Antes se llamaba pepito.
- Pero si volvemos a mostrar hs por pantalla veremos que tiene a21 con el valor luis

La clase Hashtable: Métodos

- Bolean containsKey(Object key). Indica sí la clave especificada existe o no en la colección.
- Object get(Object key) Devuelve el valor que tiene asociada la clave que se indica en el parámetro. En caso de que no exista ningún objeto con esa clave asociada, devolverá null.
- Object remove(Object key). Elimina de la colación el valor cuya clave se especifica en el parámetro. En caso de que no exista ningún objeto con esa clave, no hará nada y devolverá null si existe, eliminará el objeto y el mismo será devuelto por el método.
- int size(). Devuelve el número de objetos almacenados en la colección.
- Enumeration Keys(). Devuelve un objeto enumeration que permite iterar sobre el conjunto de claves. En el siguiente apartado se estudiará con detalle este objeto

La clase HashTable: Interfaz Enumeration

- Al no estar basado en índices, un Hashtable no se puede recorrer utilizando una instrucción for con una variable que recorra las posiciones de los objetos.
- Esto no significa que no se pueda iterar sobre un Hashtable, se puede hacer a través de un objeto enumeration.
- Enumeration es un objeto que implemento la interfaz java.util.Enumeration. Las interfaces disponen de una serie de métodos que pueden ser aplicados sobre los objetos que las implementan,
- Los métodos proporcionados por la interfaz Enumeration permiten recorrer una colección de objetos asociada y acceder a cada uno de sus elementos. En el caso concreto del método keys() de la clase Hashlable, el objeto Enumeration devuelto nos permite recorrer la colección de claves del Hashtable.
- Un objeto Enumeration lo podemos imaginar como una especie de puntero o referencia, que puede ir apuntando a cada uno de los elementos de una colección.

La clase HashTable: Interfaz Enumeration - Métodos

- Object nextElement(). La llamada al método nextElement() sobre un objeto enumeration, provoca que éste pase a apuntar al siguiente objeto de la colección, devolviendo el nuevo objeto apuntado. Hay que tener en cuenta que, inicialmente, un enumeration se encuentra apuntando a la posición que está antes del primer objeto de la colección. por lo que la primera llamada a nextElement()) devolverá el primer objeto.
- boolean hasMoreElements(). Indica si hay más elementos por recorrer en la colección. Cuando el objeto enumeration esté apuntando al último elemento, la llamada a este método devolverá false.
- Con estos dos métodos, utilizando un bucle: while, se puede acceder a todos los elementos de la colección asociada al Enumeration.

La clase HashTable: Interfaz Enumeration

- El siguiente método recibe como parámetro un objeto Hashtable en el que se han almacenado objetos String con claves asociadas de tipo String, su misión consiste en mostrar en pantalla todos los valores almacenados, tendríamos que escribir el siguiente código.

```
public void muestraDatos(Hashtable hs){
    String valor, clave;
    Enumeration e = hs.keys();
    while (e.hasMoreElements()) {
        clave = (String) e.nextElement();
        //obtiene el objeto a partir de la clave
        valor = (String) hs.get(clave);
        System.out.println(valor);
    }
}
```

La clase HashTable: Ejemplo Completo

- Para aclarar el funcionamiento de esta clase se presenta a continuación un programa para la gestión de una lista de nombres. El programa presentará inicialmente un menú en pantalla para elegir la opción deseada (1. Añadir nombre, 2. Eliminar nombre, 3. Mostrar todos. 4. Salir), menú que volverá a presentarse de nuevo en la pantalla tras completar cualquiera de las tres primeras opciones. Cada nombre llevara asociado como clave un DNI:

```
public class GestionNombres {
   public static void main(String[] args) throws IOException{
      Hashtable nombres=new Hashtable();
      String opcion;
      BufferedReader bf=new BufferedReader(new InputStreamReader(System.in));
      do {
         System.out.println("Elegir opción :\n");
         System.out.println("1 . Añadir nombre");
         System.out.println("2 . Eliminar nombre");
         System.out.println("3 . Mostrar todos los nombres");
         System.out.println("4 . Salir");
         opcion = bf.readLine();
//Continúa…
```

La clase HashTable: Ejemplo Completo

```
switch (Integer.parseInt(opcion)) {
    case 1:
                String nom, dni;
      System.out.println(" Introduce Nombre: ");
                nom=bf.readLine();
      System.out.println("DNI : ");
      dni=bf.readLine();
      almacenaNombre(nom, dni, nombres);
      break;

    case 2:
      String d;
      System.out.println("Introduzca el dni: ");
      d=bf.readLine();
      eliminaNombre(d,nombres);
      break;

    case 3:
      mostrarTodos (nombres);
      break;
  }
 }
     while(!opcion.equals("4"));
} //del main
```

```
  static void almacenaNombre (String n, String k, Hashtable lista) {
    if (!lista.containsKey(k)) {
      lista.put(k,n);
    }
  }

  static void eliminaNombre (String k, Hashtable lista) {
    if(lista.containsKey(k)) {
      lista.remove(k);
    }
  }
  static void mostrarTodos (Hashtable lista) {
    System.out.println("Los nombres son: ");
    Enumeration claves = lista.keys();
    while (claves.hasMoreElements()) {
      String k=(String) claves.nextElement();
      System.out.println(k+" - "+lista.get(k));
    }
  }
} //de la clase
```

Genéricos

- Después de haber analizado el funcionamiento de las colecciones y de haber estudiado algunas de las clases más significativas , vamos a conocer las colecciones de tipo Object.
- El problema de las colecciones de tipo Object
- Para comprender en qué consiste esta característica, analicemos de nuevo el comportamiento de las colecciones a la hora de almacenar y recuperar los objetos. Como ya se explicó al principio, y se ha visto en los programas de ejemplo, debido a que las colecciones gestionan los objetos a través del tipo Object, cada vez que se intenta recuperar un objeto de la colección es necesario realizar una conversión explícita al tipo específico para poderlo manipular posteriormente:

```
ArrayList l=new ArrayList();
l.add("Cadena de prueba");

String s=(String)l.get(0);
System.out.println(s.length());
```

Genéricos

- Aparte de la incomodidad de las conversiones explícitas en la recuperación de objetos, la utilización de Object como tipo común tiene como consecuencia que el compilador no realice ninguna comprobación de tipo, ni al agregar el objeto a la colección, ni al recuperarlo, haciendo que el código sea inseguro. Esto significa que si por ejemplo, estamos tratando con una colección de cadenas de caracteres y por error se añade un objeto de cualquier otro tipo, el compilador no generará ningún error, produciéndose durante la ejecución del programa una excepción de tipo ClassCastException al intentar convertir el objeto a cadena:

```
ArrayList l=new ArrayList();
l.add("Cadena de prueba");
l.add("Segunda cadena");
l.add(10); //compila correctamente

for(int i=0;i<l.size();i++)
{
   String s=(String)l.get(i); //compila  bien  pero, generará  una  ClassCastException cuando i valga 2
   System.out.println(s.length());
}
```

Genéricos

- Colecciones de tipos genéricos
- La utilización de colecciones basadas en tipos genéricos proporciona un mecanismo que permite notificar al compilador el tipo de objetos que va a ser almacenado en la colección. Esto supone dos mejoras respecto al funcionamiento tradicional de las colecciones:
    - Cualquier instrucción que intente almacenar en la colección un objeto de un tipo que no sea el especificado provocará un error de compilación (siempre será mejor que el compilador nos avise de que algo no está bien antes de esperar a que se produzca una excepción).
    - Dado que se conoce el tipo de objeto almacenado en la colección. no será necesario realizar una conversión explícita durante su recuperación.
- Todas las clases e interfaces de colección del paquete java.util han sido redefinidas en la versión Java SE para poder soportar tipos genéricos.
- Para especificar el tipo de objetos a utilizar en una colección basada en tipos genéricos, se debe indicar dicho tipo en la declaración de la variable de colección mediante la utilización de la siguiente expresión:

tipo_colección <tipo_objeto> variable;

- Siendo tipo_coleccion la clase de colección utilizada y tipo_objeto la clase de los objetos que serán almacenados en ella.

Genéricos: Ejemplos

- Para declarar un ArrayList de cadenas de caracteres sería: ArrayList <String> lista;
- Así mismo, la creación del objeto colección debería realizarse según la expresión:

variable = new tipo_coleccion <tipo_objeto>();

- En el caso de un ArrayList de cadenas: lista = new ArrayList <String>();
- Es importante destacar que el tipo especificado durante la utilización de colecciones genéricas solamente puede ser de tipo objeto, no siendo posible utilizar tipos básicos Java. Por ejemplo, la siguiente declaración provocaría un error de compilación: ArrayList <int> lista; //no compila
- Una vez creado el objeto de colección, la variable podrá utilizarse normalmente para realizar las operaciones habituales, con la ventaja de que ya no será necesario realizar conversiones explícitas en la recuperación de los elementos:

```
ArrayList <String> l=new ArrayList<String>();
l.add ("Cadena de prueba");
l.add ("Segunda cadena");
l.add("Nueva cadena");
for (int i=0; i<l.size ();i++) {
   String s=l.get(i); //ausencia de casting
   System.out.println(s.length());
}
```

Si en el código anterior se hubiese intentado añadir un objeto no String a la colección, el compilador habría generado un error de compilación en la instrucción de llamada al método add():

```
l.add(5); //error de compilación
```

Genéricos: Ejemplos

- En el caso de una colección de tipo Hashtable, donde además de los elementos de la colección (objetos valor) se almacenan claves (objetos clave) asociadas a cada elemento, la utilización de genéricos permite especificar tanto el tipo del elemento como el de la clave.

Hashtable<tipo_clave, tipo_elemento>variable;

- Por ejemplo, para disponer de una colección Hashtable con elementos de tipo Empleado y clave asociada String sería: Hashtable<String, Empleado> tb;
- El siguiente listado es una nueva versión del programa de gestión de temperaturas con un ArrayList utilizando genéricos: (Siguiente diapositiva→)

Genéricos: Ejemplos

```
public class Gestion {
   public static void main (String [] args ) throws IOException{
   //Se declara un ArrayList de objetos Double
      ArrayList<Double> temperaturas = new ArrayList<Double>();
      String opcion;
      BufferedReader bf=new BufferedReader
                                                (new InputStreamReader(System.in));
      do{
         System.out.println("Elegir opción :\n");
         System.out.println("1. Añadir temperatura");
         System.out.println("2. Mostrar temperatura media.");
         System.out.println("3. Mostrar temperaturas extremas");
         System.out .println("4. Salir");
         opcion=bf.readLine();
         switch(Integer.parseInt(opcion)) {
            case 1:
               double temp;
               System.out.println("Introduce la temperatura: ");
               //Convierte a tipo double la temperatura leída
               temp=Double.parseDouble(bf.readLine());
               almacenaTemperatura(temp, temperaturas);
               break;
            case 2:
               muestraMedia(temperaturas);
               break;
            case 3:
               muestraExtremas(temperaturas);
         }
      }
      while (!opcion.equals("4"));} //del main

   static void almacenaTemperatura (double d,
                                    ArrayList<Double> temperaturas) {
   //necesita convertir el número a objeto para poderlo
   //añadir al ArrayList, aunque a través
   //del autoboxing podría haberse añadido directamente el valor double
         temperaturas.add(d);
      }
      static void muestraMedia (ArrayList<Double> temperaturas){
         double media=0.0;
         for (Double tp:temperaturas) { //no es necesario hacer el casting
            media+=tp.doubleValue();
         }
         media/= temperaturas.size();
         System.out.println("La temperatura media es:"+ media);
      }
      static void muestraExtremas(ArrayList<Double> temperaturas) {
      //se inicializan las variables extremo con
      //el valor de la primera temperatura
         double maxima;
         maxima= temperaturas.get(0).doubleValue();
         double minima=maxima;
         for (Double tp:temperaturas) {
            double aux;
            aux=tp.doubleValue();
            if (aux>maxima) { maxima=aux;}
            if (aux <minima) { minima=aux;}
         }
         System.out.println ("La temperatura máxima es máxima");
         System.out.println ("La temperatura mínima es mínima");
      }
   }
```

Genéricos: Ejemplos

- Podríamos obtener una versión más simple y reducida del programa anterior si, además de los genéricos, utilizamos el autoboxing/autounboxig para eliminar la conversión explícita entre tipo primitivo y objeto.
- En vez de usar un Buffer de lectura, vamos a usar la clase Scanner.

(Diapositiva siguiente →)

Genéricos: Ejemplos

```
public class Gestion {
   public static void main(String[] args) throws IOException {
      //Se declara un ArrayList de objetos Double
      ArrayList<Double> temperaturas = new ArrayList<Double>();
      int opcion;
      Scanner sc=new Scanner (System.in);
      do {
         System.out.println("Elegir opción :\n");
         System.out.println("1. Añadir temperatura");
         System.out.println("2. Mostrar temperatura media.");
         System.out.println("3. Mostrar temperaturas extremas");
         System.out.println("4. Salir");
         opcion = sc.nextInt();
         switch (opcion) {
            case 1:
               double temp;
               System.out.println(" Introduce la temperatura: ");
               //Recupera el dato como un double
               temp = sc.nextDouble();
               almacenaTemperatura(temp, temperaturas);
               break;
            case 2:
               muestraMedia(temperaturas);
               break;
            case 3:
               muestraExtrema(temperaturas);
               break;
         }
      }
      while (opcion != 4);
   }
```

```
   static void almacenaTemperatura(double d,
                        ArrayList<Double> temperaturas) {
      //autoboxing
      temperaturas.add(d);
   }

   static void muestraMedia(ArrayList<Double> temperaturas) {
      double media = 0.0;
      for (Double tp : temperaturas) {
         media += tp; //autounboxing
      }
      media /= temperaturas.size();
      System.out.println("La temperatura media es: " + media);
   }

   static void muestraExtrema(ArrayList<Double> temperaturas) {
   //Se inicializan las variables extremo con
   //el valor de la primera temperatura
      double maxima = temperaturas.get(0); //autounboxing
      double minima = maxima;
      for (Double tp : temperaturas) {
         if (tp > maxima) {
            maxima = tp;//autounboxing}
            if (tp < minima) {
               minima = tp;//autounboxing}
            }
         System.out.println("La temperatura máxima es " + maxima);
         System.out.println("La temperatura mínima es " + minima);
         }
      }
   }
}
```

Genéricos: Ejemplos

- Del mismo modo, el programa de la gestión de nombres con un Hashtable podemos verlo en la diapositiva siguiente →.

Genéricos: Ejemplos

```
public class GestionNombres2 {
   public static void main (String [] args) {
      Hashtable<String, String> nombres=new Hashtable<String, String>();
      int opcion;
      Scanner sc=new Scanner(System.in);
      sc.useDelimiter ("\n");
      do {
         System.out.println("Elegir opción:\n");
         System.out.println("1. Añadir nombre");
         System.out.println("2. Eliminar nombre");
         System.out.println("3. Mostrar todos los nombres");
         System.out.println("4.Salir");
         opcion=sc.nextInt();
         switch(opcion) {
            case 1:
               String nom,dni;
               System.out.println(" Introduce Hombre: ");
               nom=sc.next();
               System.out.println("DNI:  ");
               dni=sc.next();
               almacenaNombre(nom,dni,nombres);
               break;
            case 2:
               String d;
               System.out.println("Introduzca el  dni :");
               d=sc.next();
               eliminaNombre(d,nombres);
               break;
            case 3:
               mostrarTodos(nombres);
               break;
         }
      }
      while(opcion!=4);
   }
```

```
   static void almacenaNombre (String n, String k,
                        Hashtable<String, String> lista) {
      if (!lista.containsKey(k)) {
         lista.put(k,n);
      }
   }

   static void eliminaNombre(String k, Hashtable<String, String> lista) {
      if (lista.containsKey(k)) {
         lista.remove(k);
      }
   }
   static void mostrarTodos( Hashtable<String,String> lista) {
      System.out.println ("Los nombres son:  ");
      Enumeration<String> claves=lista.keys();
      while(claves.hasMoreElements()) {
         String k=claves.nextElement();
         System.out.println (k+" - "+lista.get(k));
      }
   }
}
```

Genéricos: Definición de tipos

- Para que sea posible especificar en la creación del objeto de colección los tipos de elementos que se pueden añadir, es necesario definir estas clases con una sintaxis especial. Si acudimos a la documentación del API de Java SE 6 para obtener información sobre la clase ArrayList, observamos cómo dicha clase aparece declarada de la siguiente manera:

```
class ArrayList <E>
```

- A esta forma de definir una clase se la conoce como definición con tipo parametrizado o definición de tipo genérico. La anterior declaración se lee "clase ArrayList de E", donde E, llamado también parámetro de tipo, es la letra utilizada para referirse al tipo de elementos que se pueden añadir y representa a cualquier tipo de objeto Java.

Genéricos: Definición de tipos

- Como hemos visto en el apartado anterior, es en la declaración de una variable de la clase ArrayList y en la creación de un objeto de la misma cuando se tiene que especificar el tipo concreto de objetos que se van a tratar en esa colección, sustituyendo la letra E por el nombre de clase correspondiente:

```
//ArrayList de enteros
ArrayList<Integer> n;
n = new ArrayList<Integer>();

//ArrayList de cadenas
ArrayList<String> cad;
cad = new ArrayList<String>();
```

Genéricos: Definición de tipos

- La declaración de este método de la clase ArrayList genérica a partir de la versión Java 5 tiene el formato:

```
bolean add(E o);
```

- Lo que significa que al ArrayList no se le puede añadir cualquier objeto, sino solamente objetos del tipo declarado en la colección (E). Al especificar un tipo concreto en la creación del objeto ArrayList, todas las referencias a E en los métodos de ese objeto serán sustituidas automáticamente por el tipo específico. Por ejemplo, para el objeto ArrayList de Integer referenciado por la variable n anterior, el formato del métodu add() quedaría convertido en:

```
bolean add(Integer o);
```

Genéricos: Definición de tipos

- Los tipos genéricos no se limitan solamente a las colecciones, también podemos definir clases o tipos genéricos propios. Por ejemplo, la definición del siguiente tipo genérico corresponde a una especie de clase de envoltorio capaz de encapsular cualquier tipo de objeto:

```
public class Wrapper<E> {

  //E: dato encapsulado que puede ser de cualquier tipo objeto
   private E data;

   public void setData(E d) {
      data = d;
   }

   public E getData() {
      return data;
   }
}
```

Genéricos: Definición de tipos - Ejemplo

- El siguiente programa muestra un ejemplo de utilización de esta clase para encapsular cadenas de caracteres:

```
public class PruebaData {
   public static void main(String[] args) {
      Wrapper<String> w = new Wrapper<String>();
      w.setData("mi cadena.");
      String d = w.getData();
      System.out.println("La cadena es : " + d);
   }
}
```

- Es importante destacar que el parámetro de tipo E representa un tipo objeto, lo que significa que sólo podrán utilizarse como argumentos de tipo referencias a objeto, nunca tipos primitivos. La siguiente instrucción produciría, por tanto, un error de compilación:

```
Wrapper<char> w=new Wrapper<char>();
```

¿Preguntas?

Bibliografía y referencias

- Ediciones ENI - Los fundamentos del lenguaje Java Ed. 2020
- Ediciones ENI – Java Ed. 2021
- Pearson, 2016 - Estructuras de Datos con Java: Diseño de estructuras y algoritmos.
- Concepto de Programación: https://concepto.de/programacion/
- Documentos de la API de Java: https://docs.oracle.com/en/java/javase/17/docs/api/index.html