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,…

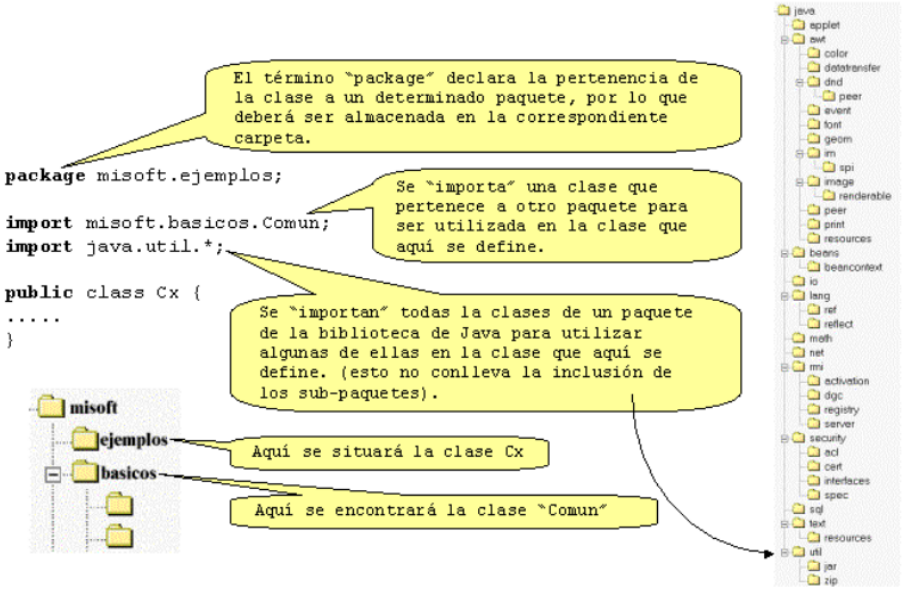
Clase es a **tipo** como **objeto** es a **variable**

```
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.

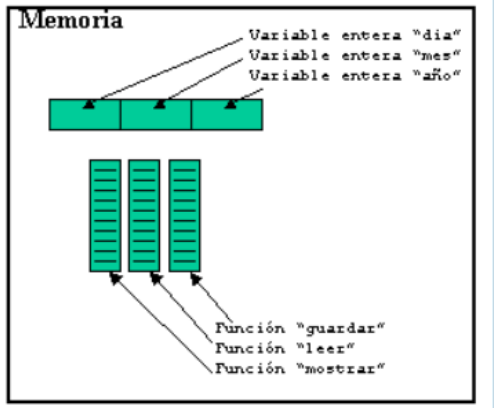


Palabras reservadas en Java				
abstract	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

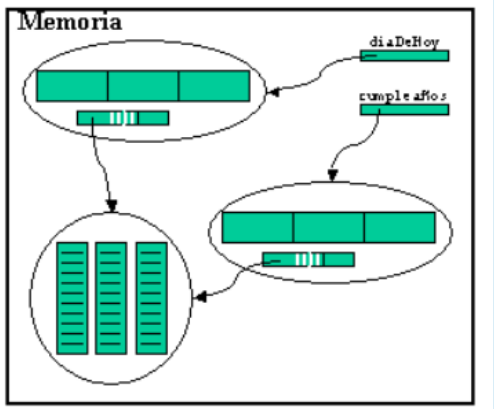
La "importación" es un mecanismo para "ahorrar" la escritura de los nombres completos de clases y objetos, limitándonos al nombre dentro del paquete. Cuando coinciden dos nombres, cada uno dentro de un paquete diferente, y se han importado ambos paquetes, será necesario referirse a cada elemento por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

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

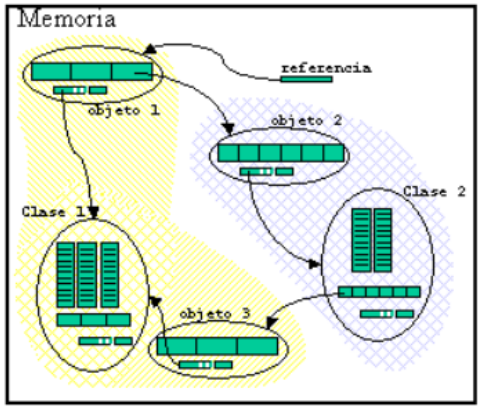


(a)

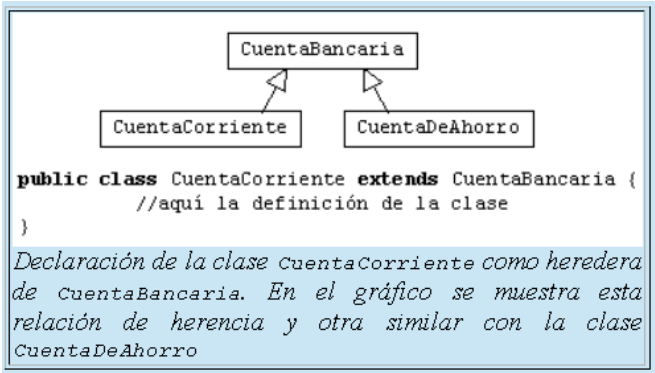


(b)

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"`*.*




```
public class CuentaCorriente extends CuentaBancaria {
        //aquí la definición de la clase
}
```

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	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

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.

Observemos detenidamente esta información en una página de documentación.

javax.swing.text

Class JTextComponent

java.lang.Object
  java.awt.Component
    java.awt.Container
      javax.swing.JComponent
        javax.swing.text.JTextComponent

~~**All Implemented Interfaces:**~~

~~ImageObserver, MenuContainer, Serializable, Accessible, Scrollable~~

Direct Known Subclasses:

JEditorPane, JTextArea, JTextField

Comentario: la sobreescritura de elementos heredados (principalmente métodos)

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

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.

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 (es preciso “avisar” con “abstract”

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

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

Palabras reservadas en Java				
abstract	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

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 {...}).

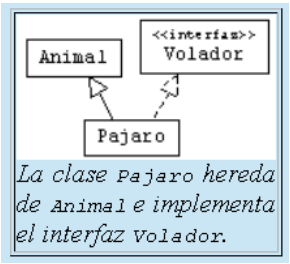
```
1- public abstract class CuentaBancaria {
2-         public abstract void reintegro(int cantidad);
3-         // resto de la definición de la clase
4- }
```

```
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 amplio. 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

Palabras reservadas en Java				
abstract	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while




```
1- public interface 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- }
```

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.

(*) Sólo pueden declararse un tipo más de elementos en un interfaz: constantes, es decir campos con el atributo final, que veremos más adelante. (bueno, en realidad desde la versión 8 de Java puede incluirse “algo más”, pero es una de tantas cosas por las que nos limitamos a Java 7.)

Volvamos de nuevo a observar detenidamente esta información en una página de documentación.

javax.swing.text

Class JTextComponent

java.lang.Object
  java.awt.Component
    java.awt.Container
      javax.swing.JComponent
        javax.swing.text.JTextComponent

All Implemented Interfaces:

ImageObserver, MenuContainer, Serializable, Accessible, Scrollable

Direct Known Subclasses:

JEditorPane, JTextArea, JTextField

javax.swing

Interface Scrollable

All Known Implementing Classes:

DefaultTreeCellEditor.DefaultTextField, JEditorPane, JFormattedTextField, JLayer, JList, JPasswordField, JTable, JTextArea, JTextComponent, JTextField, JTextPane, JTree

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	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

```
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]
```