JAVA

Lenguaje Programación OO

- Presentaciones.
  - Nombre.
  - Empresa.
  - Cargo, función y responsabilidad.
  - Experiencia relacionada con los temas del curso.
  - Razones para inscribirse al curso.
  - Expectativas con respecto al curso.

Conceptos

Introducción I

- Lenguaje de programación OO, fuertemente tipado, comercializado por primera vez en 1995 por *Sun Mycrosystems*, y adquirida en 2010 por Oracle.
  - Sigue siendo de código libre.
- Está presente en multitud de entornos y dispositivos, ordenadores, móviles, consolas, etc.
  - Su mayor implementación es en aplicaciones web de tipo cliente-servidor.
- Es multiplataforma, lo que quiere decir, que una vez su código es compilado (genera código bytes), puede ejecutarse en cualquier máquina donde haya un JRE.
  - JRE: *Java Runtime Environment*. Es decir, un entorno de ejecución del *bytecode*.
- Es portable, la JVM (*Java Virtual Machine*), es la máquina virtual de Java, esto es, la que es capaz de interpretar el bytecode generado en tiempo de compilación en cualquier arquitectura subyacente.

Introducción II

- Proporciona varias herramientas, algunas de las cuáles son:
  - Compilador (*javac*).
  - Intérprete (*java*).
  - Generador de documentos (*javadoc*).
  - Empaquetador de archivos de clases (*jar*).
- Es un lenguaje que deriva de C/C++, pero de más alto nivel.
  - No requiere de gestión de punteros.
  - Existe el denominado recolector de basura, o *garbage collector*, que se encarga de hacer limpieza de aquellas estructuras en memoria, que ya no van a ser usadas.
- Permite gestión multihilo (*threads*).
- Proporciona estructuras de datos (*collections*).
  - lists, sets, maps y queues.

Java Development Kit

- El JDK, es el entorno, que nos permite desarrollar y ejecutar aplicaciones Java de escritorio (J2SE).
  - J2EE, es más amplio, e incluye elementos, pensados para el entorno web.
- Hay varias implementaciones:
  - Open JDK. Implementación libre y de código abierto mantenida por la comunidad.
    - Es el que usaremos nosotros.
  - Oracle JDK. Similar a la anterior, pero es de pago, y da soporte a sus clientes.
  - AdoptOpenJDK.
  - Amazon Corretto.
- La diferencia entre el JDK y el JRE, es la necesidad que tengamos o no, de desarrollar código que vayamos a querer compilar.
- El JDK incluye lo mismo que el JRE, y éste último, a su vez, la JVM.

- URL Open JDK 15: http://jdk.java.net/15/

Variables de entorno

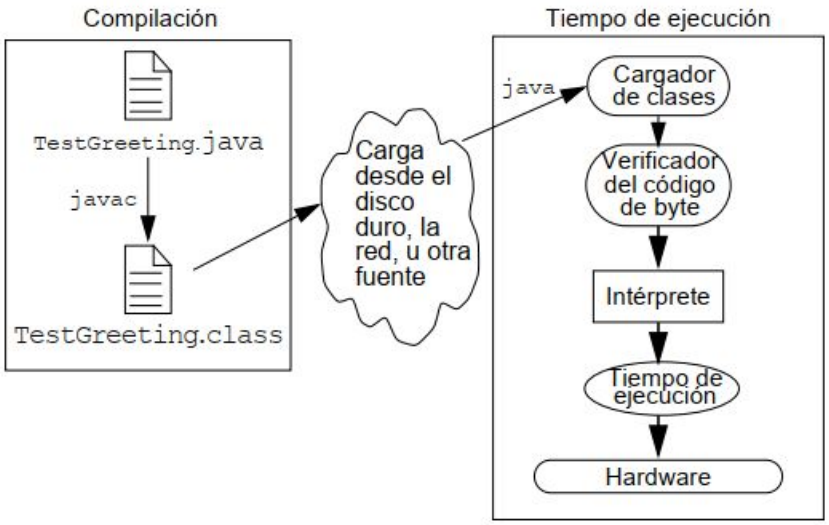
Una vez descargado y descomprimido el Open JDK en windows, precisamos…

- Indicar la ruta de acceso a la carpeta que contiene los ejecutables que nos permitirán compilar y ejecutar el código.
  - PATH='C:\Program Files\jdk-15.0.2\bin' o PATH='%JAVA_HOME%\bin'
  - JAVA_HOME='C:\Program Files\Java\jdk-15.0.2'
  - JAVA_JRE='C:\Program Files\Java\jdk-15.0.2'
- Para verificar que está correctamente configurado...

```
C:\Users\xabier.pastor>java -version
openjdk version "15.0.2" 2021-01-19
OpenJDK Runtime Environment (build 15.0.2+7-27)
OpenJDK 64-Bit Server VM (build 15.0.2+7-27, mixed mode, sharing)
```


- Las rutas JAVA_HOME y JAVA_JRE, no tienen porqué estar definidas. Son necesarias cuando hay programas, que para poder funcionar correctamente, precisan tener acceso al JDK de Java, y lo que hacen por defecto, es buscar dichas variables de entorno.
- Comandos ms-dos:
  - cd -> change directory
  - rm -> remove
  - mkdir -> make directory
  - rmdir -> remove directory
  - cd .. -> subir de nivel
  - cls -> limpiar pantalla
- CLASSPATH, se usa para indicar la ruta donde se encuentran las clases o jars, con los que trabajar tanto al compilar como ejecutar algo.
  - Normalmente, en lugar de definirla como variable de entorno, se usa la opción -cp o -classpath, en la línea de comandos.

Funcionamiento JDK/JRE



- El intérprete (comando *javac*), ejecuta el código *bytecode* (ficheros *.class) y efectúa las llamadas apropiadas al hardware, según la plataforma subyacente. Motivo por el cual, las versiones del JDK/JRE, son dependientes de la plataforma subyacente.
- El verificador de código, se asegura de que…
  - el formato de los ficheros es acorde a la especificación de la JVM (versión de compilación VS ejecución compatibles).
  - no haya intentos de infringir reglas de acceso.
  - no haya conversiones de tipos irregulares (tipos enteros a referencias de objetos).
  - etc.

Hola Mundo I

```java
public class HolaMundo{
    public static void main (String[] args){
        ComportamientosHumanos ch = new ComportamientosHumanos();
        ch.saludar();
    }
}
```

```java
public class ComportamientosHumanos{
    public void saludar(){
        System.out.println("Hola chic@s!");
    }
}
```

```
428 ComportamientosHumanos.class
107 ComportamientosHumanos.java
336 HolaMundo.class
151 HolaMundo.java
```

Para compilar este código…

1. Ir a la ruta de estos dos ficheros.
2. Ejecutar el compilador de java: *javac HolaMundo.java*

- Algo que es importante tener en cuenta, sobre todo, a la hora de comenzar a programar en un nuevo lenguaje de programación, es que es importante implementar código, y obtener errores, dado que así pulimos nuestras habilidades programáticas y algorítmicas.
- Al ejecutar la clase que contiene el método principal de 'entrada', esto es, el *main*, se determina la existencia de una clase (ComportamientosHumanos.java), que existe en la ruta actual, y que java compila (ComportamientosHumanos.class) e instancia (new).
- También sería posible usar el parámetro -d, para indicar una ruta donde dejar los ficheros compilados (bytecode), de la siguiente manera:
    - *javac -d /clasesJava *.java*
    - A tener en cuenta…
        - Es necesario que la carpeta *clasesJava* exista, porque sino, no los podrá copiar.
        - Al indicar una carpeta destino distinta a la actual, hay que indicar que queremos compilar todas las clases (*.java).

Hola Mundo II

El método main…

- Es el punto de entrada en una aplicación J2SE.
- El modificador de acceso debe ser *public*, para que sea accesible.
- La palabra clave *static*, indica al compilador, que no hay necesidad de tener una instancia de la clase, para poder ser accedido, y en este caso, por tratarse de un método, ejecutado.
- La palabra clave *void*, indica que el método *main*, no retorna valor alguno.
  - Importante porque hay una comprobación de que al llamar a un método, el retorno se corresponde con lo declarado previamente.
- *String args[]*, es el único parámetro de entrada, y se trata de un array de valores de tipo cadena.

Hola Mundo III

Para iniciar mi aplicación...

- Ejecutar el intérprete de comandos: *java HolaMundo*

```
C:\Users\xabier.pastor\Desktop\Ipartek\Cursos\Java\codigo\ej_holamundo>java HolaMundo
Hola chic@s!
```

- Podríamos pasar argumentos: *java HolaMundo Xabier*

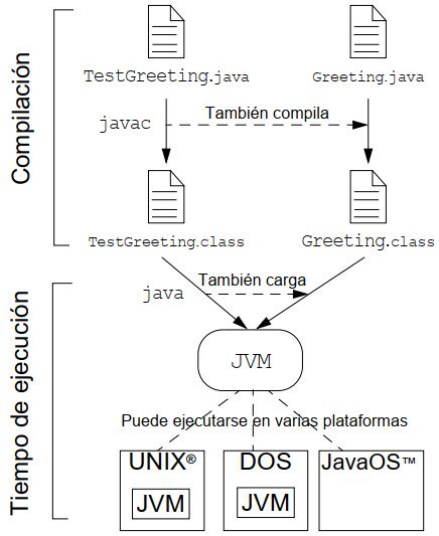
```
ic class HolaMundo{
public static void main (String[] args){
    ComportamientosHumanos ch = new Comp
    ch.saludar(args[0]);
```

```
lic class ComportamientosHumanos{
public void saludar(String nombre){
    System.out.println("Hola " + nombre + "!");
}
```

```
C:\Users\xabier.pastor\Desktop\Ipartek\Cursos\Java\codigo\ej_holamundo>java HolaMundo Xabier
Hola Xabier!
```


- Ejercicios.
  - Modificar el nombre del método println, por algo inventado, para ver qué error se produce al compilar el código.
  - Modificar el nombre de la clase HolaMundo, para ver qué error se produce al compilar el código.
    - Añadir una nueva clase, dentro del mismo fichero Java, para ver que se muestra el mismo error que antes, al compilar el código.
  - Modificar el nombre del método estático *main*, para verificar el error que se produce al ejecutar el código.
    - Se puede repetir la prueba, eliminando la palabra *static*, o eliminando el parámetro de entrada al método.
    - Si se modifica el nombre del parámetro de entrada, no es un problema, siempre y cuando siga siendo un array de String.
  - Modificar la llamada realizada, y no incluir ningún argumento de entrada, para ver el error que se produce al ejecutar el código.
  - Ejecutar la aplicación, poniendo el nombre de la clase, en minúsculas, y ver el error que se produce al ejecutar el código.

Multiplataforma



1 compilación (bytecode en ficheros .class), **n** posibles **ejecuciones** (JVM diferentes).

IDE

Integrated Development Environment, o entorno de desarrollo integrado…

- Es un software que proporciona servicios integrales a los desarrolladores.
  - Evita tener que realizar tareas usando la línea de comandos.
- Normalmente, está compuesto de:
  - Editor.
  - Herramientas de construcción automáticas.
  - Depurador para inspección y detección de errores en tiempo de ejecución.
  - *IntelliSense*, o lo que es lo mismo, autocompletado inteligente de nombre de clases, paquetes, etc.
    - Contribuyendo así, a una mejora en eficiencia de la programación.
  - Paleta con elementos visuales, para la construcción de interfaces gráficas.
- Algunos ejemplos son: Eclipse, Netbeans, JDeveloper, VS Code, etc.

Eclipse

- Debe ser compatible con el JDK que hemos instalado.
- Tras la instalación hay que definir una carpeta de trabajo por defecto para los proyectos, que puede modificarse a posteriori.

13

- Eclipse…
  - https://www.eclipse.org/downloads/
  - por defecto, al crear un nuevo proyecto, establece una carpeta para los ficheros fuentes (src), y otra para los compilados (bin).
  - escribiendo ‘syso’ y la combinación de teclas ‘Ctrl+Espace’, conseguimos un autocompletado de la sentencia ‘System.out.println();’
- Ejercicios.
  - Crear un primer proyecto en Eclipse configurando la jdk recientemente instalada (Run As/Run Configuration…), estableciendo la clase a ejecutarse (HolaMundo), y enviando un argumento con tu nombre.

POO Básico

Clase

- Es un prototipo de software que permite crear objetos a partir de la misma, mediante el operador *new*, consiguiendo instancias.
- Está compuesta por atributos (características) y métodos (comportamientos).
  - Siendo ambos, miembros de la misma clase.
- Al instanciar una clase, se crea un objeto, con determinados valores para sus atributos, e interactúa consigo mismo y con otros objetos, mediante los métodos definidos.
- Un fichero puede contener más de una clase, pero sólo una de ellas puede ser pública.

```
<modificador>* class <nombre_clase> {
  <declaración_atributo>*
  <declaración_constructor>*
  <declaración_método>*
}
```

15

- El modificador de acceso establece el tipo de clase que es:
  - *public*, visible para todas las clases, dentro o fuera del paquete en el que está esta clase.
  - La ausencia de modificador, significa que sólo es accesible por otras clases que se encuentran dentro del mismo paquete.
- El nombre de la clase, debe empezar por mayúsculas, y las distintas palabras que la conforman, deberían seguir el formato *camelCase*.
- Existe un compendio de palabras reservadas, como por ejemplo *class*, que no pueden ser usadas a la hora de declarar clases, atributos, métodos, etc.

Atributos

<modificador> <tipo> <nombre> [= <valor_inicial>];*

- Al igual que en una clase, el modificador sirve para establecer la visibilidad del atributo y el nombre debe seguir ciertas reglas, como por ejemplo, que no debe comenzar por números.
- El tipo, puede ser cualquier tipo primitivo o una clase. Y determinan el conjunto de posibles valores asignables.

```
public class Ejemplo {
  private int x;
  private float y = 10000.0F;
  private String name = "Hotel Mediodía";
}
```

Métodos

```
<modificador>* <tipo_retorno> <nombre> ( <argumentos>* ) {
  <sentencia>*
}
```

- Un retorno…
  - igual a void, significa que no devuelve ningún valor.
  - establece el tipo de valor devuelto en todas las rutas de retorno posibles.
  - se lleva a cabo, con la palabra *return*.
- Los argumentos…
  - deben ir separados por coma.
  - deben ir tipados.
  - deben ir declarados con un identificador válido.

```
public class Perro {
  private int peso;
  public int getPeso() {
    return peso;
  }
  public void setPeso(int newPeso) {
    if ( newPeso > 0 ) {
      peso = newPeso;
    }
  }
}
```

Atributos/Métodos

- En relación a los nombres, se recomienda el uso del formato camelCase.
- El modificador de acceso, es opcional, establece la visibilidad del atributo/método desde el código de otras clases contenidas, dentro o fuera de este paquete.
  - public
    - Visible desde cualquier clase.
  - protected
    - Visible desde esta clase, las del mismo paquete, y las que hereden de ella.
  - private
    - Visible sólo desde la propia clase.
- Si no se define ninguno de estos tres modificadores de acceso, la visibilidad es sólo posible desde la propia clase y las contenidas en su mismo paquete.

- Ejercicios.
  - Probar a declarar y acceder a métodos/atributos declarados en una clase, desde otras.

Ejemplo de accesos

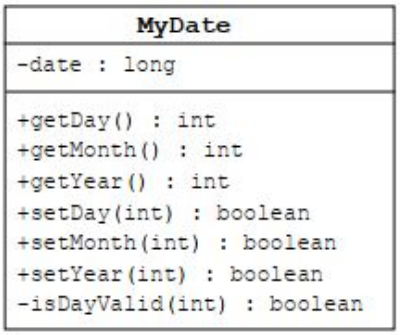
```
public class PruebaPerro {
  public static void main(String[] args) {
    Perro d = new Perro();
    System.out.println("El peso del perro d es "
                       + d.getPeso());
    d.setPeso(42);
    System.out.println("El peso del perro d es "
                       + d.getPeso());
    d.setPeso(-42);
    System.out.println("El peso del perro d es "
                       + d.getPeso());
  }
}
```

```
El peso del perro d es 0
El peso del perro d es 42
El peso del perro d es 42
```


- Ejercicios.
  - Crear una clase, que contenga tres atributos, día, mes y año, de tipo entero (int). Y añadirle métodos, para evitar que al mes, se le pueda establecer un valor que no menor que 1 o mayor que 12.

Encapsulación

- Ocultación del estado de los atributos de una clase, que sólo son accesibles y/o modificables, según los métodos definidos para ello.
- Métodos getter y setter.
  - Son unos métodos que suelen crearse para acceder a los atributos de una clase, controlando los valores que se establecen o se retornan.
- Permite definir una interfaz pública.



Constructores

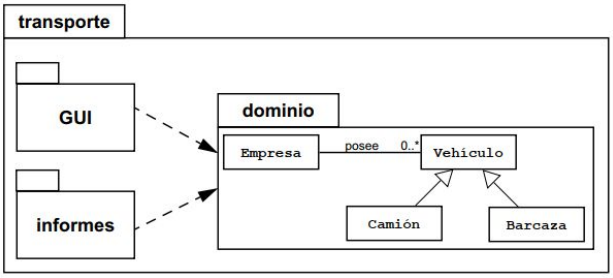
```
[<modificador>] <nombre_clase> ( <argumentos>* ) {
  <sentencia>*
}
```

```
public class Perro {
  private int peso;

  public Perro() {
     peso = 42;
  }
```

- Bloque de código, que sirve para inicializar una instancia de una clase, es decir, el que se ejecuta cuando usamos la palabra *new*.
- El nombre debe ser el mismo que el de la clase.
- En relación a los argumentos y al modificador, le aplican los mismos criterios que a los métodos.
- Se pueden crear tantos constructores como queramos (no repitiendo sintaxis), y pueden llamarse entre sí, aunque siempre debe haber al menos uno. Y de no ser así, el propio lenguaje se encargará de suministrarlo.

- En el momento en el que se suministra un constructor de manera explícita, el incluido por defecto desaparece.
- Ejercicios.
  - Añadir un constructor con un argumento y verificar cómo deja de ser posible instanciar objetos de una clase sin pasar argumentos.

Organizando el código I




```
[<declaración_paquete>]
<declaración_importación>*
<declaración_clase>+
```

- Los paquetes permiten organizar clases, normalmente por similitud semántica, es decir, de una forma lógica, según el modelo de que se trate.
- Los paquetes pueden contener a su vez más paquetes.

Organizando el código II

```
package <nombre_paq_superior>[.<nombre_paq_subordinado>]*;
```

- Sólo es posible incluir una declaración de paquete por archivo fuente, debe estar al comienzo del todo, y sólo precedido de comentarios.
  - Y si no la hay, la clase pertenece al paquete predeterminado (sin nombre).
- Lo normal es escribirlos en minúsculas, y el punto sirve para referenciar la estructura de paquetes a la que pertenece una clase.

```
import
<nombre_paq>[.<nombre_paq_subordinado>].<nombre_clase>;
```

o bien

```
import <nombre_paq>[.<nombre_paq_subordinado>].*;
```

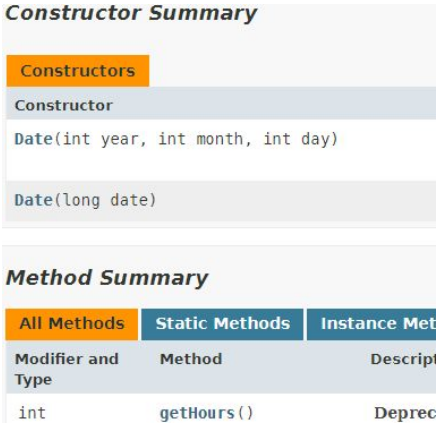

-

Organizando el código III

- Las importaciones en realidad son una ayuda para simplificar el código, y no tener que repetir, en el código implementado, accesos a clases contenidas en otros paquetes, al estilo de:
  - *private transporte.dominio.Empresa empresaParaInforme;*

```
package transporte.informes;

import transporte.dominio.*;
public class InformeCapacidadVehiculo {
     private Empresa empresaParaInforme;
  ...
}
```

API I



Constructor Summary

Constructors

Constructor
`Date(int year, int month, int day)`
`Date(long date)`

Method Summary

All Methods | Static Methods | Instance Met

Modifier and Type	Method	Descript
int	`getHours()`	Deprec

- Disponible para ser consultada de forma online:
  - https://docs.oracle.com/en/java/javase/15/docs/api/index.html
- O descargable:
  - https://www.oracle.com/es/java/technologies/javase-jdk15-doc-downloads.html

API II

- También es posible generar documentación de nuestro proyecto, mediante línea de comandos:
  - javadoc src*.java -d doc -sourcepath src -subpackages dominio dominio.animales
- O a través del IDE.

Para ello, es necesario documentar previamente las clases, junto con sus atributos y métodos.

```java
package dominio.animales;


/**
 * Esta es una clase de prueba
 * @author xabier.pastor
 *
 */
public class Perro {
    /**
     * Clase constructora que no recibe parámetros
     */
    public Perro(){}
    /**
     * Atributo del chip del perro
     */
    private int chip =0;

    /**
     * Getter del atributo chip
     * @param chip Identificación del perro
     */
    public void setChip (int chip) {
```


- En eclipse la generación de la documentación es vía menú: *Project/Genrerate Javadoc...*

Normas y Tipos

Código

- Además del comentario *javadoc*, tenemos…
  - El de línea, comienza por //.
  - El de varias líneas, que engloba al código con /**/
- Una sentencia termina, con el carácter ;
  - Aunque el código pueda estar en más de una fila.
- Un bloque de sentencias debe quedar englobado con {}
  - En estructuras de control, si sólo hay una sentencia, la inclusión de llaves, es opcional.
  - Pueden darse anidaciones.
- Los espacios en blanco son opcionales, pero proporcionan legibilidad.

```
{
  int x;

  x = 23 * 54;
}
```

```
{int x;x=23*54;}
```

Identificadores

- Nombre asignado a una variable, método o clase.
- Deben comenzar por una letra, el símbolo _, ó el $.
  - El resto de caracteres pueden ser números.
- No hay limitación de longitud máxima.
- Es case-sensitive, por lo que la variable ‘raza’, es distinta de ‘rAza’.
- Palabras reservadas:

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

Tipos primitivos I

- Lógicos: boolean.
  - Posibles valores: *true* o *false*.
- Textuales: char.
  - Representa un único carácter.
  - Valor asignado entre comillas simples.
- Enteros: byte, short, int y long.
  - Permiten notación decimal (2), octal (077) o hexadecimal (0xBAAC).
  - Representan números con signo.
  - Sufijo l/L para valores long (2L).

Longitud del entero	Nombre o tipo	Rango
8 bits	`byte`	De -2^7 a 2^7-1
16 bits	`short`	De -2^{15} a $2^{15}-1$
32 bits	`int`	De -2^{31} a $2^{31}-1$
64 bits	`long`	De -2^{63} a $2^{63}-1$

- En el caso de asignar un valor literal de tipo long, se recomienda el uso del sufijo L, por temas de legilibilidad.
- Es posible declarar varias variables a la vez, indicando una única vez el tipo de las mismas (*int i,y;*).

Tipos primitivos II

- Reales en coma flotante: double y float. Un literal…
  - la notación decimal se lleva a cabo mediante el símbolo .
  - con letra e/E, sirve para definir un exponente.
  - con sufijo f/F ó d/D define el tipo del valor.
  - por defecto, es de tipo doble.
    - Por lo que sin sufijo, no puede ser asignado a una variable de tipo float.

`3.14`	Valor en coma flotante sencillo (double) en notación estándar
`6.02E23`	Valor en coma flotante largo
`2.718F`	Valor corto de precisión simple (`float`)
`123.4E+306D`	Valor largo `double` con `D` redundante

Longitud	Nombre o tipo
32 bits	`float`
64 bits	`double`

- Si a un valor decimal no le indicas el tipo, por defecto será *double*.
- 6.02E23 = 6,02x10 a la 23 (float)
- 123.4E+306D = 123,4x10 a la 306 (double)

Tipos primitivos III

Lista de tipos de datos primitivos del lenguaje Java			
Tipo	*Tamaño*	*Valor mínimo*	*Valor máximo*
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807
float	32 bits	-3.402823e38	3.402823e38
double	64 bits	-1.79769313486232e308	1.79769313486232e308
char	16 bits	**'\u0000'**	**'\uffff'**

Nota: un dato de tipo carácter se puede escribir entre comillas simples, por ejemplo **'a'**, o también indicando su valor Unicode, por ejemplo **'\u0061'**.

String

- Tipo de clase (referencia), que sirve para representar cadenas de caracteres.
- Se pueden asignar cadenas literales directamente, o mediante el constructor de la clase *String()*, junto con el operador *new*.

```
String nombrePerro = "Dako";
String nombreGato = new String("Missi");
```

- Los valores literales deben escribirse entre comillas.
- Las secuencias de escape se especifican con el carácter \
  - String saludo = “Hola!\n”
  - Existen lo métodos print/println, del objeto “System.out”, para imprimir por pantalla.

Tipo referencia I

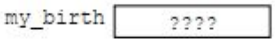
Creamos la clase *MyDate*, y la instanciamos...

```
public class MyDate {
  private int day = 1;
  private int month = 1;
  private int year = 2000;
  public MyDate(int day, int month, int year) { ... }
  public String toString() { ... }
}
```

```
public class TestMyDate {
  public static void main(String[] args) {
    MyDate today = new MyDate(22, 7, 1964);
  }
}
```

Tipos referencia II

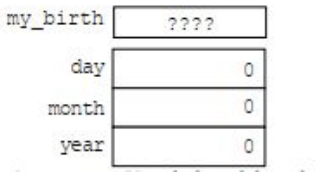
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



El uso de la palabra clave *new*...

1. reserva espacio en memoria, asignando 0/null.

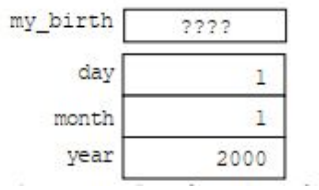
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



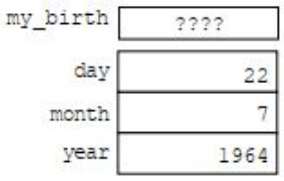
Tipos referencia III

2. se aplica, si existe, la inicialización explícita especificada.
   - que aplica a la declaración de atributos de la clase.
3. se ejecuta el constructor de la clase, pasándole, si existen, y por tanto han sido declarados con anterioridad, los argumentos especificados.

```
MyDate my_birth = new MyDate(22, 7, 1964);
```



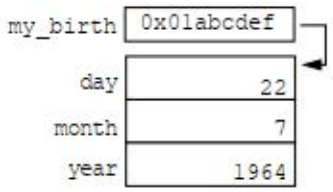

```
MyDate my_birth = new MyDate(22, 7, 1964);
```



Tipos referencia IV

4. se asigna la referencia del nuevo objeto en memoria dinámica a la variable.

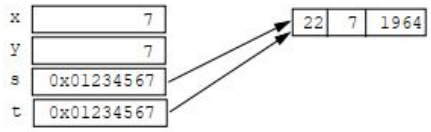
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



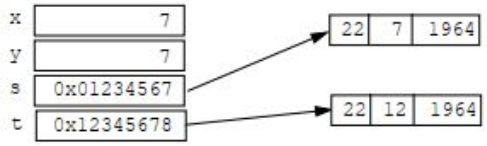
Tipo referencia V

Asignación de referencias a otras variables…

```
int x = 7;
int y = x;
MyDate s = new MyDate(22, 7, 1964);
MyDate t = s;
```




```
t = new MyDate(22, 12, 1964); // reasigna la variable
```



Paso por valor

- Un método que recibe parámetros, no puede modificar sus valores…
  - Si es de tipo primitivo, el valor es un literal.
  - Si es de tipo referencia, el valor es una referencia en memoria.
    - Los elementos del objeto, sí pueden modificarse.

```
public static void changeInt(int value) {
  value = 55;
}
public static void changeObjectRef(MyDate ref) {
  ref = new MyDate(1, 1, 2000);
}
public static void changeObjectAttr(MyDate ref) {
  ref.setDay(4);
}
```

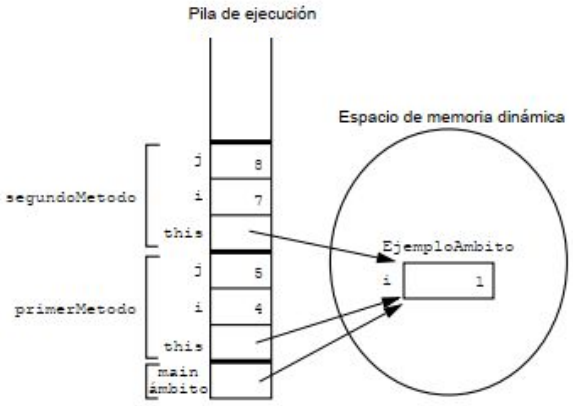
Ámbito I

- Las variables fuera de los métodos pueden ser…
  - De instancia o miembro.
    - Se construyen cuando se realiza una llamada a *new*.
    - Existen mientras exista el objeto asociado.
  - De clase.
    - Se construyen cuando se carga la clase.
    - Se diferencian por incluir la palabra reservada *static* en su declaración.
- Las variables locales, son aquellas que se definen dentro de un método.
  - Precisan ser inicializadas explícitamente antes de ser usadas.
  - Los parámetros de un método también son variables locales, pero son inicializadas por el código llamante.
  - Existen mientras se ejecuta el método.

- Las variables locales, también se denominan automáticas, temporales o de pila.

Ámbito II

```
public class EjemploAmbito {
  private int i=1;

  public void primerMetodo() {
    int i=4, j=5;
    this.i = i + j;
    segundoMetodo(7);
  }
  public void segundoMetodo(int i) {
    int j=8;
    this.i = i + j;
  }
}

public class PruebaAmbito {
  public static void main(String[] args) {
    EjemploAmbito ambito = new EjemploAmbito();
    ambito.primerMetodo();
  }
}
```



Inicialización

Si el código no incluye una inicialización explícita para una variable…

- fuera de un método…
  - se lo proporcionará el compilador.
- dentro de un método…
  - el compilador lo señalará con un error de uso antes de su inicialización.

Estas reglas, son independientes del tipo de variable de que se trate, esto es, primitivo, o clase/referencia.

Variable	Valor
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000'
boolean	false
Todos los tipos de referencia	null

El intento de usar una variable con valor *null*, genera una excepción en tiempo de ejecución.

Referencia *this*

Usada con dos fines...

- resolver la ambigüedad entre variables de instancia y parámetros.

```
public MyDate(int day, int month, int year) {
  this.day   = day;
  this.month = month;
  this.year  = year;
}
```

- pasar el objeto actual a otro método.

```
boolean esBisiesto = OperacionesFecha.esBisiesto(this);
String aImprimir = new String ("La fecha actual, ");
aImprimir += esBisiesto ? " es un año bisiesto. ": "no es un año bisiesto." ;
System.out.println(aImprimir);
```

Convenciones

- Nombres de paquetes: en minúsculas.
  - *package transporte.objetos;*
- Nombres de Clases o Interfaces: sustantivos *UpperCamelCase*.
  - *class LibroContabilidad {...}*
  - *interface Contabilidad*
- Métodos: verbos *lowerCamelCase*.
  - *public void imprimirGastos(){...}*
- Variables: nombres *lowerCamelCase*.
  - *Animal miMascota = new Animal("Dako");*
- Constantes primitivas: nombres *UPPER_CASE*.
  - *final int GASTO_MAXIMO=2500;*
- Uso de {}, incluso cuando hay una sola sentencia.
- Una única sentencia por línea.
- Usar indentaciones para facilitar la lectura del código.

Estructuras de Control

Operadores

- Aritméticos. +, -, *, /, %
  - Respetando el orden de precedencia estudiado en matemáticas.
- Relacionales. >, <, >=, <=, ==, !=
- Asignación. =, +=, -=, *=, /=, %=
- Unario. ++, --
  - Distinto efecto, si es colocado antes/después de la variable.
- Lógicos. ! (NOT), & (AND), ^ (XOR), | (OR)
- Cortocircuito. || ó &&.
- Concatenación de cadenas: +
  - Si uno de los argumentos del operador es un String, el resto son convertidos a objetos String.
- Casting: ()

Casting

Es posible asignar un valor de un tipo de variable a otra de otro tipo, siempre y cuando, haya compatibilidad entre ellos.

```
long valorGrande = 99L;
int squashed = valorGrande;       // Incorrecto, necesita
                                  //  conversión de tipos
int squashed = (int) valorGrande;   //  Correcto

int squashed = 99L;               // Incorrecto, necesita
                                  //  conversión de tipos
int squashed = (int) 99L;           //  Correcto, pero...
int squashed = 99;                //  literal entero
                                  //  predeterminado

long valgrande = 6;     // 6 es un tipo int, correcto
int valormenor = 99L;  // 99L es un tipo long, incorrecto

double z = 12.414F;  // 12.414F es un tipo float, correcto
float z1 = 12.414;   // 12.414 es double, incorrecto
```

Estructuras de control - If

Ejecución selectiva de partes del código en función de una expresión…

```
if ( <expresión_booleana> )
  <sentencia_o_bloque>
```

```
if ( <expresión_booleana> )
  <sentencia_o_bloque>
else
  <sentencia_o_bloque>
```

```
if ( <expresión_booleana> )
  <sentencia_o_bloque>
else if ( <expresión_booleana> )
  <sentencia_o_bloque>
```


- Expresiones booleanas, no valores numéricos.
- Los bloques else o else if, son opcionales.

Estructuras de control - Switch

```
switch ( <expresión> ) {
  case <constante1>:
    <sentencia_o_bloque>*
    [break;]
  case <constante2>:
    <sentencia_o_bloque>*
    [break;]
  default:
    <sentencia_o_bloque>*
    [break;]
}
```

```
switch ( modeloAutomovil) {
  case DELUXE:
    agregarAireAcondicionado();
  case ESTANDAR:
    agregarRadio();
  default:
    agregarRuedas();
    agregarMotor();
}
```

- El valor de la expresión debe ser compatible con un tipo entero.
  - Es posible usar tipos enumerados.
- El bloque default se ejecuta cuando no hay coincidencia con ninguno de los valores indicados en los case.
- break/return evita que el resto del código se siga ejecutando.

Estructuras de control - for

Bloque repetitivo que se puede ejecutar de 0 a n (siendo n un nº determinado).

```
for ( <expr_inicial>; <expr_prueba>; <expr_alter> )
  <sentencia_o_bloque>
```

```
for ( int i = 0; i < 10; i++ ) {
  System.out.println(i + " al cuadrado es " + (i*i));
}
```

- Aunque el bloque tenga una única sentencia, se recomienda el uso de {}.
- Las variables declaradas tienen ámbito de bucle, fuera de ellas no existen.
- Es posible inicializar y usar más de una variable, aunque sólo es posible declararlas de un tipo.

Estructuras de control - while y do/while

- Bloque repetitivo donde el número de ejecuciones no está establecido.

```
while ( <expr_prueba> )
  <sentencia_o_bloque>
```

```
do
  <sentencia_o_bloque>
while ( <expr_prueba> );
```

- La principal diferencia…
  - bucle while, la expresión se evalúa antes de ejecutar el bloque de código (0-n).
  - bucle do/while, la expresión se evalúa tras cada ejecución del bloque de código (1-n).
- Es importante actualizar la variable de control para evitar bucles infinitos.
- Tanto en estas estructuras como la de tipo *for*, es posible detener la iteración actual (*continue*), o todas las pendientes (*break*).

Arrays

Declaración

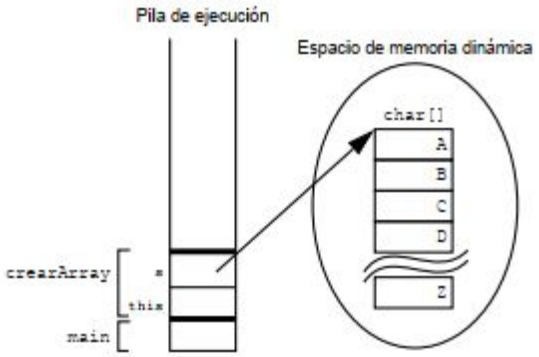
- Sirven para agrupar objetos, de tipos primitivos o referencia.
- Un Array es un objeto, pese a ser declarado de tipos primitivos.
- Al declarar una variable, no se crea un objeto, sino una referencia que puede ser usada para hacer referencia a un array.
  - La memoria se asigna de forma dinámica con un *new* o inicializador del array.

```
char[] s;
Point[] p; // donde Point es una clase
```

- Es posible declarar las variables, con los corchetes después del nombre de la variable, pero se desaconseja por legilibilidad.

Creación - Tipos primitivos

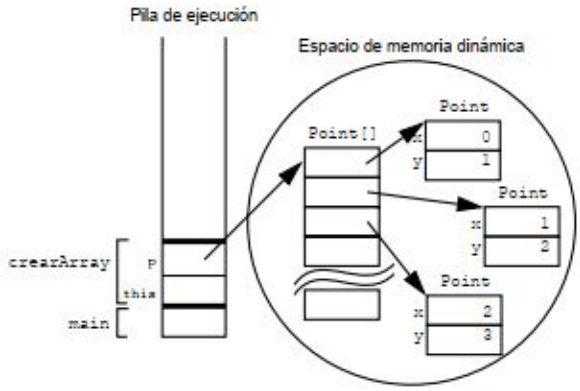
```
public char[] crearArray() {
  char[] s;

  s = new char[26];
  for ( int i=0; i<26; i++ ) {
    s[i] = (char) ('A' + i);
  }

  return s;
}
```



- Después de la declaración (*new char[26]*), los elementos del array se inicializan con el caracter null (‘\u0000’), que es un espacio en blanco.
- Los índices de un array siempre empiezan por 0 y terminan en n-1.

Creación - Tipos referencia

```
public Point[] crearArray() {
  Point[] p;

  p = new Point[10];
  for ( int i=0; i<10; i++ ) {
    p[i] = new Point(i, i+1);
  }

  return p;
}
```



Después de la **declaración** (*new Point[10]*), los elementos del array se inicializan con el valor *null*. En la **creación** es donde hacen referencia a un objeto Point.

Inicialización

Es posible inicializar los elementos del array de manera abreviada...

```
String[] names = {
  "Julia",
  "Juan",
  "Alfredo"
};
```

≃

```
String[] names;
names = new String[3];
names[0] = "Julia";
names[1] = "Juan";
names[2] = "Alfredo";
```

```
MyDate[] dates = {
  new MyDate(22, 7, 1964),
  new MyDate(1, 1, 2000),
  new MyDate(22, 12, 1964)
};
```

Multidimensionales

Es posible crear arrays de arrays...

```
int[][] dosDimen = new int [4][];
dosDimen[0] = new int[5];
dosDimen[1] = new int[5];
```

```
dosDimen[0] = new int[2];
dosDimen[1] = new int[4];
dosDimen[2] = new int[6];
dosDimen[3] = new int[8];
```

```
int[][] dosDimen = new int[4][5];
```


- En la declaración, no hubiera sido posible: *int[][] dosDimen = new int [][4];*

Iteración

- Todos los arrays poseen la propiedad *length*, que permite saber cuál es el número de elementos de un array, algo que facilita su iteración.

```
public void printElements(int[] list) {
  for ( int i = 0; i < list.length: i++ ) {
    System.out.println(list[i]); } }
```

- Desde J2SE 1.5, existe el bucle for mejorado:

```
public void printElements(int[] list) {
  for ( int element : list ) {
    System.out.println(element); } }
```

Manipulación

- No es posible redimensionar un array, sólo cambiar el objeto al que hace referencia la variable:

```
int[] miArray = new int[6];
miArray = new int[10];
```

- El método *arraycopy* de la clase *System*, proporciona la utilidad de copia, tanto de tipos primitivos como de clase.

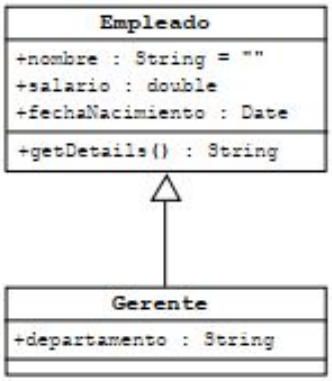
```
// array original
int[] miArray = { 1, 2, 3, 4, 5, 6 };

// nuevo array más largo
int[] copia = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

// copia todos los elementos de miArray en el array
// copia, empezando por el índice 0
System.arraycopy(miArray, 0, copia, 0,
                 miArray.length);
```


- Si el array contiene tipos referencia, si asignamos el mismo objeto a dos arrays distintos, al tocar el objeto en uno de los arrays, el otro se ve afectado. Para evitarlo, podemos hacer uso de *arraycopy*, que en la copia crea nuevos objetos y asigna los valores que lo componen.

POO Avanzado

Herencia

- Creación de una clase que extiende la funcionalidad de una padre.




```
public class Empleado {
  public String nombre = "";
  public double salario;
  public Date fechaNacimiento;

  public String getDetails() {...}
}


public class Gerente extends Empleado {
  public String departamento;
}
```

- Una clase sólo puede heredar de una única clase (herencia simple).

- Los constructores de una clase, no se heredan, sólo se da con las propiedades y métodos.

Sobrescritura métodos

Método que una subclase reescribe, manteniendo nombre, argumentos y retorno.

- El modificador de acceso no puede ser más restrictivo.
- El orden de los argumentos y su tipo deben ser los mismos.
- El retorno puede ser un subtipo del tipo definido.

```
public class Empleado {
  protected String nombre;
  protected double salario;
  protected Date fechaNacimiento;

  public String getDetails() {
    return "Nombre: " + nombre + "\n"
           + "Salario: " + salario;
  }
}
```

```
public class Gerente extends Empleado {
  protected String departamento;

  public String getDetails() {
    return "Nombre: " + nombre + "\n"
           + "Salario: " + salario + "\n"
           + "Gerente de: " + departamento;
  }
}
```

- Mediante *super*, es posible invocar métodos heredados.

- Con métodos heredados, nos referimos a cualquier método, de la clase padre, o antecesoras a ella.
- Un método marcado con la anotación *@Override*, existente desde la versión 1.5 de J2SE, debería estar sobreescribiendo un método de alguna clase padre. Si esto no es así, el compilador de java generará un error.

Polimorfismo

Cuando un variable de declara de un tipo, y el objeto asignado es de un subtipo, a la hora de ejecutar un método se hace sobre el del objeto en memoria.

```
Empleado e = new Gerente();
e.getDetails();
```

- Posibilita gestionar objetos de varios subtipos como un único tipo.
  - En colecciones.

```
Empleado [] personal = new Empleado[1024];
personal[0] = new Gerente();
personal[1] = new Empleado();
personal[2] = new Ingeniero();
```

  - En argumentos o retornos de un método.

```
public TipoImpositivo hallarTipoImpositivo(Empleado e) {...}
```


- Todas las clases son una subclase de tipo Object (*extends Object*), por lo que sería posible crear colecciones o métodos de este tipo, y usarlos con cualquier clase.
- Tanto en la colección como en el método, sólo se tendrá acceso a los miembros (variables/métodos) que estén definidos en la clase del tipo declarado.

Instanceof / Casting

Permite saber el objeto al que realmente hace referencia una variable, y con ello, la posibilidad de convertir la variable de referencia para acceder a sus miembros.

```
public void hacerAlgo(Empleado e) {
  if ( e instanceof Gerente ) {
     Gerente m = (Gerente) e;
     System.out.println("Éste es el gerente de "
                        + m.getDepartamento());
  } else if ( e instanceof Ingeniero ) {
    // Procesar un Ingeniero
  } else {
    // Procesar cualquier otro tipo de Empleado
  }
}
```


- Si hacemos el casting a un tipo, que en tiempo de ejecución, no es posible, el compilador lanzará una excepción.

Sobrecarga métodos

Reutilización del mismo nombre de método donde...

- El número y/o tipo de argumentos *debe* variar.
- El tipo de retorno *puede* variar.

```
public void saltar() {
    System.out.println( "sin args"); }
public void saltar(int i) {
    System.out.println("salto de '"+ i +"' metros."); }
public void saltar(String forma) {
    System.out.println( "salto con la forma: " + forma); }
public void saltar(int i, String forma) {
    this.saltar(i);
    this.saltar(forma);  }
```

Argumentos variables

Permiten definir un único método, que puede ser invocado con distintos argumentos del mismo tipo.

- El argumento es un array de objetos del tipo definido.

```
public class Estadistica {
  public float average(int... nums) {
    int sum = 0;
    for ( int x : nums ) {
      sum += x;
    }
    return ((float) sum) / nums.length;
  }
}
```

```
float gradePointAverage = stats.average(4, 3, 4);
float averageAge = stats.average(24, 32, 27, 18);
```

Sobrecarga constructores

Proporcionan distintas posibilidades para la creación de un objeto de una clase.

```
public class Empleado {
  private static final double SALARIO_BASE = 15000.00;
  private String nombre;
  private double salario;
  private Date fechaNacimiento;

  public Empleado(String nombre, double salario, Date FdeNac) {
    this.nombre = nombre;
    this.salario = salario;
    this.fechaNacimiento = FdeNac;
  }
  public Empleado(String nombre, double salario) {
    this(nombre, salario, null);
  }
  public Empleado(String nombre, Date FdeNac) {
    this(nombre, SALARIO_BASE, FdeNac);
  }
}
```


- Si se utiliza la palabra *this*, dentro de un constructor, para llamar a otro de la misma clase, debe aparecer en la primera sentencia.

Constructores superiores

Se puede invocar a constructores, no privados, de la clase padre.

```
public class Gerente extends Empleado {
  private String departamento;

  public Gerente(String nombre, double salario, String depto) {
    super(nombre, salario);
    departamento = depto;
  }
  public Gerente(String nombre, String depto) {
    super(nombre);
    departamento = depto;
  }
  public Gerente(String depto) { // Este código da error: no hay super()
    departamento = depto;
  }
}
```


- Si no se indica un constructor por defecto, el compilador, proporciona uno por defecto.
  - Si se indica al menos uno explícitamente, el de por defecto no se añade.
- Todo constructor tiene una llamada a la clase padre sin argumentos (*super();*).
  - Si la clase padre no dispone de tal constructor, se produce un error de compilación.
  - Si se utiliza la palabra *super*, dentro de un constructor, para llamar a otro de una clase superior, debe aparecer en la primera sentencia. Y conlleva la no inclusión, por parte del compilador, de esta llamada al constructor padre sin argumentos.
- Ejercicios.
  - Incluir un System.out.println, en el cuerpo de los constructores para comprobar el código que se ejecuta en primer lugar.

Object - equals() / toString()

Toda clase, al heredar de la clase Object, hereda también el método…

- *public boolean equals (Object obj)*, que por defecto compara dos referencias.
- *public String toString()*, que por defecto retorna el nombre de la clase y su dirección de referencia.

```
class Punto{
    int x,y;

    Punto (int x, int y){ this.x =x; this.y = y; }

    public boolean equals(Punto p) {
        return this.x == p.x && this.y == p.y;
    }

    public String toString(){
        return "Punto x: " + this.x + " y: " + this.y;
    }
}
```


- Ejercicios.
  - Crear 2 objetos, sin haber llegado a sobrescribir el método *equals*, crear dos objetos punto con los mismos valores, y realizar una comparativa con *equals*.
  - Hacer lo mismo con el método *toString*.

Clases envoltorio I

Permiten usar los tipos primitivos como objetos mediante el paquete *java.lang*:

Tipo de datos primitivos	Clase envoltorio
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Clases envoltorio II

- Una vez se inicializa su valor, éste es inmutable.
- Boxing / Unboxing:

```
int entero = 3;
Integer iEntero = Integer.valueOf(entero);
System.out.println("Valor de iEnero: " + iEntero);
int enteroAux = iEntero.intValue();
System.out.println("Valor de enteroAux: " + enteroAux);
```

- Auto-boxing / Auto-unboxing:

```
int entero = 3;
Integer iEntero = entero;
System.out.println("Valor de iEnero: " + iEntero);
int enteroAux = iEntero;
System.out.println("Valor de enteroAux: " + enteroAux);
```


- Previo a la versión 9, también se solía usar *Integer iEntero = new Integer(entero);* para construir un wrapper, pero ahora está *Deprecated*.
- Se recomienda el uso de boxing/unboxing explícito, por temas de rendimiento, dado que empeora con el automático (especialmente si nos encontramos con expresiones matemáticas dentro de bucles).

Miembros de clase I

Mediante la palabra reservada *static*, podemos definir atributos y métodos accesibles sin la necesidad de tener que instanciar la clase.

- El método *main()*, es un método de clase, llamado por la JVM.

```
public class Empleado {
    private static int cantidad = 0;
    public Empleado() {
        Empleado.cantidad++;
    }
    public static int getCantidad() {
        return Empleado.cantidad;
    }
}
```


- En el ejemplo, es posible no indicar el prefijo *Empleado*, para acceder a los miembros estáticos de la clase, dado que estamos dentro de ella. Aunque incluyéndolo, mejoramos la legilibilidad del código.

Miembros de clase II

- Intentar acceder a miembros no estáticos, genera error de compilación.

```
public static int getCantidad() {
    this.nombre;
    return cantidad; }
```

- No existe sobreescritura de métodos estáticos.
- Es posible llamar a miembros estáticos desde una variable de referencia.

```
Empleado e = new Empleado ("Juan");
System.out.println("Nº empleados: " + e.getCantidad());
```

- Se pueden crear bloques estáticos, que se ejecutan en la carga de la clase.

```
public class Empleado {
    private static int cantidad;
    static{ cantidad = 0; }
```

Modificador Final

Es una palabra reservada que según dónde se aplique tiene una connotación.

- Clase.
  - Impide que su extensión por subclases.
  - Usada por ejemplo con *java.lang.String*.

```
public final class Utilidades { }
```

- Método.
  - Impide que sea sobrescrito por subclases.
  - Usada para evitar que determinada implementación cambie.

```
public final void imprimirNomina() {}
```

- Variable.
  - La define como constante.
  - Si hace referencia a un objeto, el tipo no podrá cambiar, aunque sus valores subyacentes, sí.

```
public class Empleado {
    private static final float CUOTA_SOCIO = 6.5F;
```


- El enlazado dinámico de un método, puede obviarse por el compilador cuando éste se encuentra marcado como *static*, *private* o *final*.
  - Esto conlleva una mejora en el rendimiento en tiempo de ejecución.
- Las variables finales no tienen porqué inicializarse en la declaración, aunque al hacerlo sólo será posible en un único punto:
  - Si está definida como miembro de una clase, tendrá que hacerse en el constructor.
  - Si está definida como variable local a un método, tendrá que hacerse en algún punto del bloque.

Tipos enumerados I

Número finito de nombres simbólicos representando valores de un atributo.

```
public enum Palo {
  PICAS,
  CORAZONES,
  TREBOLES,
  DIAMANTES
}
```

```
public class NaipeBaraja {
  private Palo palo; private int rango;
  public NaipeBaraja(Palo palo, int rango) {
    this.palo = palo; this.rango = rango;  }
  public Palo getPalo() { return palo; }
}
```

```
public class TestNaipeBaraja {
  public static void main(String[] args) {
    NaipeBaraja naipe1 = new NaipeBaraja(Palo.PICAS, 2);
    // NaipeBaraja naipe2 = new NaipeBaraja(47, 2);
  } }
```

Tipos enumerados II

Se pueden usar atributos y métodos, al igual que en una clase.

```
public enum Palo {
  PICAS    ("Picas"),
  CORAZONES    ("Corazones"),
  TREBOLES     ("Treboles"),
  DIAMANTES ("Diamantes");

  private final String name;

  private Palo(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

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

    NaipeBaraja naipe1
      = new NaipeBaraja(Palo.PICAS, 2);
    System.out.println("el naipe1 es: " +
      naipe1.getPalo().getName());
  }
}
```

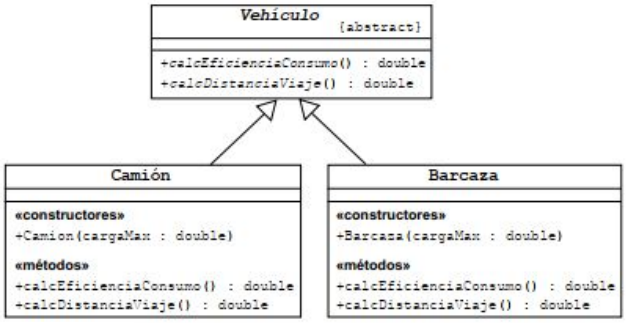

Es posible realizar importaciones estáticas, que afectan tanto a los miembros estáticos de una clase, como a los tipos enumerados.

- *import static dominio.Empleado.*;* o *import static dominio.cartas.Palo.*;*
- Básicamente permiten reducir el código a la hora de implementar, pero pueden conllevar también la pérdida de legilibilidad del código, por lo que deben ser usadas con mesura.

Modificador Abstract I

Es una palabra reservada que según dónde se aplique tiene una connotación.

- Clase.
  - Impide su instanciación, por lo que los constructores deberían ser *protected*.
- Método.
  - Delega su implementación,
  - Obliga a definir la clase como *abstract*.




```
public abstract class Vehiculo {
  public abstract double calcEficienciaConsumo();
  public abstract double calcDistanciaViaje();
}
```

Modificador Abstract II

```
public class Camion extends Vehiculo {
  public Camion(double cargaMax) {...}
  public double calcEficienciaConsumo() {
    // calcula el consumo de combustible de un camión
  }
  public double calcDistanciaViaje() {
    // calcula la distancia de este viaje por autopista
  }
}
```

```
public class Barcaza extends Vehiculo {
  public Barcaza(double cargaMax) {...}
  public double calcEficienciaConsumo() {
    // calcula el consumo de combustible de una barcaza
  }
  public double calcDistanciaViaje() {
    // calcula la distancia de este viaje por vías fluviales
  }
}
```

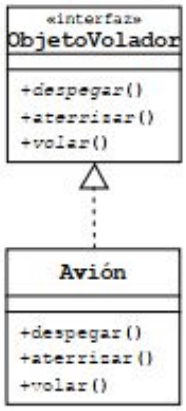
Interfaces I

Contrato entre el código *cliente* y la clase que implementa el servicio.

- Todos los atributos son constantes (aunque no se indique explicitamente).
  - *public static final*.
- Todos los métodos son públicos (*public*) y abstractos (*abstract*).
- Si una clase no implementa todos los métodos del interfaz debe ser *abstract*.
- Puede ser implementada por más de una clase, sin relación jerárquica.
- Una clase puede implementar más de una interface.
- Los nombres se utilizan como tipos de variables de referencia.
  - Se produce el enlace dinámico en tiempo de ejecución (polimorfismo / instanceof).

Permite compartir funcionalidades entre clases sin relación jerárquica.

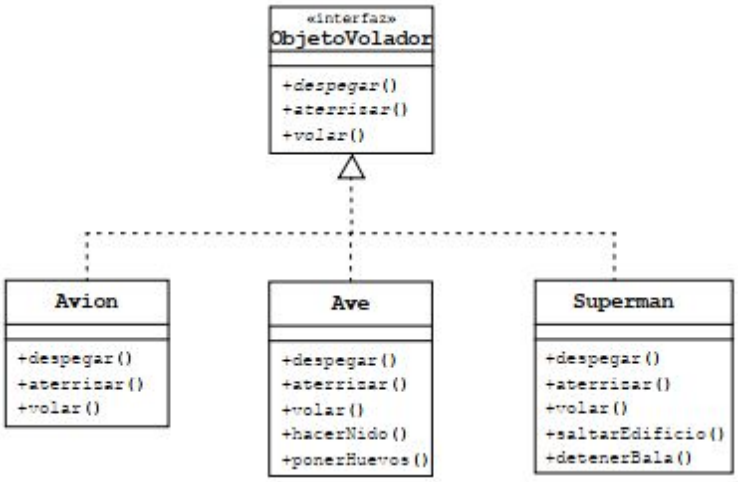
Interafaces II




```
public interface ObjetoVolador {
  public void despegar();
  public void aterrizar();
  public void volar();
}

public class Avion implements ObjetoVolador {
  public void despegar() {
    // acelerar hasta despegar
    // subir el tren de aterrizaje
  }
  public void aterrizar() {
    // bajar el tren de aterrizaje
    // decelerar y desplegar flaps hasta tocar tierra
    // frenar
  }
  public void volar() {
    // mantener los motores en marcha
  }
}
```

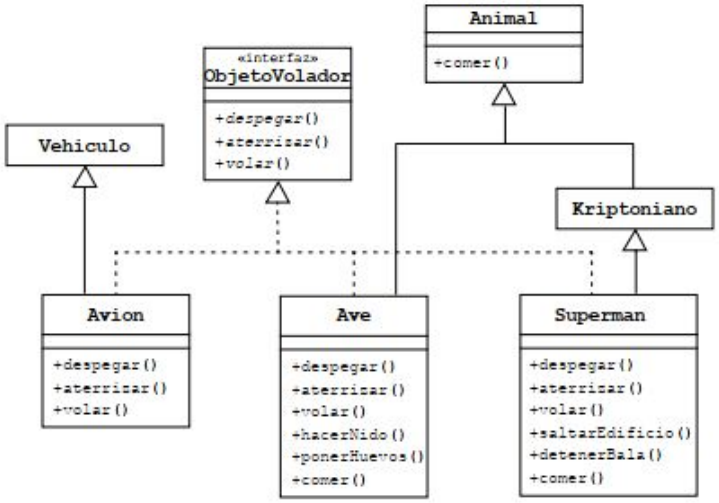
Interfaces III

Varias clases implementando el mismo interfaz:



Interfaces IV

No es herencia múltiple, dado que no es posible heredar dos implementaciones diferentes del mismo método



Interfaces V

```
public class Ave extends Animal implements ObjetoVolador {
  public void despegar()  { /* implementación de despegar   */ }
  public void aterrizar()     { /* implementación de aterrizar   */ }
  public void volar()      { /* implementación de volar  */ }
  public void hacerNido(){ /* comportamiento de nidificación   */ }
  public void ponerHuevos()  { /* implementación de puesta de huevos  */
}
  public void comer()  { /* sobrescritura de la acción de comer  */ }
}
```

- La cláusula *extends* debe aparecer antes que la cláusula *implements*.
- La clase *Ave*…
  - Puede sobrescribir los métodos de la clase *Animal*.
  - Debe implementar todos los métodos de la interfaz *ObjetoVolador*.
  - Puede suministrar sus propios métodos.

- La clase *Ave*, podría implementar algunos de los métodos de la interfaz *ObjetoVolador*, pero en ese caso, obligaría a declarar la clase como de tipo *abstract*.

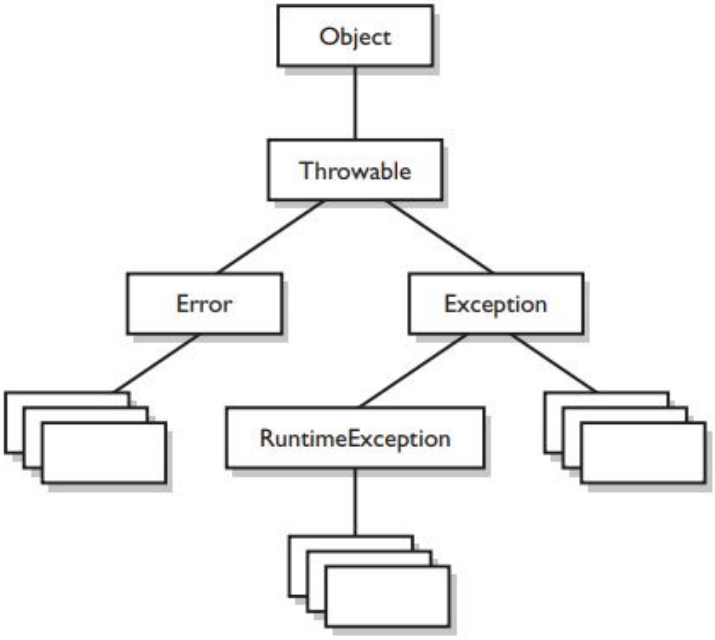
Excepciones

Excepciones I

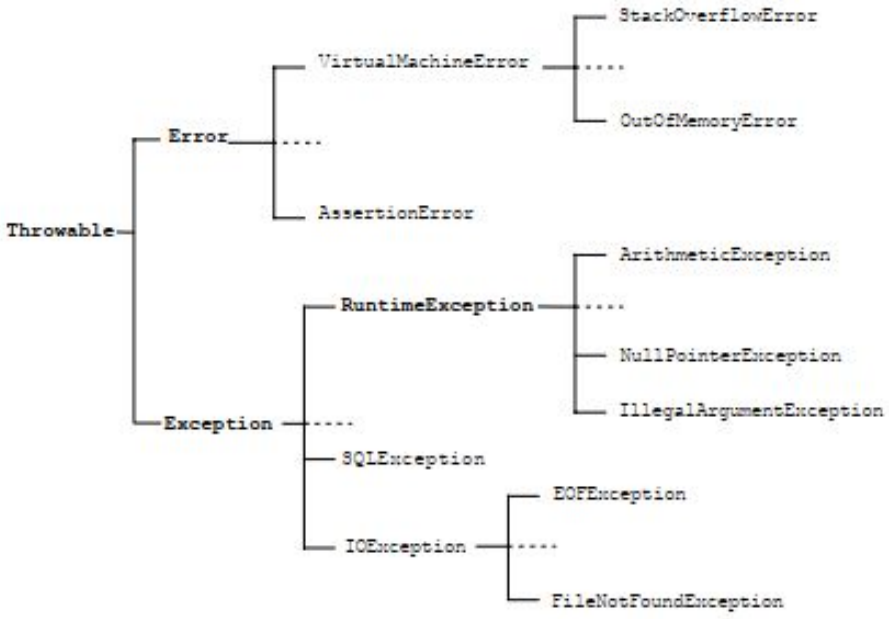
Sirven para describir lo que debe hacerse ante un error inesperado. Dos tipos:

- Comprobadas.
  - Se generan por causas externas que afectan al programa en ejecución.
    - Clase base, Exception.
  - Se espera que el programador las gestione en el código.
  - Ejemplo, solicitud de un fichero que no se encuentra.
- No comprobadas.
  - Consideradas como errores de código (RuntimeException) o difíciles para ser gestionadas por código (Error).
  - No se espera que el programador las gestione vía código.
  - Ejemplo, acceso a una posición de un array, más allá del final.
  - Errores. Son las más graves de las excepciones no comprobadas.
    - Se derivan de problemas del entorno y de las que es complicada una recuperación.
    - Ejemplo, agotamiento de la memoria RAM asignada a la JVM.

- Aunque ante una excepción no comprobada, no se espera que se intente una recuperación, se puede tratar de preservar la mayor parte del trabajo realizado.



- Error.
  - Problema grave de difícil o imposible recuperación.
  - Ejemplo: OutOfMemoryError.
- RuntimeException.
  - Problema de diseño o implementación.
  - Refleja situaciones que no deberían producirse.
  - Ejemplo: NullPointerException.
- Exception
  - Problema causado por efectos del entorno y puede gestionarse.
  - Ejemplo: FileNotFoundException.


Throwable
Error
VirtualMachineError
StackOverflowError
....
OutOfMemoryError
....
AssertionError
Exception
RuntimeException
ArithmeticException
....
NullPointerException
IllegalArgumentException
....
SQLException
IOException
EOFException
....
FileNotFoundException

Excepciones II

Cuando se genera una excepción, el método actual puede gestionarla o no, y supondría su reenvío al método llamante, y así sucesivamente, hasta como mucho el hilo padre (en este caso main), que de no gestionarla se interrumpe.

```
1   public class AddArguments {
2     public static void main(String args[]) {
3       int sum = 0;
4       for ( int i = 0; i < args.length; i++ ) {
5         sum += Integer.parseInt(args[i]);
6       }
7       System.out.println("Sum = " + sum);
8     }
9   }
```

```
java AddArguments 1 dos 3.0 4
Exception in thread "main" java.lang.NumberFormatException:
For input string: "dos"at
java.lang.NumberFormatException.forInputString(NumberFormat
Exception.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AddArguments.main(AddArguments.java:5)
```

Excepciones III

Mediante la sentencia try-catch es posible capturar excepciones de un tipo.

```
public class AddArguments2 {
  public static void main(String args[]) {
    try {
      int sum = 0;
      for ( int i = 0; i < args.length; i++ ) {
        sum += Integer.parseInt(args[i]);
      }
      System.out.println("Sum = " + sum);
    } catch (NumberFormatException nfe) {
      System.err.println("Uno de los argumentos de la linea de "
                         + "comandos no es un entero.");
    }
  }
}
```

```
java AddArguments2 1 dos 3.0 4
Uno de los argumentos de la línea de comandos no es un entero.
```


- Mediante la variable alimentada en el bloque catch, es posible llamar a algunos métodos de utilidad:
  - System.out.println("Pila: ");
  - nfe.printStackTrace();
  - System.out.println("Causa: " + nfe.getCause());
  - System.out.println("Mensaje: " + nfe.getMessage());
- También es posible capturar, y relanzar una excepción, porque queramos hacer algo en el método local, pero aún así, deseamos que el método llamante la reciba.

Excepciones IV

Es posible el uso del bloque try-catch en bloques más pequeños de código.

```
public class AddArguments3 {
  public static void main (String args[]) {
    int sum = 0;
    for ( int i = 0; i < args.length; i++ ) {
      try {
        sum += Integer.parseInt(args[i]);
      } catch (NumberFormatException nfe) {
        System.err.println("[" + args[i] + "] no es un entero"
                           + " y no se incluirá en la suma.");
      }
    }
    System.out.println("Sum = " + sum);
  }
}
```

```
java AddArguments3 1 dos 3.0 4
[dos] no es un entero y no se incluirá en la suma.
[3.0] no es un entero y no se incluirá en la suma.
Sum = 5
```


- Otros tipos de excepciones de tiempo de ejecución típicos:
  - ArithmeticException
    - int num=121/0;
  - ArrayIndexOutOfBoundsException
    - int misEnteros[] = new int[10];
    - System.out.println("posición final: " + misEnteros[10]);
  - NullPointerException
    - Perro p = null;
    - System.out.println(p.getChip());
  - ClassCastException
    - Object x = new Integer(0);
    - System.out.println((String)x);
  - IllegalArgumentException
    - Se suele lanzar por métodos de librerías ya implementadas, y también podemos hacerlo nosotros con las nuestras.

Excepciones V

Se pueden anidar bloques catch, que capturen distintos tipos de excepciones.

- El orden de aparición de los bloques influye, si hay jerarquía entre los tipos.

```
try {
  // código que podría generar una o varias excepciones

} catch (MiException e1) {
  // código que debe ejecutarse si se envía MiException

} catch (MiOtraException e2) {
  // código que debe ejecutarse si se envía MiOtraException

} catch (Exception e3) {
  // código que debe ejecutarse si se envía cualquier otra
  // excepción
}
```


- Si el bloque *catch* del tipo *Exception*, fuese colocado en primer lugar, supondría que el resto nunca se llegaría a ejecutar, por eso el compilador se quejaría, mostrando un error.

Excepciones VI

La cláusula *finally*, permite definir un bloque de código que siempre se ejecuta.

- Usado normalmente para cerrar las conexiones que pudiera haber abiertas a bbdd, sockets, red, ficheros etc.

```
InputStream input = null;
try {
    input = new FileInputStream("inputfile.txt");
}
finally {
    if (input != null) {
       try {
         in.close();
       }catch (IOException exp) {
           System.out.println(exp);
        }
    }
}
```


- Si el bloque englobado por la cláusula *try*, resultase contener un *return*, el contenido del bloque *finally*, también se ejecutaría, justo antes de ejecutar el return en sí mismo.

Excepciones VII

Las excepciones de tipo *Exception*, excluyendo las de tipo *RuntimeException*, deben ser controladas mediante el bloque *try-catch-finally*, o lanzadas…

```
public void leerFichero() throws IOException{
    InputStream input = new FileInputStream("ficheros/ListaCompra.txt");
    String contenidoFichero = new String(input.readAllBytes(), StandardCharsets.UTF_8);
    System.out.println(contenidoFichero);
    input.close();
}
```

También es posible capturar excepciones de manera genérica, como por ejemplo, *IOException*, abarca a las excepciones *EOFException* y *FileNotFoundException*.

Excepciones VIII

Es posible crear y luego lanzar nuevas excepciones.

```
public class ServerTimedOutException extends Exception {
  private int port;

  public ServerTimedOutException(String message, int port) {
    super(message);
    this.port = port;
  }

  public int getPort() {
    return port;
  }
}
```

```
public void connectMe(String serverName)
       throws ServerTimedOutException {
  boolean successful;
  int portToConnect = 80;

  successful = open(serverName, portToConnect);

  if ( ! successful ) {
    throw new ServerTimedOutException("Imposible conectar",
                                      portToConnect);
  }
}
```

Excepciones IX

Capturar la excepción lanzada, e incluso hacer un tratamiento parcial y relanzarla.

```
public void findServer() {
  try {
    connectMe(defaultServer);
  } catch (ServerTimedOutException e) {
    System.out.println("El servidor no responde, buscando servidor alternativo");

    try {
      connectMe(alternativeServer);
    } catch (ServerTimedOutException e1) {
      System.out.println("Error: " + e1.getMessage() +
                         " conectando con el puerto " + e1.getPort());
    }
  }
}
```

```
try {
  connectMe(defaultServer);
} catch (ServerTimedOutException e) {
  System.out.println("Error detectado y reenviado");
  throw e;
}
```

Aserciones

Aserciones I

Permiten verificar ciertos supuestos sobre la lógica de un programa.

- Usadas normalmente para verificar supuestos dentro de un método.
- Por defecto están deshabilitadas.
  - Para habilitarlas hay que ejecutar el comando java, con el parámetro -ea/enableassertions.
  - Lo que redunda en una mejora del rendimiento (es como si no estuvieran en el código).
- Admiten dos sintaxis…

```
assert <expresión_booleana> ;
assert <expresión_booleana> ; <expresión_detallada> ;
```

- Si la expresión booleana se evalúa a false, se lanza un *AssertionError*.
  - El Error no debería capturarse, y el programa finaliza, hace falta revisión desarrollador.

Aserciones II

Tipos de aserciones...

- Invariantes internas.
  - Evitan detectar un problema demasiado tarde.
- Invariantes del flujo de control.
  - Similares a las internas, pero enfocadas en el flujo y no tanto en el valor.

```
if (x > 0) { // hacer esto }
else { assert (x == 0); }
```

```
switch ( palo ) {
   case Palo.TREBOLES: // ...
     break;
   case Palo.DIAMANTES: // ...
     break;
   case Palo.CORAZONES: // ...
     break;
   case Palo.PICAS: // ...
     break;
  default: assert false : "Palo desconocido";
     break;
}
```


- Determinar el origen de un problema, en ocasiones puede resultar complicado. Mediante las aserciones, adelantamos el momento en el que algo que debería estar cumpliéndose, se controla, y con ello, se puede resolver, con el consiguiente ahorro de tiempo de revisión.

Aserciones III

- Postcondiciones e invariantes de clase.
  - Similares a las internas, pero enfocadas al estado de la variables al finalizar un método.
    - Invariante de clase, aplica a una propiedad estática (cantidad de Empleados)
  - El comportamiento externo o los parámetros de entrada, deben ser gestionados mediante excepciones, dado que así la prueba se realizará siempre (no dependiente de -ea).

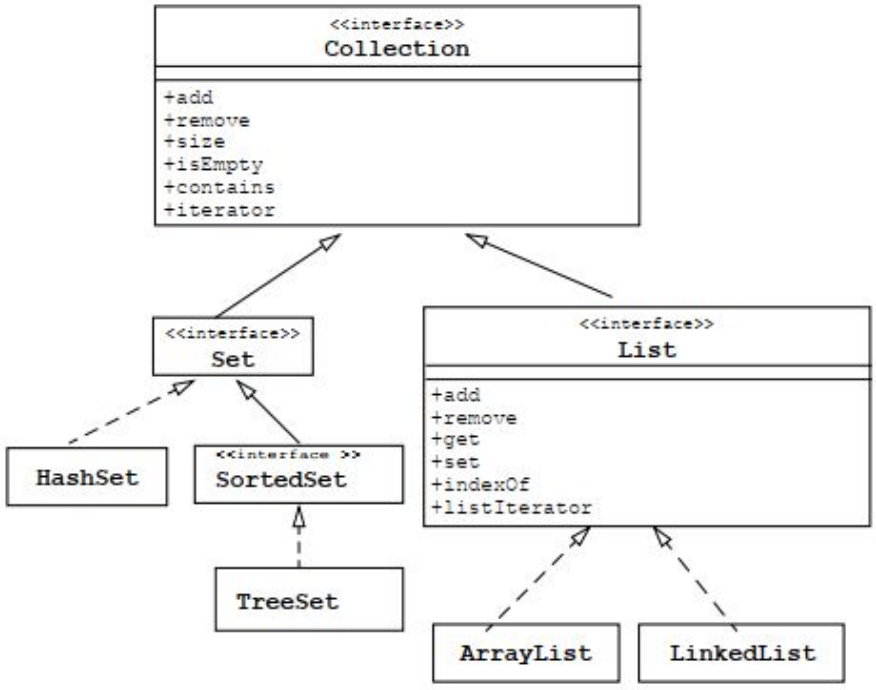
```
public Object pop() {
  int size = this.getElementCount();
  if (size == 0) {
    throw new RuntimeException("Intento de desapilar una pila vacía");
  }
  Object result = /* código para recuperar el elemento desapilado */
  // probar la postcondición
  assert (this.getElementCount() == size - 1);
  return result;
}
```

Colecciones y Genéricos

Colecciones

Permiten administrar un grupo de objetos, denominados elementos. Tipos...

- Collection. Interfaz principal para colecciones.
  - Set. Sin orden y no admite duplicados.
    - HashSet, implementa la interfaz *Set*.
    - SortedSet. Extiende la interfaz *Set*, incluyendo orden.
      - TreeSet, implementa la interfaz *SortedSet*.
  - List. Ordenada, y admite duplicados.
    - ArrayList y LinkedList, implementan la interfaz *List*.
- Las colecciones mantienen referencias a objetos de tipo *Object*.
  - Permite almacenar cualquier objeto, pero requiere hacer un casting a la clase correspondiente, para poder acceder a sus miembros particulares.
  - Con la introducción de los *Genéricos* en J2SE 1.5, se puede especificar el tipo almacenado.


<<interface>>
Collection
+add
+remove
+size
+isEmpty
+contains
+iterator
<<interface>>
Set
<<interface>>
List
+add
+remove
+get
+set
+indexOf
+listIterator
HashSet
<<interface >>
SortedSet
TreeSet
ArrayList
LinkedList

-

Set

- Si intentamos añadir un objeto ya existente, se ignora:

```
import java.util.*;
public class EjemploSet {
  public static void main(String[] args) {
   Set set = new HashSet();
   set.add("one");
   set.add("second");
   set.add("3rd");
   set.add(new Integer(4));
   set.add(new Float(5.0F));
   set.add("second");       // duplicado, no se agrega
   set.add(new Integer(4)); // duplicado, no se agrega
   System.out.println(set);
  }
}
```

- El orden de inserción, no se respeta: `[one, second, 5.0, 3rd, 4]`

103

- Cuando llevamos a cabo un añadido, se llama al método *equals* heredado de *Object*.
  - En el caso de una clase Perro, en la que no se ha implementado este método, por defecto, se retorna un valor construido, a partir del objeto que hay en memoria.
  - Las clases wrapper, implementan el método *equals*, en base al valor del entero que hay encapsulado, por eso los considera iguales, en el ejemplo de la transparencia, dado que en realidad, son dos variables de referencia distintas, con dos enteros encapsulados, cuyo valor es el mismo.
- Ejercicios.
  - Crear una clase 'Punto', con dos enteros, *x* e *y*. Crear a continuación dos instancias, con los mismos valores para ambos atributos, y asignarlos a una colección de tipo *Set*.

HashCode

- Es usado por las colecciones de tipo hash, para optimizar el almacenamiento de los elementos de manera más eficiente.
- Es necesario a la hora de comparar si dos elementos son iguales (*equals*).

Condition	Required	Not Required (But Allowed)
`x.equals(y) == true`	`x.hashCode() == y.hashCode()`	
`x.hashCode() == y.hashCode()`		`x.equals(y) == true`
`x.equals(y) == false`		`No hashCode() requirements`
`x.hashCode() != y.hashCode()`	`x.equals(y) == false`	

List

- Si intentamos añadir un objeto ya existente, se agrega:

```
import java.util.*;
public class EjemploList {
  public static void main(String[] args) {
   List list = new ArrayList();
   list.add("one");
   list.add("second");
   list.add("3rd");
   list.add(new Integer(4));
   list.add(new Float(5.0F));
   list.add("second");        // duplicado, se agrega
   list.add(new Integer(4));  // duplicado, se agrega
   System.out.println(list);
   }
}
```

- El orden de inserción, sí se respeta:

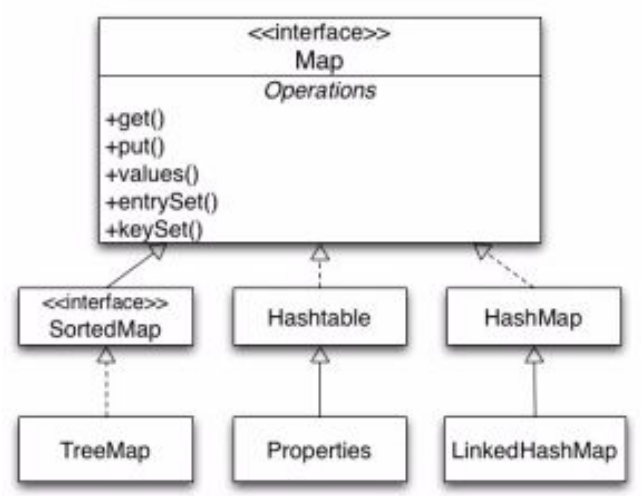
```
[one, second, 3rd, 4, 5.0, second, 4]
```

Map I

También denominados como arrays asociativos, gestionan duplas clave-valor.

- No admiten claves duplicadas.
- No implementa la interfaz *Collection*, porque representa asignaciones, no una colección de objetos.
- Proporciona 3 métodos para ver el contenido como colecciones…
  - entrySet.
    - Devuelve una variable *Set*, que contiene los pares formados por clave-valor.
  - keySet.
    - Devuelve una variable *Set*, con todas las claves del mapa.
  - values.
    - Devuelve una variable *Collection*, con todos los valores del mapa.

Map II



- Hashtable, a diferencia de HashMap, es synchronized, es decir, puede ser compartido por distintos hilos de ejecución, sin necesidad de tener que añadir código para controlar los accesos.

Map III

```
import java.util.*;
public class EjemploMap {
   public static void main(String args[]) {
      Map map = new HashMap();
      map.put("one","1st");
      map.put("second", new Integer(2));
      map.put("third","3rd");
      // Sobrescribe la asignación anterior
      map.put("third","III");
      // Devuelve el conjunto de las claves
      Set set1 = map.keySet();
      // Devuelve la vista Collection de los valores
      Collection collection = map.values();
      // Devuelve el conjunto de las asignaciones de claves a valores
      Set set2 = map.entrySet();
       System.out.println(set1 + "\n" + collection + "\n" + set2);
   }
}
```

```
[second, one, third]
[2, 1st, III]
[second=2, one=1st, third=III]
```

Retrocompatibilidad

En la versión 1.0 de J2SE, existían clases colección, que se han actualizado para interactuar con la nueva versión de la versión 1.5…

- La clase *Vector*, implementa la interfaz *List*.
- La clase *Stack*, amplía la clase *Vector*, para añadir las operaciones habituales de apilado: *push*, *pop* y *peek*.
- *Hashtable* implementa la interfaz *Map*.
  - La clase *Properties*, es una ampliación de *Hashtable*, que sólo utiliza valores de tipo String tanto para las claves, como para los valores.
- Estas colecciones tienen un método *elements*, que retorna un objeto *Enumeration* (1.0), similar, pero no compatible con la interfaz *Iterator* (1.2).
  - *hasMoreElements*, ha sustituido a *hasNext*.

- La diferencia de *peek*, con respecto a *pop*, es que el primero muestra el primer elemento de la pila sin eliminarlo de la misma.
- La principales diferencias entre Enumeration e Iterator…
  - Enumeration sólo está presente en colecciones *legacy* (antiguas), como son Vector y Hashtable.
  - Iterator, además de recorrer los elementos y leerlos, permite eliminarlos, cosa que Enumeration, no.

Ordenación - Comparable I

- Interfaz que pertenece a *java.lang* y que permite indicar a la JVM el orden lógico de los objetos de una clase.
  - La clase *Long*…
    - aplica una implementación cuya lógica es de tipo numérica.
  - La clase *String*…
    - aplica una implementación cuya lógica es de tipo alfabética.
  - La clase *Date*...
    - aplica una implementación cuya lógica es de tipo cronológica.
- El método *sort*, de la interfaz *Collections*, se basa en esta implementación (método *compareTo*) para ordenar los objetos que contiene.

Ordenación - Comparable II

```
class Estudiante implements Comparable {
  String Nombre, primerApellido;
  int IDestudiante=0;
  double Media=0,0;
  public Estudiante(String Nombre, String primerApellido, double Media) {
    if (Nombre == null || primerApellido == null || IDestudiante == 0
      || Media == 0,0) {throw new IllegalArgumentException();}
    this.Nombre = Nombre;
    this.primerApellido = primerApellido;
    this.Media = Media;
  }
  public int compareTo(Object o) {
    double f = Media-((Estudiante)o).Media;
    if (f == 0,0) return 0;
    else if (f<0,0) return -1;
    else return 1;
  }
}
```

El valor retornado significa…

- 0: iguales.
- -: antes de.
- +: después de.

Ordenación - Comparable III

```
import java.util.*;
public class ComparableTest {
  public static void main(String[] args) {
    TreeSet estudianteSet = new TreeSet();
    estudianteSet.add(new Estudiante("Miguel", "Herraiz", 4,0));
    estudianteSet.add(new Estudiante("Juan", "Lino", 2,8));
    estudianteSet.add(new Estudiante("Jaime", "Marcos", 3,6));
    estudianteSet.add(new Estudiante("Clara", "Genio", 2,3));

    Object[] estudianteArray = estudianteSet.toArray();
    Estudiante s;
    for(Object obj : estudianteArray){
      s = (Estudiante) obj;
      System.out.printf("Nombre = " + s.Nombre() + " ID = " +
       s.primerApellido() + " Media = " + s.Media() + "\n");
    }
  }
}
```

```
Nombre = Clara Genio ID = 104 Media = 2,3
Nombre = Juan Lino ID = 102 Media = 2,8
Nombre = Jaime Marcos ID = 103 Media = 3,6
Nombre = Miguel Herraiz ID = 101 Media = 4,0
```


- La clase *TreeSet*, es ordenada, por lo que llama al método *compareTo*, cada vez que lo precisa.
  - Si añadimos un *System.out.println* dentro del método, podremos ver en cuántas ocasiones es invocado.

Ordenación - Comparator I

- Interfaz que pertenece a *java.util* y que permite indicar a la JVM el orden lógico de los objetos mediante la implementación del método...
  - *int compare (Object o1, Object o2)*.
- Permite ordenar objetos que no implementen la interfaz *Comparable*.

```
import java.util.*;
public class CompNota implements Comparator {
  public int compare(Object o1, Object o2) {
    if (((Estudiante)o1).Media == ((Estudiante)o2).Media)
      return 0;
    else if (((Estudiante)o1).Media < ((Estudiante)o2).Media)
      return -1;
    else
      return 1;
   }
}
```

Ordenación - Comparator II

```
import java.util.*;
public class CompNombre implements Comparator {
  public int compare(Object o1, Object o2) {
    return
      (((Estudiante)o1).Nombre.compareTo(((Estudiante)o2).Nombre));
  }
}
```

```
import java.util.*;
public class ComparatorTest {
  public static void main(String[] args) {
    Comparator c = new CompNombre();
    TreeSet estudianteSet = new TreeSet(c);
```

```
Nombre = Jaime Marcos ID = 103 Media = 3,6
Nombre = Juan Lino ID = 102 Media = 2,8
Nombre = Clara Genio ID = 104 Media = 2,3
Nombre = Miguel Herraiz ID = 101 Media = 4,0
```

Genéricos

Permite garantizar la seguridad de los tipos almacenados en una colección/map.

- Supone no tener la necesidad de hacer conversiones de tipo explícitas.
- Impide la inserción de tipos de objetos que no coincidan con la declaración.
- Disponible desde la versión 1.5 de J2SE.
  - Pueden ser deshabilitados con la opción *-source 1.4* del comando *javac*.

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

```
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

```
list.add(0, 42);
int total = list.get(0);
```

Set Genérico

El compilador controla que los valores añadidos sean del tipo declarado.

```
import java.util.*;
public class EjemploSetGen {
 public static void main(String[] args) {
    Set<String> set = new HashSet<String>();
    set.add("one");
    set.add("second");
    set.add("3rd");
    // Esta línea genera un error de compilación
    set.add(new Integer(4));
    // Duplicado, no se agrega
    set.add("second");
    System.out.println(set);
  }
}
```

Map Genérico

```
import java.util.*;
public class RepositorioMapasJugadores {
  HashMap<String, String> jugadores;
  public RepositorioMapasJugadores() {
    jugadores = new HashMap<String, String> (); }
  public String get(String posicion) {
    String jugador = jugadores.get(posicion);
    return jugador; }
  public void put(String posicion, String nombre) {
    jugadores.put(posicion, nombre); }
  public static void main(String[] args) {
    RepositorioMapasJugadores equipodeensueño = new RepositorioMapasJugadores();
    equipodeensueño.put("delantero", "henry");
    equipodeensueño.put("lateral derecho", "ronaldo");
    equipodeensueño.put("portero", "cech");
    System.out.println("El delantero es " + equipodeensueño.get("delantero"));
    System.out.println("El lateral derecho es " +
                       equipodeensueño.get("lateral derecho"));
    System.out.println("El portero es " + equipodeensueño.get("portero"));
  }
}
```

```
El delantero es henry
El lateral derecho es ronaldo
El portero es cech
```

ArrayList Genérico

Categoría	Clase no genérica	Clase genérica
Declaración de clase	`public class ArrayList extends AbstractList implements List`	`public class ArrayList<E> extends AbstractList<E> implements List <E>`
Declaración de constructor	`public ArrayList (int capacity)`	`public ArrayList (int capacity)`
Declaración de método	`public void add(Object o)` `public Object get(int index)`	`public void add(E o)` `public E get(int index)`
Ejemplos de declaraciones de variables	`ArrayList list1;` `ArrayList list2;`	`ArrayList <String> a3;` `ArrayList <Date> a4;`
Ejemplos de declaraciones de instancias	`list1 = new ArrayList(10);` `list2 = new ArrayList(10);`	`a3= new ArrayList<String> (10);` `a4= new ArrayList<Date> (10);`

- Por convención se utiliza…
  - la letra ‘E’, para elementos de colecciones.
  - la letra ‘K’, para clave-valor en maps.
  - la letra ‘T’, para el resto de tipos.

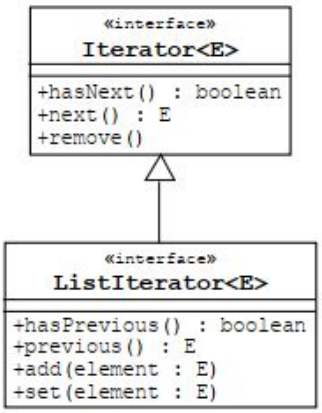
Iteradores

Permiten explorar/iterar, hacia delante, una colección.

- El orden de iteración depende del tipo de clase.

```
List list<Estudiante> = new ArrayList<Estudiante>();
// agregue algunos elementos
Iterator<Estudiante> elements = list.iterator();
while (elements.hasNext()) {
  System.out.println(elements.next());
}
```

- La clase List tiene un iterador (ListIterator) que permite explorar la lista hacia atrás e insertar/modificar elementos.



Bucle for mejorado

Elimina el uso de métodos iteradores distintos, y minimiza el número de veces que aparece la variable iterator.

```
public void deleteAll(Collection<NameList> c) {
  for (Iterator<NameList> i=c.iterator(); i.hasNext();)

    NameList nl = i.next();
    nl.deleteItem();
  }
}
```

```
public void deleteAll(Collection<NameList> c) {
  for (NameList nl: c) {
    nl.deleteItem();
  }
}
```

Conclusiones

- Posibles mejoras en el contenido y/o orden del curso (u de otra índole)
- Qué cosas sí me han aportado a nivel personal y/o profesional.
- Si hay margen de tiempo, proyecto final en el que diseñar e implementar una solución que aglutine, sino todas, la gran mayoría de los elementos aprendidos durante el curso.