MASTER EN MODELIZACIÓN MATEMÁTICA, ESTADÍSTICA Y COMPUTACIÓN
2011-2012

Curso: Bases de datos y programación orientada a objetos
Parte POO

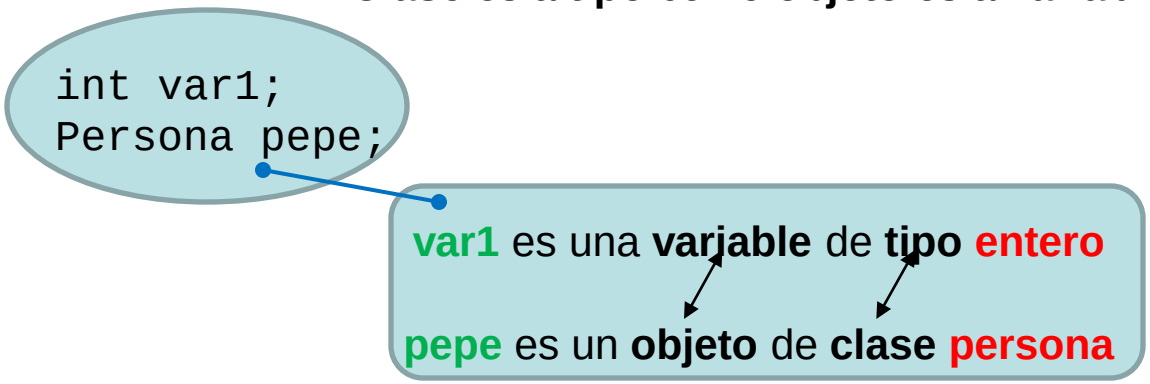
Programación con Orientación a Objetos.

german.bordel@ehu.es
http://gtts.ehu.es/German

Repetimos…

Una MÍNIMA idea de lo que es un “objeto” hasta que nos detengamos en ello…

Clase es a **tipo** como **objeto** es a **variable**



Una clase es un “tipo complejo”; una agrupación de variables (constantes), objetos, e incluso código que puede actuar sobre sus propios elementos u otros.

Un objeto es una cápsula (de memoria de ordenador) que tiene un “estado” (determinado por los valores de sus variables y el estado de sus objetos) así como un comportamiento (definido por el código que encierra).

Orientación a Objetos

Alan Kay (Smalltalk)

Todo es un objeto.
Un programa es un grupo de objetos diciendose unos a otros qué deben hacer mandándose mensajes.
Cada objeto tiene su propia memoria construida en base a otros objetos.
Todo objeto tiene un tipo.
Todos los objetos de un tipo particular pueden recibir los mismos mensajes.

En realidad no es algo diferente a lo que vinieran haciendo ya los buenos programadores: estructurar correctamente.

Esta estructuración encapsulaba datos con funciones que actuaban sobre los mismos de alguna manera (p.ej. en un mismo ".c" con su correspondiente ".h" en lenguaje C)

La conceptualización de esta estructuración como "objeto" (más o menos real o no)
- supone la vía a una modelización de los problemas a resolver mediante programas que ha resultado adecuada
- ha dado pie a conceptos asociados de gran ayuda (herencia, polimorfismo, etc)
- ha permitido descargar esfuerzo de desarrollo en sistemas automáticos.

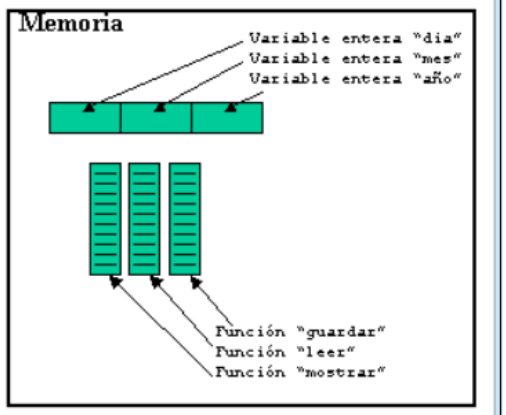
Ejemplos de clase: Coche, Fecha,…
Ejemplos de objeto: miCoche, hoy,…

Clase es a **tipo** como **objeto** es a **variable**

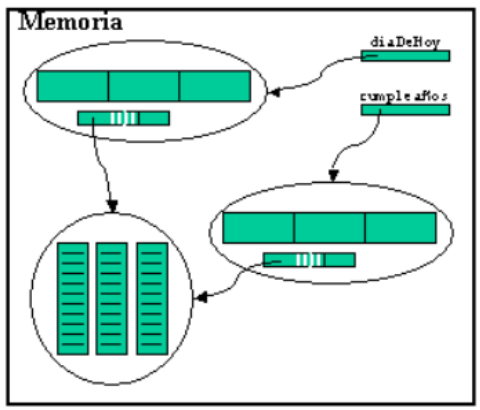
```
Coche miCoche;
Fecha hoy;
```

Un Coche cualquiera (hablamos de la clase por tanto)
- tendrá un estado compuesto por
  - objetos de otras clases: volante, asientos, etc.
  - variables y constantes: la velocidad, el identificador del color de pintura, etc.
- y tendrá un comportamiento
  - la capacidad de acelerar y frenar (una actuación sobre la velocidad)
  - la posibilidad de abrir y cerrar puertas (una actuación sobre los objetos puerta)
  - etc.

miCoche es un objeto de la clase Coche con color gris#444444, velocidad cero en este momento, etc.

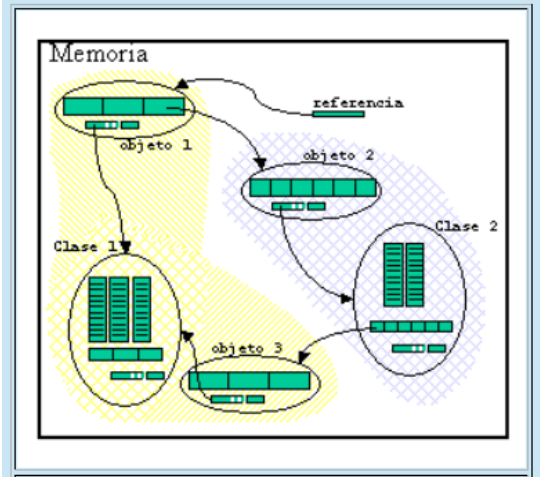


(a)

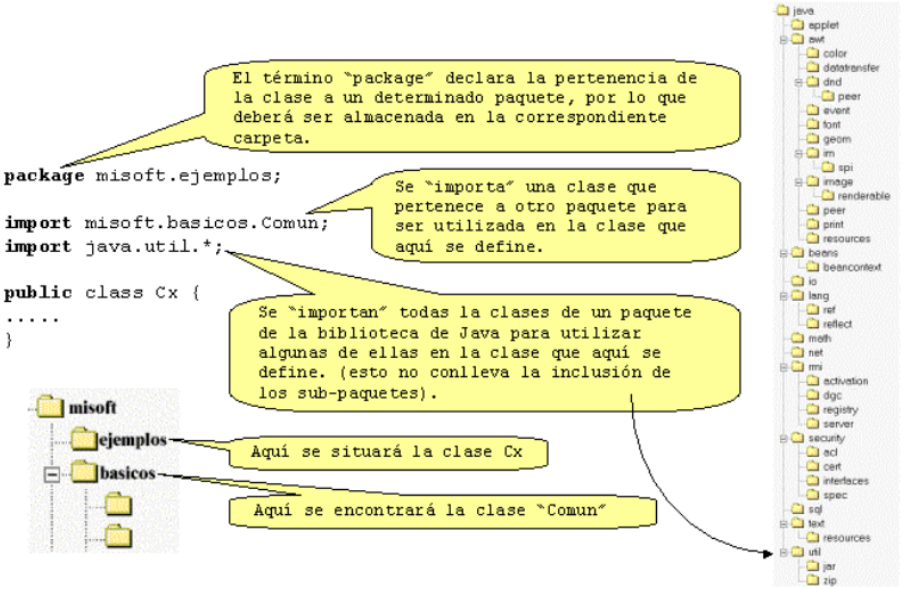


(b)

Un objeto es un espacio de memoria capaz de almacenar ciertos datos y un conjunto de funciones que pueden actuar sobre ellos. En (a) se representa un objeto que podemos denominar p.ej. `diaDeHoy`*. En la memoria puede haber varios objetos similares a este (son fechas) a los que se accede a traves de unas variables de referencia y que comparten las funciones (b). Los objetos tienen una referencia interna para tener localizadas estas funciones, así como otros elementos que les permiten funcionalidades que aún no se han visto en este curso.*



Un ejemplo de estructura en memoria relacionada con un objeto declarado en un programa. Al declarar el objeto se dispone de una referencia al mismo. Este objeto es de clase `"clase 1"` *y tiene entre sus variables internas otro objeto de clase* `"clase 2"`*, el objeto* `"objeto 2"`*. Este a su vez tiene su referencia a la clase a la que pertenece y dentro de ella hay una nueva referencia a otro objeto de clase* `"clase 1"`*.*



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

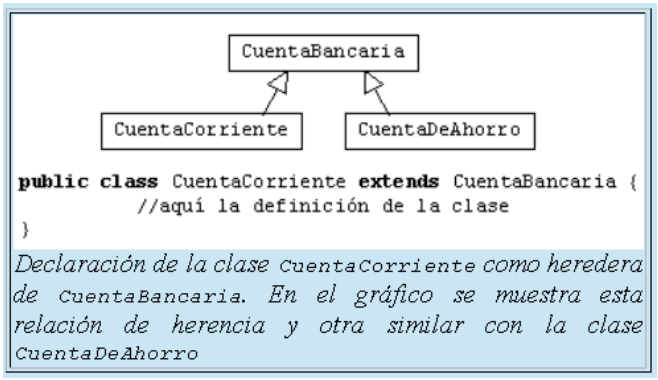
assert

enum

La "importación" es un mecanismo para "ahorrar" la escritura de los nombres completos de clases y objetos, limitándonos al nombre dentro del paquete. Cuando coinciden dos nombres, cada uno dentro de un paquete diferente, y se han importado ambos paquetes, será necesario referirse a cada elemento por su nombre completo.

```
[ámbito]
[abstract | final]
class <id.clase> [extends <id.clase>] [implements <id.interface>[,<id.interface>]*] {
        definición de la clase:
                Propiedades
                        variables primitivas
                        arrays de variables primitivas
                        objetos
                        arrays de objetos
                métodos
                        constructores
                        destructor
                        otros métodos
                                "getters"
                                "setters"
                                etc...
}
```

La definición de la clase se engloba entre llaves precediéndola por una declaración que al menos contiene la palabra reservada "class" seguida por el identificador de la clase. Opcionalmente esto puede ir acompañado de otros elementos que se irán viendo más adelante, y que incluyen, una declaración del ámbito de acceso, las características de ser "abstracta" o "final", así como el hecho de "extender" a otra clase y de "implementar" uno o varios interfaces.



Declaración de la clase `CuentaCorriente` como heredera de `CuentaBancaria`. En el gráfico se muestra esta relación de herencia y otra similar con la clase `CuentaDeAhorro`

Palabras reservadas en Java

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

assert

enum

La herencia permite definir clases que son "casos particulares" de otras. Heredan de estas otras sus características y añaden elementos específicos o alteran alguno de sus aspectos (sobrescriben o redefinen campos y métodos).

Por el hecho de "extender" a una clase, se "hereda" toda su definición (en este sentido es un mecanismo de ahorro de escritura de código).

Todas las clases están integradas en el árbol de herencia. La raíz de esta jerarquía es la clase "Object" (todos nuestros objetos son casos particulares del "objeto" genérico). Sintácticamente, no extender nada es equivalente a "`extend Object`".

La clase Object contiene determinado "material" que, consecuentemente, es compartido por todos los objetos java.

- java.lang.**Object**
  - java.awt.**Component**
    - java.awt.**Button**
    - java.awt.**Canvas**
    - java.awt.**Checkbox**
    - java.awt.**Choice**
    - java.awt.**Container**
      - java.awt.**Panel**
      - java.awt.**ScrollPane**
      - java.awt.**Window**
        - java.awt.**Dialog**
          - java.awt.**FileDialog**
        - java.awt.**Frame**
    - java.awt.**Label**
    - java.awt.**List**
    - java.awt.**Scrollbar**
    - java.awt.**TextComponent**
      - java.awt.**TextArea**
      - java.awt.**TextField**

Una pequeña zona de la jerarquía de clases

No hay que confundir la jerarquía de clases con la estructura de paquetes. Suele existir "cierta relación" subárbol-paquete ya que la proximidad de dos clases en estas estructuras implica que pueden tener "cierta relación", pero en todo caso son relaciones independientes

	Ámbito de acceso			
	clase	+paquete	+subclases	+todo
private	X			
package	X	X		
protected	X	X	X	
public	X	X	X	X

```
private int enteroPrivado=7;
character caracterPackage='X';
protected void metodoProtegido(){…}
public double metodoPublico() {…}
```

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

Los ámbitos de acceso son aplicables a las clases, y a sus componentes (campos y métodos), si bien en el caso de las clases, evidentemente sólo tienen sentido los ámbitos "`public`" y "`package`".

```
1- package packA;
2-
3- public class A {
4-     protected int i;
5-     // resto de definición de la clase
6- }
```

```
1- package packB;
2-
3- import packA.A;
4-
5- public class B extends A {
6-     void unMetodo(A a, B b) {
7-         a.i = 10;    // Ilegal. a pertenece a una clase de otro paquete.
8-         b.i = 10;    // Legal. b puede acceder a su miembro i heredado de la definición de A
9-     }
10-    // resto de la definición de la clase
11- }
```

El acceso "privado" debe "abrirse" a "package" si queremos que las extensiones de la clase no vean imposibilitada la actuación sobre elelemento

Refinamiento de lo visto anteriormente:
En realidad en un mismo fichero podemos definir más de una clase, pero sólo una podrá ser pública, y será esta la que determine el nombre del fichero.

```
1-  //
2-  // Aplicación ejemplo "HolaMundo"
3-  //
4-
5-  public class HolaMundo {
6-          public static void main(String[] args) {
7-              System.out.println("Hola, mundo");
8-          }
9-  }
```

Comentario: la importancia de "getters" y "setters"

El atributo "final" puede ser aplicado tanto a clases como a sus campos y métodos. Indica que una vez definido el elemento no puede volverse a definir:

- •para clases, no pueden tener subclases
- •para variables, no puede alterarse (constantes, pero si predefinición)
- •para métodos, no pueden ser redefinidos en una subclase.

“static” tiene dos utilidades

- •actuar como atributo aplicable a cualquier campo o método.
  - •para campos, residirán en la estructura de la clase
  - •para métodos, no están ligados a objetos (se invocan a través de la clase)

Palabras reservadas en Java

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

```
1- public class ClaseA {
2-         private static int cont=0;
3-         public static inc() {cont++;}
4-         public static dec() {cont--;}
5-         // el resto de la definición de la clase
6- }
```

```
1-         //Dentro de un bloque de código cualquiera...
2- ClaseA objetoA=new ClaseA();    //se genera el objeto de clase "ClaseA"
3- ClaseA.inc()                     //y se aumenta el contador asociado a la clase
```

- •inicializar la clase. (ver siguiente apartado)

```
1- public class ClaseA {
2-         private static int cont;
3-         public static inc() {cont++;}
4-         public static dec() {cont--;}
5-         static {cont=(otroObjeto.offset()>0)?otroObjeto.offset():0;}
6-         // el resto de la definición de la clase
7- }
```

```
1- //
2- // Aplicación ejemplo "HolaMundo"
3- //
4-
5- public class HolaMundo {
6-         public static void main(String[] args) {
7-                 System.out.println("Hola, mundo");
8-         }
9- }
```

```
1- MiClase miObjeto1 = new MiClase();
2- MiClase miObjeto2 = new MiClase("Hola");
3- MiClase miObjeto3 = new MiClase("Hola",10);
```

el operador **new** tiene "aspecto" de llamada a un método, con un identificador y una lista de parámetros entre paréntesis, y efectivamente esta es su función

Secuencia de acciones de la instanciación:

- Se reserva espacio en cantidad determinada por los campos definidos como propios de cada instancia (incluyendo los que hereda y los que implementan las capacidades del lenguaje)
- Se inicializan los campos que en la clase se han definido con inicialización.
- Se ejecuta un método denominado "constructor" (inicialización en tiempo de ejecución)

La primera vez que se invoca un objeto se dan estos mismos tres pasaos para la clase, donde la inicialización en tiempo de ejecución viene dada por el bloque “static” mencionado en el apartado anterior.

```
1- public class MiClase extends OtraClase {
2- private int n;
3-
4- public MiClase() {n=0;};
5- public MiClase(String s) {super(s); n=0;}
6- public MiClase(String s, int n) {super(s); this.n=n;}
7- // resto de la definición de la clase
8- }
```

Un constructor se distingue de un método en que:

- Su identificador coincide con el de la clase.
- No tiene tipo/clase de retorno en su definición (ni siquiera "void“)

Palabras reservadas en Java

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

Comportamiento de Java con los constructores:

- Si no definimos ninguno, existe uno sin parámetros y vacío.
- Si definimos al menos uno, el sistema no pone nada por defecto (OJO!).
- Si no se llama a “super” hay una llamada sin parámetros (super ha de ser la primera acción).

La contrapartida de los constructores es el destructor (heredado de Object y reescribible). Es llamado por el recolector de basuras.

protected void finalize()

(nota.- en realidad es un poco más complejo. Ejercicio: buscar la información)

Si
•planteamos un método en una clase "A" con el objeto de que siempre sea sobreescrito por toda subclase
•el conjunto de las subclases de "A" cubran toda la variedad posible de objetos de tipo "A“
Entonces
•deja de tener sentido la definición del método en la clase padre.

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

assert
enum

Pero si todas las subclases añaden la característica es algo común a todas y por tanto puede considerarse heredado.
Podemos declarar el método en la clase padre dejándolo sin definición (se preciso “avisar” con “abstract”

```
public abstract int reintegro(int cantidad);
```

Esto tiene la virtud de "obligar" a las subclases a implementar el método.

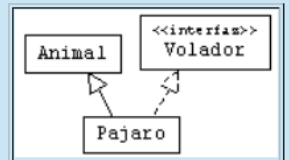
El hecho de que exista en una clase uno o varios métodos abstractos supone que su definición está incompleta y por tanto sólo tiene utilidad como clase padre de otras que definan totalmente sus elementos. Para indicar que esta circunstancia es "voluntaria" por parte del programador, debe incluirse el término "abstract" también en la declaración de la clase (p.ej. public abstract class CuentaBancaria {...}).

```
1- public abstract class CuentaBancaria {
2-         public abstract void reintegro(int cantidad);
3-         // resto de la definición de la clase
4- }
```

```
1- public class CuentaCorriente extends CuentaBancaria {
2-         public void reintegro(int cantidad) {
3-                 //definición del método reintegro
4-         }
5-         // resto de la definición de la clase
6- }
```

Los interfaces implementan la idea de obligación introducida por la abstracción de un modo más ámplio. Un interfaz contiene declaraciones de métodos abstractos únicamente, de modo que es lo que en ocasiones se entiende como un "contrato" que obliga a un cierto cumplimiento a las clases que lo implementan (las clases se "heredan", los interfaces se "implementan").

Son la alternativa a la herencia múltiple de otros lenguajes orientados a objetos



La clase `Pajaro` hereda de `Animal` e implementa el interfaz `Volador`.

Palabras reservadas en Java

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

Sólo pueden declararse un tipo más de elementos en un interfaz: constantes, es decir campos con el atributo final. Un interfaz puede implementar a su vez otros de manera que puede llegar a ser la unión de varios y/o una ampliación de ellos. Esto hace que la relación establecida entre interfaces no se limite a un árbol, sino que sea un grafo de tipo jerarquía con herencia múltiple.

```
1- public interfaz Volador{
2-         public void despegue();
3-         public void aterrizaje();
4- }
```

```
1- public class Pajaro extends Animal implements Volador{
2-         public void despegue(){
3-                 //definición del método
4-         }
5-         public void aterrizaje() {
6-                 //definición del método
7-         }
8-         //definición del resto de la clase
9- }
```

El polimorfismo es la capacidad de considerar a un objeto según diferentes "formas" dependiendo de la ocasión. Todo objeto de una determinada clase puede ser considerado como objeto de sus clases ascendientes o como objeto de una "clase identificada por uno de los interfaces que implementa".

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

assert

enum

```
1- private  Pajaro  gorrion=new Pajaro();
2- private  Animal  gorrion_A=gorrion;
3- private  Volador gorrion_V=gorrion;
```

```
1- private  Animal  gorrion_A = new Pajaro();
2- private  Pajaro  gorrion = (Pájaro) gorrion_A;
```

Un detalle a tener en cuenta es que aunque se acceda a través de una referencia a una clase más general que la del objeto real, en caso de invocar a un método que se encuentre sobreescrito en la clase más específica, será el código específico el que se ejecute.

Esto impone una restricción a la hora de sobreescribir métodos en lo que se refiere a los ámbitos de acceso: una sobreescritura de un método no puede restringir el ámbito de acceso (p.ej sobreescribir como "privado" un método que era "público" en la clase padre) ya que en caso de acceso a través de una referencia de la clase padre se estaría permitiendo un acceso ilegal. (hay aún otra limitación en relación con el proceso de errores que se verá en el capítulo correspondiente).

El operador instanceof

```
gorrion_A instanceof Pajaro --> [true]
gorrion_A instanceof Animal --> [true]
gorrion_A instanceof Object --> [true]
gorrion_A instanceof String --> [false]
```

Como cualquier otro lenguaje algorítmico, Java tiene la posibilidad de manejar Arrays, es decir estructuras que almacenan de forma contigua un determinado número de elementos del mismo tipo o clase. Nótese que en los lenguajes clásicos existe otra estructura capaz de almacenar un grupo de elementos heterogéneos (denominada "struct" en C o "record" en pascal) y que no existe en Java ya que es sustituida y ampliada con el concepto de clase.

Los arrays en Java no son elementos propios de la orientación a objeto, pero se les ha dotado de alguna característica que los asemeja puntualmente.

```
1- public   int       indices1[];    //array de enteros declarado con "estilo C"
2- public   int[]     indices2=null; //array de enteros
3- private  Animal    zoo1[];        //array de objetos "Animal" declarado con "estilo C"
4- private  Animal[]  zoo2=null;     //array de objetos "Animal"
```

Declaración estilo C vs. estilo Java

```
1- public  int[]     indices=new int[10];  //array de 10 enteros
2- private Animal[]  zoo=new Animal[20];   //array de 20 objetos "Animal"
```

Tamaño predeterminado

```
1-  public   int[]     indices=new int[10];
2-  private  Animal[]  zoo=new Animal[20];
3-
4-  indices[5]=7;
5-  zoo[12]=new Pajaro();
6-  zoo[17]=new Roedor();
```

Asignación

```
1- public int[] indices;
2- // otras definiciones y bloque de sentencias
3- indices=new int[2*numParejas()];
```

Tamaño determinado en ejecución

```
1- public   int[]     indices={3,2,5,4,7,1,9,8,6,0};
2- private  Animal[]  zoo={null,null,new Pajaro(),null,new Roedor()};
```

Asignación en bloque

```
1- public int[][] indices1;
2- public int[][] indices2=new int[10][];
3- public int[][] indices3=new int[10][3];
4- public int[][] indices4={{3,2,5},{4,7},{1,9,8,6,0}};
5- // otras definiciones y bloque de sentencias
6- indices2[5]=new int[3];
7- indices4[2][1]=7;
```

multidimensionales

```
1-  //
2-  // Aplicación ejemplo "EchoParameters"
3-  //
4-
5-  public class EchoParameters {
6-         public static void main(String[] args) {
7-                  for (int i=0; i<args.length; i++)
8-                           System.out.println(args[i]);
9-         }
10- }
```

El campo “length”

```
1-  //
2-  // Aplicación ejemplo "HolaMundo"
3-  //
4-
5-  public class HolaMundo {
6-         public static void main(String[] args) {
7-                  System.out.println("Hola, mundo");
8-         }
9-  }
```

Estudiaremos superficialmente este tema con un ejemplo.

En Java disponemos, además de clases e interfaces, de "enumeraciones", que son clases de las que pueden instanciarse un conjunto predefinido de objetos.

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

```
import java.util.*;

public class Card {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
        SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private final Rank rank;
    private final Suit suit;
    private Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public Rank rank() { return rank; }
    public Suit suit() { return suit; }
    public String toString() { return rank + " of " + suit; }

    private static final List<Card> protoDeck = new ArrayList<Card>();

    // Initialize prototype deck
    static {
        for (Suit suit : Suit.values())
            for (Rank rank : Rank.values())
                protoDeck.add(new Card(rank, suit));
    }

    public static ArrayList<Card> newDeck() {
        return new ArrayList<Card>(protoDeck); // Return copy of prototype deck
    }
}
```

Comentario:
clases internas y clases anónimas

(el ejemplo muestra las enumeraciones como meras listas de identificadores, pero son realmente objetos y su definición puede "complicarse" considerablemente. Pueden estudiarse en la documentación de Sun.)

Orientación a Objetos… gestión de errores

En la ejecución de sentencias de un programa, pueden producirse situaciones diferentes de las deseadas:

- la imposibilidad de abrir un fichero por diversos motivos,
- la imposibilidad de efectuar una división porque la expresión denominador da cero como resultado de su evaluación,
- la imposibilidad de acceder a un determinado objeto por utilizar erróneamente una referencia nula,
- etc.

Es necesario un mecanismo de detección y control de modo que para cada situación no deseada se actúe en consecuencia.

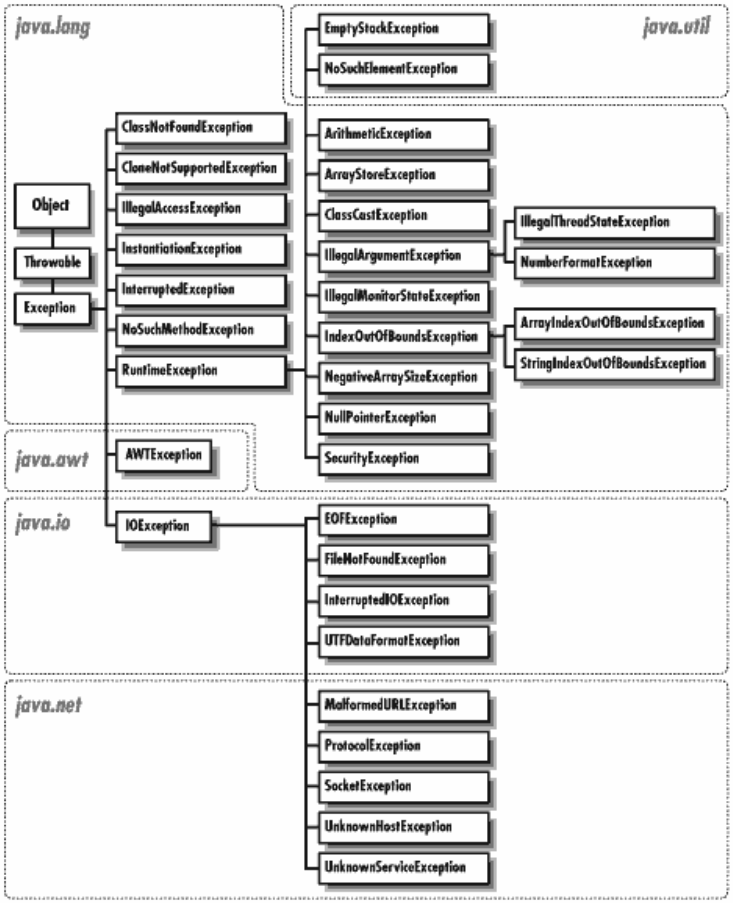
En los lenguajes de programación "clásicos" no se suele disponer de otro mecanismo que la sentencia condicional, que permite establecer dos caminos de ejecución, uno para la situación "deseada", y otro para la "problemática". De este modo ambos caminos forman parte del algoritmo, que ya no solo es el algoritmo de ejecución de una determinada tarea, sino que es esto más el control de toda la combinatoria de posibles situaciones "problemáticas".

Java permite separar el algoritmo de ejecución de una tarea de los algoritmos de solución de las diversas situaciones "problemáticas":

> Cuando se da una de dichas situaciones, la ejecución del algoritmo "principal" (camino de éxito) se aborta y se trasfiere a otro punto donde se encuentra el algoritmo de gestión del problema concreto producido, pasándole información de lo sucedido encapsulada en un objeto.

Terminología: los objetos con información del problema son "arrojados" en el momento de la excepción y "capturados" por el bloque de código que se hace cargo de la situación.

Todas las clases correspondientes se encuentran en un subárbol de la jerarquía que comienza con "`Throwable`" y que tiene a su vez dos subárboles cuyas raíces son "`Error`" y "`Exception`".



La separación de los bloques de código para una ejecución correcta y sus posibles situaciones excepcionales se lleva a cabo mediante una sentencia que se dejo sin ver en su momento para hacerlo ahora:

TRY-CATCH-FINALLY

```
try sentencia [catch (clase_arrojable id) sentencia]+ finally sentencia
```

Analizaremos el siguiente ejemplo:

```
1-  FileInputStream f=null;
2-  int d;
3-  try {
4-      f=new FileInputStream("pepe");
5-      d=f.read();
6-      // aqui el resto de acciones que se quieran realizar con f
7-  } catch (FileNotFoundException e) {
8-      // aquí el código a ejecutar en caso de que no existiera el fichero "pepe"
9-  } catch (IOException e) {
10-     // aquí el código a ejecutar en caso de que no se haya podido obtener f por otro motivo
11- } finally {
12-     if (f!=null) f.close();
13-     // aquí otro código que deba ejecutarse en todo caso
14- }
```

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

(pinchando en las clases se obtiene su documentación)

`FileInputStream` es capaz de abrir un fichero para realizar lecturas de datos. "f" es una referencia que utilizaremos para un objeto de este tipo.

1. En las lineas 3 a 7 se encuentra un bloque de código tras el término **`try`** que recoge la ejecución que queremos realizar con nuestro fichero (abrir uno de nombre "pepe", leer un entero sobre la variable d, etc.)

2.a. En el momento de instanciar el objeto "f" (línea 3) puede producirse un error de clase `FileNotFoundException`, de modo que el bloque de las líneas 7 y 8, recibe un objeto de esta clase en caso de producirse el problema y actua en consecuencia (nótese que el código de este **`catch`** puede hacer uso del objeto recibido o no).

2.b Igualmente, en el método `read()` pueden producirse indeterminados errores de entrada/salida identificados con un objeto de clase `IOException` (p.ej. faltas de permisos de acceso, ausencia de datos sin fin de fichero, etc). Las acciones a ejecutar en tal caso se llevan a cabo en el bloque 9-10.

3. Por último, el fragmento **`finally`** de las líneas 11-14 se ocupará de terminar con todo aquello que pueda haber quedado en situación inconclusa, como por ejemplo de cerrar el fichero en caso de que se hubiese llegado a abrir.

Detalles:

- El bloque finally se ejecuta SIEMPRE, incluso si nos encontramos dentro de un método y hay un `return` en los bloques `try/catch` (la ÚNICA posibilidad de que no sea así es que la aplicación termine en uno de los bloques mediante `System.exit(.)`)
- Si tenemos varios `catch` asociados a clases de una misma línea de jerarquía, es preciso situarlos ordenadamente del más específico al más general.

Una vez que se produce la excepción, se aborta la ejecución del código, de modo que si se encontraba dentro de un bloque **`try`** pasará a ejecutarse el correspondiente **`catch`**. De no ser así, terminará el método y este "arrojará" el objeto "hacia" el método le que llamó.

Cuando un método deja "escapar" objetos arrojables es preciso que se indique en su declaración, para lo que basta con añadir a su prototipo, después de los parámetros la palabra reservada **`throws`** y la lista, separada con comas, de todas aquellas clases de las que puede lanzar objetos. P.ej:

Palabras reservadas en Java

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

assert

enum

```
1- public void desahogada() throws IOException, NumberFormatException, ArithmeticException {
2-   //cuerpo del método, donde se producen y no se atienden las excepciones indicadas.
3- }
```

Atención!!: Ante las situaciones en que puedan producirse excepciones (o errores) tendremos que plantearnos cuidadosamente qué será lo adecuado, si atenderlas en un bloque **`catch`** o dejarlas salir del método declarándolo explícitamente.

Sobreescritura de métodos que arrojan excepciones o errores

Puede sobreescribirse un método que en la clase padre arrojaba excepciones variando el comportamiento al respecto, pero sólo en el sentido de hacerla más restrictiva, es decir tratando internamente ciertas excepciones que el método de la clase padre no trataba

```
1- import java.io.*;
2- import java.text.*;
3-
4- public class A {
5-
6-     public void metodo() throws IOException, ParseException {
7-         // ...
8-         int i=f.read(); //IOException no atendida
9-         //...
10-        Double d=df.parse(s); //ParseException no atendida
11-        //...
12-    }
13-
14-    //....
15- }
```

```
1- import java.io.*;
2-
3- public class B extends A{
4-
5-     public void metodo() throws FileNotFoundException {
6-         // ...
7-         FileInputStream f=new FileInputStream();
8-         //...
9-     }
10-
11-    //....
12- }
```

Nota: El compilador se encarga bastante de que hagamos bien la gestión de errores ya que avisa posibles excepciones no consideradas, de ordenaciones incorrectas de los catch, nos obliga a tomar una determinación de atender o declarar que una excepción sale del método… No obstante permite que algunas excepciones no sean tratadas ya que entiende que obedecen a situaciones que el programador ha podido evitar (por ejemplo `ArithmeticException` puede darse en una división de enteros cuando el denominador es cero, pero puede que el algoritmo no permita esta situación, por lo que no es razonable obligar a "controlarla").

Nuestros programas pueden generar excepciones:

Detectada la situación excepcional puede activarse el mecanismo instanciando un objeto de la subclase "arrojable" que se considere adecuada, y utilizando "`throw`":

```
throw objeto_arrojable
```

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

assert

enum

Es habitual no añadir información alguna ya que la identidad de la clase en sí misma puede ser suficiente información.

```
1- if (condición_de_error) throw new IOException();
```

En ocasiones es interesante incluir un texto informativo:

```
1- if (condición_de_error) throw new IOException("Rebasado límite autorizado en salida");
```

Pero en caso de querer introducir información específica deberemos generar nuestras propias instrucciones

Definir una nueva excepción o error no es otra cosa que definir una clase que herede de `Exception` o `Error` respectivamente o de alguna de sus clases descendientes si se considera preciso.

Típicamente esta definición suele ser meramente "de trámite" ya que su comportamiento puede ser suficiente con el heredado, de modo que solo se reescriben los constructores:

```
1- public class MyException extends Exception {
2- public MyException(){super();}
3- public MyException(String s){super(s);}
4- }
```

Naturalmente nada se opone a que hagamos una definición de la excepción "a nuestra medida" añadiéndole las características que deseemos.

Si se produce una excepción que aborta la ejecución de un programa (el compilador no nos ha obligado a considerarla y no lo hemos hecho), se nos muestra la "traza de ejecución"

```
java.lang.NullPointerException
        org.apache.jsp.MiCursoJava.tema5.error_jsp._jspService(error_jsp.java:56)
        org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:97)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:332)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```

Si la atendemos y evitamos que la aplicación se aborte, podemos mostrar esta misma información valiéndonos del método "printStackTrace()" sobre el objeto arrojado.

```
catch (IOException e) {
    e.printStackTrace();
}
```

Hay excepciones que contienen excepciones:

excepción

```
org.apache.jasper.JasperException
        org.apache.jasper.servlet.JspServletWrapper.handleJspException(JspServletWrapper.java:510)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:393)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```

causa raíz

```
java.lang.NullPointerException
        org.apache.jsp.MiCursoJava.tema5.error_jsp._jspService(error_jsp.java:56)
        org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:97)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
        org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:332)
        org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:314)
        org.apache.jasper.servlet.JspServlet.service(JspServlet.java:264)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```