4.- Elementos relacionados con la Orientación a Objeto

4.1 - OBJETOS Y CLASES

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)
definirá un estado compuesto por
- objetos de otras clases: volante, asientos, etc.
- variables y constantes: la velocidad, el identificador del color de pintura, etc.

y definirá 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.

4.- Elementos relacionados con la Orientación a Objeto

4.2 - ESTRUCTURA DE LA DEFINICIÓN DE UNA CLASE

http://gtts.ehu.es/German

```
package misoft.ejemplos;

import misoft.basicos.Comun;
import java.util.*;

public class Cx {
.....
}
```

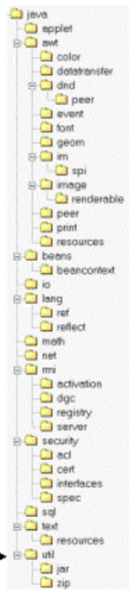
El término "package" declara la pertenencia de la clase a un determinado paquete, por lo que deberá ser almacenada en la correspondiente carpeta.

Se "importa" una clase que pertenece a otro paquete para ser utilizada en la clase que aquí se define.

Se "importan" todas la clases de un paquete de la biblioteca de Java para utilizar algunas de ellas en la clase que aquí se define. (esto no conlleva la inclusión de los sub-paquetes).

Aquí se situará la clase Cx

Aquí se encontrará la clase "Comun"

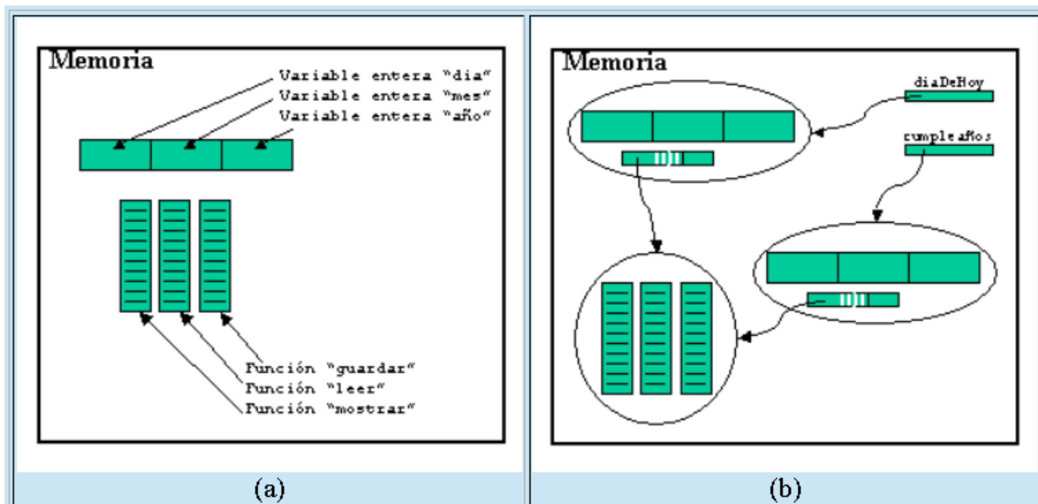


La "importación" es un mecanismo para "ahorrar" la escritura de los nombres completos de clases y objetos, limitándonos al nombre dentro del paquete. Cuando coinciden dos nombres, cada uno dentro de un paquete diferente, y se han importado ambos paquetes, será necesario referirse a cada elemento por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

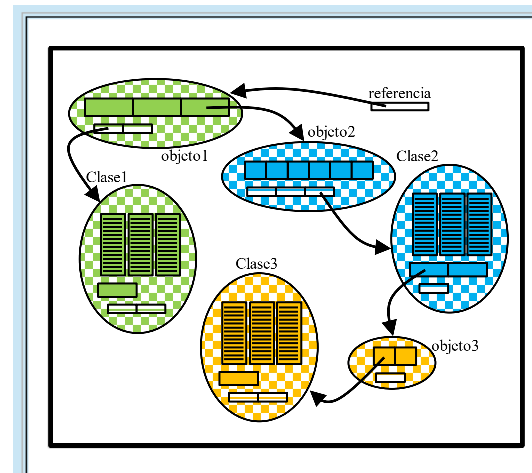
Palabras reservadas en Java				
abstract	assert(1.4)	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum(5.0)	extends	final	finally	float
for	goto*	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp(1.2)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

```
[ámbito]
[abstract | final]
class <id_clase> [extends <id_clase>] [implements <id_interface>[,<id_interface>]*] {

    definición de la clase:

      PROPIEDADES                          MÉTODOS
        variables primitivas                 constructores — "especial"
        arrays de variables primitivas       destructor        "normales"
        objetos                              otros métodos     Siguen
        arrays de objetos                      "getters"       una misma
                                               "setters"       sintaxis
                                               etc...
}
```

La definición de la clase se engloba entre llaves precedidas por una declaración que al menos contiene la palabra reservada 'class' seguida del 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 'implementar' uno o varios interfaces.



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.*

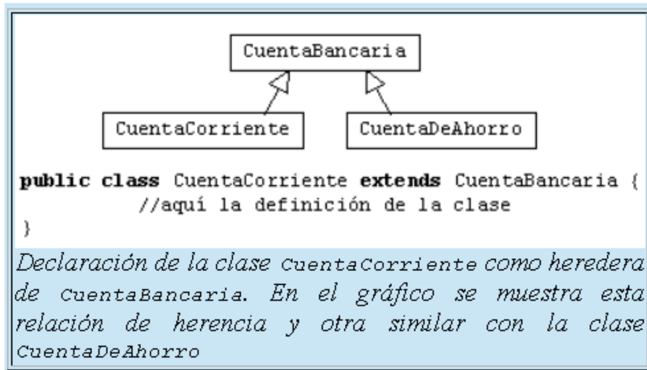


Un ejemplo de estructura en memoria relacionada con un objeto declarado en un programa. Al declarar el objeto se dispone de una referencia al mismo. Este objeto es de clase `"clase 1"` *y tiene entre sus variables internas otro objeto de clase* `"clase 2"`*, el objeto* `"objeto 2"`*. Este a su vez tiene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

4.- Elementos relacionados con la Orientación a Objeto

4.3 - HERENCIA

http://gtts.ehu.es/German



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 "`extends Object`".

La clase `Object` contiene determinado "material" que, consecuentemente, es compartido por todos los objetos java. (estudiaremos la clase `Object` más adelante)

Observemos detenidamente la información sobre herencia 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
```

extends

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

~~All Implemented Interfaces:~~ ← Esto se verá más adelante

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

Direct Known Subclasses:

`JEditorPane, JTextArea, JTextField`

Comentario: la **sobreescritura** de elementos heredados (principalmente métodos)

4.- Elementos relacionados con la Orientación a Objeto

4.4 - CLASES Y MÉTODOS ABSTRACTOS

http://gtts.ehu.es/German

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- }
```

4.- Elementos relacionados con la Orientación a Objeto

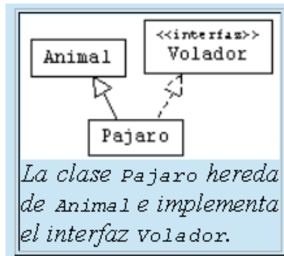
4.5 - INTERFACES

http://gtts.ehu.es/German

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



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

```
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- }
```

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

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.)

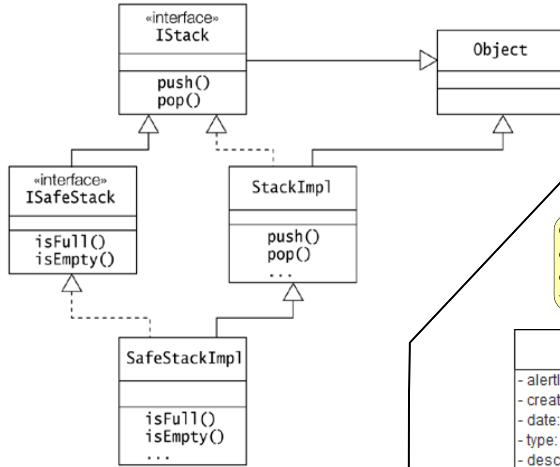
javax.swing

Interface Scrollable

All Known Implementing Classes:

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

Una estructura de clases e Interfaces




```
interface IStack {
    void push(Object o);
    Object pop();
}

class StackImpl implements IStack{
    @Override public void push(Object o){
        //TODO código de la rutina
    }
    @Override public Object pop(){
        //TODO código de la rutina
    }
}

interface ISafeStack extends IStack {
    boolean isFull();
    boolean isEmpty();
}

class SafeStackImpl
  extends StackImpl implements ISafeStack {
    @Override public boolean isFull(){
        //TODO código de la rutina
    }    @Override public boolean isEmpty(){
        //TODO código de la rutina
    }
}
```

- El término "@override" es una anotación. Es algo que veremos más adelante (aunque aquí resulta evidente su significado, veremos qué sentido tiene usarlo)
- En las clases los métodos son "public". Esto también lo veremos más adelante y tiene que ver con que de forma implícita los métodos en los interfaces so son y por tanto es parte de la herencia y debe explicitarse.

Estructura de una aplicación (Android)

Github es un "git" online, un sistema de control de versiones

https://github.com/WheresMyBus/android/wiki/System-Design-Specification

App. usada en Seatle

Alert
- alertId: int
- creatorId: String
- date: Date
- type: String
- description: String
- upvotes: int
- downvotes: int
+ upvote(Callback):
+ downvote(Callback):
+ unvote(Callback):
+ getComments(Callback):
+ getCreatorID():
+ setId():

about

Comment
- id: int
- creatorId: String
- data: String
- upvotes: int
- downvotes: int
- date: Date
+ upvote(Callback)
+setId(): boolean
+ downvote(Callback)
+ getCreatorId()
+ unvote(Callback)

Bus
- latitude: double
- longitude: double

is on a

BusStop
- latitude: double
- longitude: double
- coordinates: pair<double>
- direction: String
- name: String
- routes: List<Route>

is along

RouteAlert
- route: Route

about 1

Route
- number: String
- name: String
- id: String

1

NeighborhoodAlert
- neighborhood: Neighborhood
- routesAffected: List<Route>
+addAffectedRoute():

is about a 1

Neighborhood
- id: int
- name: String

M
C
V

WMBController
+ getAlerts(): list<Alert>
+ getAlerts(routeNumber: int): list<Alert>
+ getAlerts(neighborhood: Neighborhood): list<Alert>
+ getAlerts(coords: pair<double>, radius: double): list<Alert>
+ postAlert(alert: Alert): boolean
+ upvote(alert: Alert): boolean
+ downvote(alert: Alert): boolean
+ getNeighborhoods(): list<Neighborhood>
+ getComments(alert: Alert): list<Comment>
+upvoteComment(comment: Comment): boolean
+downvoteComment(comment: Comment): boolean

OBAController
+ getRouteNumbers(): set<int>
+ getBusStops(routeNumber: int): set<BusStop>
+ getBuses(routeNumber: int): list<Bus>
+ getBusLocation(id: int): pair<double>

AlertForumActivity

SubmitAlertActivity

MainActivity

MapActivity

CatalogActivity

4.- Elementos relacionados con la Orientación a Objeto

4.6 - POLIMORFISMO

http://gtts.ehu.es/German

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.

El polimorfismo, para ser consistente, debe restringir determinadas posibilidades que veremos más adelante a la hora de reescribir métodos (tienen que ver con los ámbitos de acceso y con el procesamiento de errores)
: 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]
```

Pasar a patrón Delegation Event Model (Observer) en GUIs y volver

4.- Elementos relacionados con la Orientación a Objeto

4.7 - ENCAPSULAMIENTO (ÁMBITOS DE ACCESIBILIDAD)

http://gtts.ehu.es/German

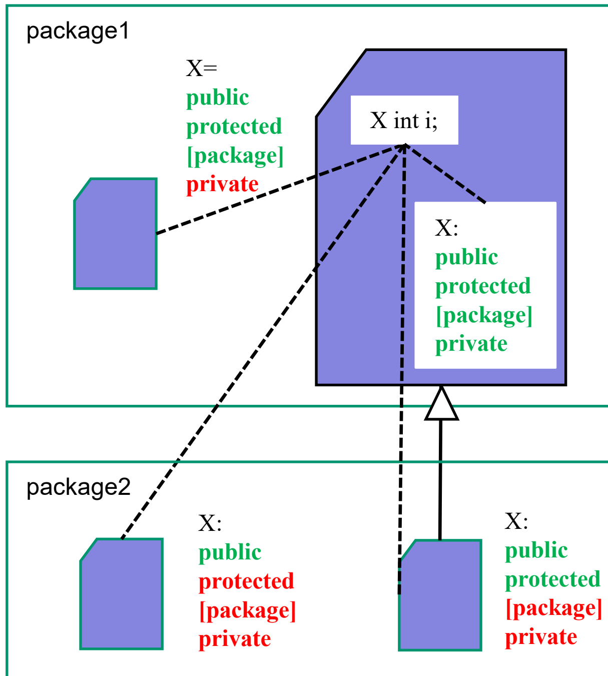
	Ámbito de acceso			
	clase	+paquete	+subclases	+todo
private	X			
package	X	X		
protected	X	X	X	
public	X	X	X	X

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

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

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- //
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- }
```



Posibilidades de acceso a una variable en función de su atributo de acceso (X)

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

Ámbitos de acceso y reescritura de métodos

```
package packA;

public class A {
  protected int i;
  // resto de definición de la clase
}
```

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

```
package packB;

public class B extends packA.A {
  void unMetodo() {
    this.i=10;
    ((packA.A)this).i=10;
  }
  static void otroMetodo(packA.A a, packB.B b) {
    a.i = 10;
    b.i = 10;
  }
  // resto de definición de la clase
}

class C {
  void unMetodo(packA.A a, packB.B b) {
    a.i = 10;
    b.i = 10; }
  // resto de definición de la clase
}
```

¡OJO!
La reescritura de un método no puede ser menos accesible que el método heredado.

```
public class A {
  public void m(){}
}

class B extends A{
  private void m(){}
}
```

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 → lo veremos en detalle más adelante.

Java 9 refuerza el encapsulamiento: los módulos

Módulo= conjunto de paquetes diseñado para ser reutilizado

Nos limitaremos a una visión muy superficial.
En Java Magazine sept/oct de 2017, a partir de la página 18, se da una explicación detallada de los módulos así como de sus motivaciones y el modo de ser propuestos e incluidos en una versión

El fichero module-info.java

```
module abc.xyz{
        exports com.foo.bar;
    }
```

servicios

```
module java.base {
  //no requiere nada al ser el módulo básico
  exports java.io;
  exports java.lang;
  ...
  exports jdk.internal.org.xml.sax to jdk.jfr;
  exports jdk.internal.org.xml.sax.helpers to jdk.jfr;
  ...
  uses java.util.random.RandomGenerator;
  uses java.util.spi.CalendarDataProvider;
  ...
  provides java.nio.file.spi.FileSystemProvider with
           jdk.internal.jrtfs.JrtFileSystemProvider;
  provides java.util.random.RandomGenerator with
    java.security.SecureRandom,
    java.util.Random,
    java.util.SplittableRandom;
}
```

Palabras clave restringidas (fuera de la declaración de módulos no son palabras clave)
exports, module, open, opens, provides, requires, uses, with, to, transitive

requires [p] - (dependencia) - se utiliza material de otro módulo
Cuando "A requires B" se dice que "A lee B" y "B es leído por A"

requires transitive [p] - id. "requires" y transmite la dependencia a quien lea el módulo
Cuando "A requires transitive B", si "Z requires A" implicitamente "Z requires B"

exports [p] – pone a disposición su material público y protegido.

exports [p] to [q,r,…,t] – (qualified export) - id. "export" limitando lectores.

uses [X] – (consume servicio) – utiliza objetos de una clase que concreta o implementa X

provides [X] with [C] – (proporciona servicio) – pone a disposición la clase C (servicio) que implementa o extiende X
El modulo será un "proveedor de servicio".

opens [p] – permite acceso en tiempo de ejecución. (clases públicas y protegidas accesibles)
En consecuencia permite la introspección, que queda imposibilitada si no hay "export" o "opens"

opens [p] to [q,r,…,t] – id "opens" limitando módulos con acceso.

open – atributo de módulo indicando que todos sus paquetes son abiertos
open module modulename { … }

A, B – módulos
p,q,r,t – paquetes
X – clase abstracta o interfaz
C – clase concreta

4.- Elementos relacionados con la Orientación a Objeto

4.8 - EL BLOQUE STATIC Y LOS ATRIBUTOS STATIC Y FINAL

http://gtts.ehu.es/German

final

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 definibles en tiempo de ejecución)
- para métodos, no pueden ser redefinidos en una subclase.

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- //
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- }
```

static

"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 pueden invocar a través de la clase)

```
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. (el siguiente apartado muestra todo sobre inicialización)

```
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- }
```

Un bloque de código sin el "static" delante, actúa igualmente de inicializador pero, como es de esperar, con accesos a los campos no estáticos (de los objetos)

4.- Elementos relacionados con la Orientación a Objeto

4.9 - INSTANCIACIÓN, INICIALIZACIÓN Y ELIMINACIÓN DE OBJETOS

http://gtts.ehu.es/German

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“)

```
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- }
```

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- 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

¡OJO!

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).

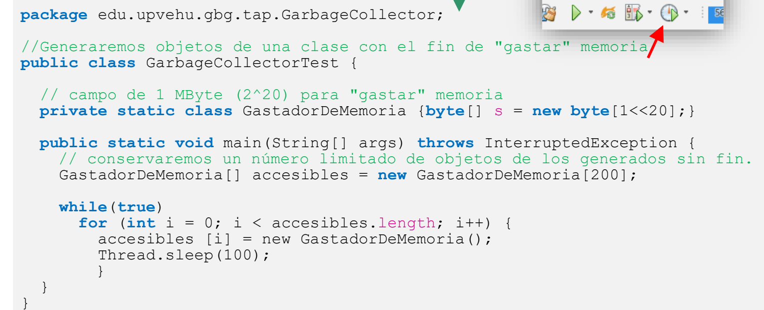
Analizaremos cómo se construyen los objetos con un ejemplo en la siguiente página, pero antes hablaremos de Clonación.

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.)

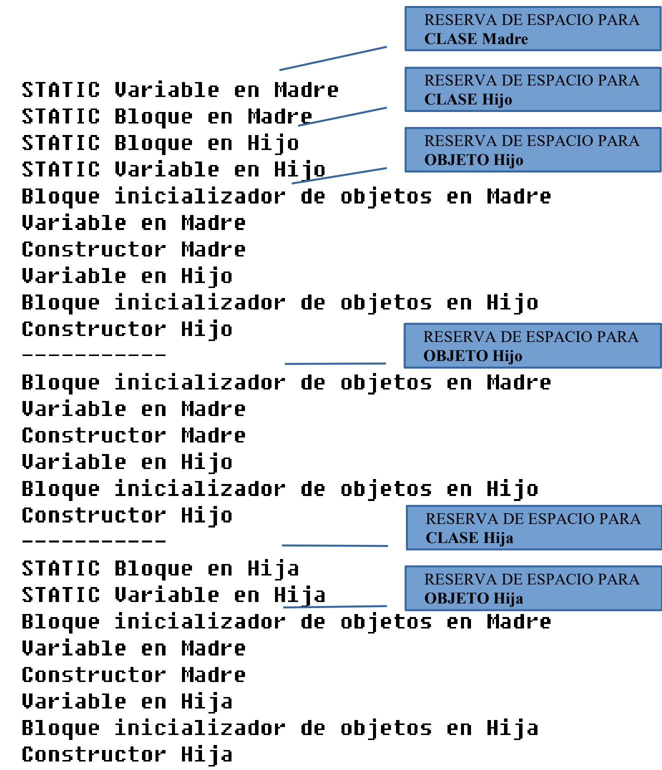
Probar el recolector de basuras con profiling




```
package edu.upvehu.gbg.tap.GarbageCollector;

//Generaremos objetos de una clase con el fin de "gastar" memoria
public class GarbageCollectorTest {

  // campo de 1 MByte (2^20) para "gastar" memoria
  private static class GastadorDeMemoria {byte[] s = new byte[1<<20];}

  public static void main(String[] args) throws InterruptedException {
    // conservaremos un número limitado de objetos de los generados sin fin.
    GastadorDeMemoria[] accesibles = new GastadorDeMemoria[200];

    while(true)
      for (int i = 0; i < accesibles.length; i++) {
        accesibles [i] = new GastadorDeMemoria();
        Thread.sleep(100);
        }
  }
}
```

Tanto la clonación como la finalización son mecanismos que presentan “debilidades” y por ello están puestos en cuestión y se estudian alternativas.

```java
public class InitDemo {

    public static void main(String[] args) {
        new Hijo();System.out.println("-----------");
        new Hijo();System.out.println("-----------");
        new Hija();
    }

    static String muestraYRetorna(String s) { System.out.println(s); return s; }
}

class Madre {
    { InitDemo.muestraYRetorna("Bloque inicializador de objetos en Madre"); }
    String varMadre = InitDemo.muestraYRetorna("Variable en Madre");
    static String staticVarMadre = InitDemo.muestraYRetorna("STATIC Variable en Madre");
    static { InitDemo.muestraYRetorna("STATIC Bloque en Madre"); }
    Madre() { InitDemo.muestraYRetorna("Constructor Madre");}
}

class Hijo extends Madre {
    static { InitDemo.muestraYRetorna("STATIC Bloque en Hijo"); }
    String varHijo = InitDemo.muestraYRetorna("Variable en Hijo");
    static String staticVarHijo = InitDemo.muestraYRetorna("STATIC Variable en Hijo");
    { InitDemo.muestraYRetorna("Bloque inicializador de objetos en Hijo"); }
    Hijo() { InitDemo.muestraYRetorna("Constructor Hijo"); }
}

class Hija extends Madre {
    static { InitDemo.muestraYRetorna("STATIC Bloque en Hija"); }
    String varHijo = InitDemo.muestraYRetorna("Variable en Hija");
    static String staticVarHijo = InitDemo.muestraYRetorna("STATIC Variable en Hija");
    { InitDemo.muestraYRetorna("Bloque inicializador de objetos en Hija"); }
    Hija() { InitDemo.muestraYRetorna("Constructor Hija"); }
}
```



En primer lugar se inicializa la parte estática, ordenadamente desde la clase más general a la más particular y en el orden en que se han definido los campos en cada caso (incluyendo el bloque **static** como un campo más. Si una parte de la herencia ya se ha inicializado con la creación de otro objeto, no se re-inicializa.

A continuación se inicializan los campos del objeto desde la clase más general a la más particular y en el orden en que se han definido los campos en cada caso (incluyendo los bloques de pre-inicialización), y después se ejecuta el constructor, que llama recursivamente a los constructores de las clases madre y por tanto se ejecutan de la más general a la más particular.

4.- Elementos relacionados con la Orientación a Objeto

4.10 - ARRAYS

http://gtts.ehu.es/German

Como hemos adelantado en el tema anterior, los arrays son objetos.
Su contenido no sólo puede ser de tipo primitivo, sino que pueden contener referencias a objetos

Declaración estilo C vs. estilo Java

```
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"
```

Tamaño predeterminado

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

Asignación

```
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();
```

Tamaño determinado en ejecución

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

Asignación en bloque

```
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()};
```

Multidimensionales

```
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;
```

```
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- }
```



```
for (referencia :colección) sentencia
```

```
1- String[] saludos={"Hola","adios"};
2- for (String s: saludos) ...
```

... o con definición anónima

```
For (String s: new String[]{"Hola","adios"} ) ...
```

El FOR de "colecciones"

4.- Elementos relacionados con la Orientación a Objeto

4.11 - ENUMERACIONES

http://gtts.ehu.es/German

Estudiaremos superficialmente este tema con un ejemplo.

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

```
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
    }
}
```

(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.)

4.- Elementos relacionados con la Orientación a Objeto

Adenda – Clases dentro de clases. Clases estáticas y dinámicas. Clases en métodos. Clases anónimas.

http://gtts.ehu.es/German

```
public class A {                //Clase pública. El fichero DEBE llamarse A.java (sólo puede haber una clase pública)

    static class B {            //Clase interna estática (como se declara "package" puede accederse como A.B)
    //TODO Código de la clase B
    }

    class C {                   //Clase interna dinámica. Cada objeto tendrá asociada su propia clase interna
    //TODO Código de la clase C
    }

    A objA = new A();           //Objeto de la clase (externa) A
    B objB = new B();           //Objeto de la clase estática interna B
    C objC = new C();           //Objeto de la clase C (no podría ser estático)
    A anonimo = new A() {       //Objeto de una subclase anónima de A (hacemos la referencia "anonimo" a nivel A
    //Código de la subclase anónima de A
    };

    void metodo1(E objE) {
        class D {               //Clase local
        //TODO código de la clase D
        }
        D objD = new D();       //Objeto de la clase local D
        //TODO código del método m (que usará los objetos d y e
    }

    void metodo2() {            //En este método llama a m(.) aportando un objeto de clase anónima como parámetro
        metodo1(new E(){        //el parámetro es un nuevo objeto de una subclase anónima de E
                @Override
                void metodo(){
                //TODO código del método
                };
        });
    }
}

class E {                       //Clase no pública dentro del fichero A.java
    A.B ab = new A.B();         //Objeto de clase interna A.B (no podría ser A.C)
    void metodo(){
    //TODO código del método
    }
}
```