Java 2

Índice

Capítulo 1 - Primeros pasos en Java	2
Capítulo 2 - Lenguaje Java	2
Capítulo 3 - Conceptos básicos de Java	7
Capítulo 4 - Programas básicos en Java	10
Capítulo 5 - Clases Java	11
Capítulo 6 - Ficheros en Java	13
Capítulo 7 - Excepciones en Java	15
Capítulo 8 - Tareas y multitarea	16
Capítulo 9 - Delegación de eventos	18
Capítulo 10 - AWT	20
Capítulo 11 - Swing	21
Capítulo 12 - Gráficos	23
Capítulo 13 - Comunicaciones en red (java.net.*)	27

Capítulo 1 - Primeros pasos en Java

1) El hola mundo:

```
public class HolaMundo {
	public static void main (String[] args) {
		System.out.println("¡Hola mundo!");
	}
}
```

2) El hola mundo en applet:

```
import java.awt.Graphics;
import java.applet.Applet;

public class HolaMundo extends Applet {
	public void paint (Graphics g) {
		g.drawString("Hola mundo.", 25, 25);
	}
}
```

3) Argumentos en la línea de comandos:

Para tener en cuenta los posibles argumentos en main hay que preocuparse de la variable args:

```
public class ejemplo {
	public static void main (String[] args) {
		for (int i = 0; i < args.length; i++) {
			System.out.println(args[i]);
		}
	}
}
```

Capítulo 2 - Lenguaje Java

1) Comentarios:

```
// Comentario de una linea...
/* Comentario de
   varias lineas... */
/** Comentario para documentar el
    código con javadoc... */
```

2) Identificadores:

Funcionan como en C/C++, Pascal y C#:

```
tipo identificador [= valor][, ...];
```

3) Tipos de datos y literales:

+ Enteros: byte, short, int, long (complemento a 2)
8 16 32 64 bits
Ejemplos: 221, 077, 0xDC00
+ Reales: float, double (IEEE 754)
32 64 bits
Ejemplos: 3.14, 2e12, 3.1E12
+ Booleanos: boolean
Ejemplos: true (!= 0), false (== 0)
+ Caracteres: char
Ejemplos: 'a', '\t', '\u0A13' (Número unicode).
+ Cadenas: String, StringBuffer
Ejemplos: "Yo soy Tetsuo..."

Notas: Para los enteros y reales 14l o 23L serán tratados como long, 3.14f como float y 3.14d como double.

4) Operadores:

+ Aritméticos: + - * / %
+ Concatenación: +
+ Unarios: + - ++ --
+ Relacionales: > >= < <= == !=
+ Condicionales: && || !
+ A nivel de bits: >> << >>> & | ^ ~
+ Asignación: = += -= *= /= %= &= |= ^= <<= >>= >>>=
+ Operador if: expresión ? sentenciaTrue : sentenciaFalse
+ Cast: (tipoSimple)

5) Constantes:

```
final tipo nombre = valor;
```

6) Arrays:

```
tipo nombre[] = { new tipo[tamaño];
                | { valor, ...};
                | null;
```

Los arrays en java son objetos frente a los de C o Pascal. Por ese detalle tienen el campo **length** que contiene el número de elementos o tamaño del array. Además si accedemos a una posición fuera de los límites se lanzará una excepción de tipo **ArrayIndexOutOfBounds**. Ejemplos:

```
int lista[][] = new int[10][5];
String nombres[] = new String[10];
nombres[0] = new String("Juan");
```

7) Control de flujo:

```
if(condición) {
      //...
}[else {
      //...
}]

switch (expresión) {
      case valor:
            //...
            break;
      [//...]
      [default:
            //...]
}

try {
      //...
} catch (excepción) {
      //...
}
```

```
for(inicio; condición; iteración) {
      //...
}

while(condición) {
      //...
}

do {
      //...
} while(condición);

break;
continue;
return [expresión];
```

8) Almacenamiento de datos:

Al ser los arrays tipos de un tamaño fijo y que no varía de forma flexible, hay en Java una serie de clases para almacenar datos de formas más eficientes según el caso. Los más importantes son **`Vector`**, **`BitSet`**, **`Stack`**, **`Hashtable`**, **`Map`**, **`List`** y **`Set`**. Siendo Java un lenguaje de POO, todo tipo de datos es una clase y para garantizar la genericidad absoluta, toda clase es hija de **`Object`** en Java. Esto tiene sus ventajas, como que podemos usar Vector para cualquier tipo de dato, pero también ocurre que en un Vector se podrían almacenar diferentes tipos de datos, teniendo el programador que cuidarse con este detalle.

+ **Enumeration:** Para recorrer los diferentes tipos de contenedores de forma genérica, en C++ se usaría un iterador y en java se usa la clase Enumeration. Esta tiene los métodos:
  - boolean **hasMoreElements** () → Indica si quedan elementos por recorrer en el contenedor.
  - Object **nextElement** () → Devuelve el elemento actual y pasa al siguiente.
+ **Vector:** Rápido de recorrer frente una lista, pero más lento añadiendo.
  - void **addElement** (Object e) → Añade un elemento por el final.
  - Object **elementAt** (int index) → Devuelve un elemento.
  - int **size** () → Tamaño actual del vector.
  - Enumeration **elements** () → Da un Enumeration para recorrerlo.
+ **Bitset:** Se trata de un vector de bits, cuyo tamaño mínimo es 64 bits. Por ello tiene métodos parecidos a Vector, además de:
  - void **set** (int index) → Pone a 1 una posición del vector.
  - void **clear** (int index) → Pone a 0 una posición del vector.
  - boolean **get** (int index) → Devuelve una posición.
+ **Stack:** Sirve para tener una pila con su funcionalidad.

- void **push** (Object e) → Mete un elemento en la pila.
- Object **pop** () → Saca un elemento de la pila.
- boolean **empty** () → Indica si está la pila vacía.

+ **Hashtable:** Es un vector que no necesariamente usa un número como clave de acceso a la posición ocupada (un String por ejemplo).
  - int **size** () → Tamaño de la tabla, Nº de elementos.
  - boolean **isEmpty** () → Indica si está vacía.
  - void **put** (Object clave, Object valor) → Añade o cambia un elemento.
  - Object **get** (Object clave) → Devuelve un elemento.
  - void **remove** (Object clave) → Borra un elemento.
  - Enumeration **keys** () → Para listar las claves.
  - Enumeration **elements** () → Para listar los elementos.
  - boolean **containsKey** (Object clave) → Indica si existe ese elemento.

  Para ganar rapidez frente a Dictionary, Hashtable utiliza de cualquier clase los métodos `int hashCode ()` y `boolean equals (Object obj)`, para generar el código hash de la clave. Y puede ocurrir que deseamos usar como clave un tipo de dato nuestro, deberemos sobre-escribir ambos métodos para que no sean llamados desde Object.
+ **Collection:** De esta clase surgen List y Set.
+ **Map:** Es una lista de objetos clave-valor, parecido a Hashtable.

9) <u>Colecciones</u>:

Todo el que herede de Collection (List y Set entre otros) tendrán:

- boolean **add** (Object e) → Añade un elemento.
- boolean **add** (Collection c) → Añade varios elementos.
- void **clear** () → Borra todos los elementos.
- boolean **contains** (Object e) → Indica si ya esta ese elemento.
- boolean **isEmpty** () → Indica si está el objeto vacío.
- Iterator **iterator** () → Devuelve un iterador para recorrer el contenido.
- boolean **remove** (Object e) → Borra un elemento.
- boolean **removeAll** (Collection c) → Borra varios elementos.
- boolean **retainAll** (Collection c) → Borra los elementos que no están en C.
- int **size** () → Número de elementos.
- Object [] **toArray** () → Devuelve el contenido en forma de array. Si puede, lanza una UnsupportedOperationException.

+ **List:** Permiten la repetición.
  - <u>ArrayList</u>: Equivale a Vector.
  - <u>LinkedList</u>: Es una lista doblemente enlazada. Dispone además de los métodos addLast(), getFirst(), getLast(), removeFirst(), removeLast().
+ **Set:** No permite elementos repetidos.
  - <u>HashSet</u>: Se usa para conjuntos grandes.
  - <u>ArraySet</u>: Se usa para conjuntos pequeños.
  - <u>TreeSet</u>: Sirve para tener un conjunto ordenado con un árbol AVL.
+ **Map:** No hereda de Collection y sirve para guardar parejas de valores.
  - <u>HashMap</u>: Usa una tabla hash para funcionar.
  - <u>ArrayMap</u>: Usado para Maps pequeños.
  - <u>TreeMap</u>: Usa un árbol AVL.

10) Ordenación y busqueda:

+ **Arrays:** Hay un par de funciones implementadas para todos los tipos básicos y también para String y Object. Están en la clase Arrays y son los métodos estáticos:
    - void **sort** (tipo variable[] [, Comparator cmp])
    - int **binarySearch** (tipo variable[], tipo elemento [, Comparator cmp])

+ **Comparable y Comparator:** Pero para aprovechar esas funciones con las clases que nos hagamos tendremos que realizar unos trapicheos:
    + **Comparator:** Esta interfaz tiene una función que sort y binarySearch pueden llamar para trabajar:

```
import java.util.*;
public class StringCmp implements Comparator {
        public int compare (Object obj1, Object obj2) {
              String s1 = ((String) obj1).toLowerCase();
              String s2 = ((String) obj2).toLowerCase();
              return s1.compareTo(s2);
        }
        public int Find (String Lista[], String Nombre) {
              Arrays.sort(Lista, this);
              return Arrays.binarySearch(Lista, Nombre, this);
        }
}
```

La función compare devolverá:
- val < 0 → obj1 < obj2
- val == 0 → obj1 == obj2
- val > 0 → obj1 > obj2

    + **Comparable:** Sin embargo el método anterior es un poco farragoso y tenemos el siguiente modo de hacer las cosas:

```
import java.util.*;
public class Numero implements Comparable {
        private int num;
        public Numero (int n) { num = n; }
        public int compareTo (Object obj) {
              int aux = ((Numero) obj).num;
              if(num < aux) return -1;
              else if(num == aux) return 0;
              else return 1;
        }
        public static void main (String args []) {
              Numero lista[] = new Numero[20];
              for (int i = 0; i < lista.length; i++)
                    lista[i] = new Numero((int) (Math.random() *
                                                  1000));
              Arrays.sort(lista);
              System.out.println(Arrays.binarySearch(lista,
                                       lista[8]));
        }
```

```
}
```
+ **Listas:** Los contenedores de datos de Collection, en vez de usar la clase Arrays, usa la clase `Collections`, que tiene los métodos estáticos:
   - void **sort** (Collection c [, Comparator cmp])
   - int **binarySearch** (Collection c, Object objeto [, Comparator cmp])

  Y también para estos métodos funciona el sistema de Comparator y Comparable.

+ **Utilidades:** También en la clase Collections hay otros métodos estáticos usados para:
   - Enumeration **enumeration** (Collection c) → Devuelve un Enumeration para poder recorrer el contenedor.
   - Object **max** (Collection c [, Comparator cmp])
   - Object **min** (Collection c [, Comparator cmp])
   - List **nCopies** (int n, Object o) → Crea una lista de tamaño n, con todas las posiciones referenciando a o.
   - List **subList** (List lista, int ini, int fin) → Devuelve una sublista con los elementos de la posición ini hasta fin.
   - Collection **unmodifiableCollection** (Collection c) → Crea una copia de solo lectura del contenedor que le hemos pasado.
     - List **unmodifiableList** (List l)
     - Set **unmodifiableSet** (Set s)
     - Map **unmodifiableMap** (Map m)
   - Collection **synchronizedCollection** (Collection c) → Crea una copia del contenedor que tendrá un control síncrono en la multitarea.
     - List **synchronizedList** (List l)
     - Set **synchronizedSet** (Set s)
     - Map **synchronizedMap** (Map m)

Capítulo 3 - Conceptos básicos de Java

1) Objetos:

En Java todo son objetos (menos los tipos básicos), de aquí sacamos que para definirlos hemos de programar clases de objetos. Para manejar un objeto hay que declararlo, pedir memoria y llamar a sus miembros con el punto:

```
Casa micasa;
micasa = new Casa(direccion);
micasa.mostrarInformacion();
micasa = null; // Con esto es "destruido".
```

2) La memoria:

Java gestiona la memoria a su modo, tan solo hemos de pedirla y nos la da con un **new**. ¿Pero como se libera la memoria? Java tiene lo que se llama un **recolector de basura**, que cada cierto tiempo o si se queda sin memoria llama a sus rutinas para liberar espacio. Para decidir que borrar y que no, se mira que regiones de memoria no están referenciadas desde nuestro programa y se borran, así que si hay algo que no deseamos usar más lo mandamos a referenciar a **null** (micasa = null;). También podemos llamar al recolector invocándolo de este modo:

```
System.gc();
```

Si hace su trabajo luego, es problema de Java, no nuestro. También se puede "destruir" un objeto llamando su método **`finalize()`**, que se hereda de **`Object`**:

```
protected void finalize () {}
```

Por eso hay que reescribirlo y rezar para que todo vaya bien al invocarlo. Pero también hay un **`System.runFinalization()`** para intentar obligar al recolector de basura a ejecutar los métodos finalize() de los objetos que libera.

3) Las clases:

Las clases en Java se definen:

```
[public] [abstract | final] [synchronizable] class nombre
    [extends superclase] [implements interfaz1 [, ...]] {
    // Declaraciones de los miembros...
}
```

Los modificadores aceptados sirven, según el que pongamos, para:
- **public:** Permite que la clase pueda ser usada fuera del paquete donde fue declarada. Y solo puede haber una clase pública en cada fichero .java de nuestro proyecto.
- **abstract:** La clase tiene al menos un método abstracto.
- **final:** La clase no puede ser heredada.
- synchronizable: Una clase síncrona solo permite el acceso a sus miembros, por un solo hilo cualquiera al mismo tiempo.

Otro punto ya comentado es que todas las clases heredan de **`Object`**, así que si no indicamos en **`extends`** otra cosa, la super clase por defecto será **`Object`**.

+ Las variables miembro:

```
[public | private | protected | package] [static] [final] [transient]
    [volatile] tipo nombre [= valorInicial];
```

El modificador de acceso por defecto es **`package`** y así funcionan:

	Misma clase	Hija en el paquete	Hija fuera del paquete	Clase en el paquete	Clase fuera
private	Sí	No	No	No	No
package	Sí	Sí	No	Sí	No
protected	Sí	Sí	Sí	No	No
public	Sí	Sí	Sí	Sí	Sí

Y los otros modificadores sirven para:
+ **static:** Crea un miembro estático y no instanciable (lo que implica que dicho miembro será global a toda la clase). Esto se utiliza normalmente para las constantes o para hacer cosas como un singleton.
+ **final:** Indica que su valor no va a ser cambiado, vamos que es una constante. Se recomienda usar static para no gastar la memoria absurdamente en las constantes.
+ **transient:** Indica que no es parte del estado estable de la clase.
+ **volatile:** Expresa que la variable es vulnerable a los accesos desde hilos.

+ Los métodos miembro:

```
[public | private | protected | package] [static] [abstract] [final]
   [native] [synchronized] tipo nombre (argumentos, ...)
   [throws excepciones, ...] { ; | {
   // Código o cuerpo de la función...
   [return [valor];]
} }
```

Los modificadores que existen son:
+ **static:** Para que el método sea estático (o método de la clase).
+ **abstract:** El método será abstracto, luego no tendrá código.
+ **final:** El método no podrá ser sobreescrito.
+ **native:** El código del método está escrito en otro lenguaje de programación.
+ **synchronized:** Solo se permite un acceso al mismo tiempo.

Cuando devolvemos un valor se usa **`return`** y si no queremos devolver nada en el tipo usamos **`void`**. A diferencia de C++, al devolver objetos hay que tener cuidado, porque lo que se devuelve es una referencia de este. Y respecto a los argumentos se definen:

```
(tipo nombre [[]][, ...])
```

Hay que tener muy en cuenta que si el tipo es un **objeto** (los arrays también son objetos), este se pasará **por referencia**, con lo que podrá ser cambiado, pero si es un **tipo básico** como int, el argumento se habrá pasado **por valor**. El **constructor** se define como una función que lleva el mismo nombre que la clase y no se devuelve nada (ni siquiera se pone el void):

```
public MiClase () {}
public MiClase (int dato) {}
```

De este ejemplo vemos que con el mismo nombre podremos tener varias funciones, a condición de que ninguna tenga los mismos argumentos.

+ La herencia:
Siempre que heredemos de una clase podremos sobreescribir los métodos que no estén capados con final. Y podremos llamar a los métodos del padre usando la variable **`super`**, que es una referencia a la clase padre, igual que **`this`** es una referencia a la propia clase.

4) La clase Object:

Tiene los siguientes métodos de interés:
- public boolean **equals** (Object obj) → Sirve para comparar si dos objetos (this y obj) son iguales. Por defecto lo que comprueba es si la dirección de memoria donde están ambos objetos es la misma, por lo que hay que sobreescribirlo a menudo para nuestros tipos.
- public final native Class **getClass** () → Devuelve un descriptor de tipo Class con información sobre la clase, como el nombre, el padre y otros. Por ejemplo para saber el nombre sería: `this.getClass().getName();`
- public String **toString** () → Devuelve una cadena que normalmente representaría los datos de la clase en formato texto.
- protected void **finalize** ()

- public final void **wait** ()
- public final native void **wait** (long timeout)
- public final native void **notify** ()
- public final native void **notifyAll** ()

5) Las clases abstractas y las interfaces:

Las clases que declaramos como abstract, han de llevar al menos un método abstract:

```
public abstract class Animal {
    public abstract void Jugar ();
}
```

Obviamente una clase abstracta no puede ser instanciada. Pero en Java hay un "problema", solo se puede heredar de una clase, por lo que se inventaron las interfaces, que a diferencia de las clases, ningún método de una interfaz se declara con código:

```
[public] interface nombre [extends interfaz, ...] {
    // Declaración de variables y métodos...
}
```

6) Los paquetes:

En C++ se parecería mucho a los namespace. Para crear uno hemos de crear un árbol de directorios de tal modo que si hacemos "`import java.applet.*;`" añadiremos las clases del paquete java.applet, que estará almacenado en "java\applet" del disco duro o del zip que Java esté usando. En el caso anterior en cada .java la primera sentencia es un: "`package java.applet;`" Así es como se crea uno.

Cuando importemos un paquete es recomendable intentar indicar la clase en vez del *, para no cargar el paquete entero. Y si queremos cargar una clase que no está en ningún paquete, solo hay que poner "`import MiClase;`"
...

Capítulo 4 - Programas básicos en Java

1) Los applets y el html:

La etiqueta **applet** tiene obligatorios los campos code, with y height, pero hay más y dentro de applet hay una forma de pasar parámetros al programa.
+ **Code** = Nombre de la clase principal.
+ **Width** = Ancho del programa.
+ **Height** = Alto del programa.
+ **Codebase** = URL base del applet.
+ **Alt** = Texto alternativo.
+ **Name** = Nombre de la instancia.
+ **Align** = Justificación del objeto html.
+ **Vspace** = Espaciado vertical.
+ **Hspace** = Espaciado horizontal.

Para pasar parámetros se usa <**param** name= value= >, en el que name será el nombre del parámetro y value el valor de tipo String. Para conseguir esos valores dentro de la clase hija de Applet tenemos que llamar a:

```
String getParameter (String name);
```

Por último la etiqueta applet tiene un campo llamado **Archive**, que sirve para indicar que el programa está dentro de un fichero zip (Archive = "programa.zip"). Nota: Los ficheros jar también son ficheros zip.

2) Escribir applets java:

La clase Applet tiene diversos métodos que hay que reescribir para hacer uso de este sistema de aplicaciones:
- public void **init** () → Esta función es llamada al crearse el Applet. Se suele usar para cargar recursos, redimensionar el tamaño en pantalla o asignar valores a globales.
- public void **destroy** () → Se llama cuando el applet es destruido de memoria y suele usarse para finalizar todos los hilos.
- public void **start** () → Esta función es llamada cuando el applet recibe el foco.
- public void **stop** () → Cada vez que desaparezca de pantalla el applet, perderá el foco y se invocará esta función.

Además hay otros métodos útiles como:
- void **resize** (int width, int height) → Cambia la resolución en pantalla del applet.
- **width**, **height** → Son propiedades y no métodos, que almacenan el tamaño del applet en la medida de pixels, siendo width la medida del ancho y height la del alto.
- void **repaint** () → Manda a repintar la pantalla.
- String **getParameter** (String name) → Obtiene un parametro, si no hubiera sido definido en el html, devuelve null al no existir.
- String **getDocumentBase** () → Da el directorio base de la aplicación.
- void **print** (Graphics g) → Imprime el contenido de la pantalla.

Por último para actualizar el contenido de la pantalla se puede reescribir los métodos:
- public void **paint** (Graphics g) → Cada vez que la aplicación determina que tiene que actualizar el contenido en pantalla, llama a esta función (por ejemplo al ocultar y destapar una zona con otra ventana).
- public void **update** (Graphics g) → En realidad paint es llamada a través de update, es decir, que es a update a la que realmente se llama al actualizar la pantalla y si queremos podemos cambiar su funcionamiento.

Capítulo 5 - Clases Java

1) Math:

Contiene de forma estática diversas **operaciones** matemáticas: abs, sin, cos, tan, asin, acos, atan, atan2, exp, log, sqrt, ceil, floor, rint, pow, round, random, max, min. También contiene **constantes** como E y PI.

2) Números enormes:

Para almacenar números absurdamente grandes y con total precisión están las clases **BigInteger** y **BigDecimal**, la primera para enteros y la segunda para reales. Si al asignarle un valor se lo indicamos mal, lanzará una **NumberFormatException**.

3) Clases de tipos simples:

char	→	Character
byte	→	Byte
short	→	Short
int	→	Integer
long	→	Long
float	→	Float
double	→	Double
boolean	→	Boolean

4) Random:

Esta clase intenta generar números aleatorios mediante el algoritmo de Donald Knuth (The Art of Computer Programming).

5) String:

Es la clase para almacenar cadenas y tiene sobre-cargado el operador + para concatenar dos cadenas creando un nuevo String. Por ello para copiar una cadena hay que hacer por ejemplo:

```
// Como si fuera un constructor copia como en C++.
String cad2 = new String((new StringBuffer(cad1)).toString());
```

Aunque en realidad no serviría eso para nada, porque los objetos String no se pueden modificar en su contenido. Para manipular cadenas se usa StringBuffer.

6) StringBuffer:

Para manipular cadenas se emplea este tipo que se declara:

```
StringBuffer str = new StringBuffer([tamaño | cadena]);
```

Para manipularlas no se usa ni +, ni += como en String, sino los métodos:
- StringBuffer **append** (String str) → Añade al final.
- StringBuffer **insert** (String str) → Añade al inicio.

Además tenemos como ayuda otros métodos:
- int **length** () → Tamaño del buffer.
- char **charAt** (int index) → Devuelve un caracter.
- void **getChars** (int begin, int end, char dst[], int dstBegin) → Devuelve en dst un fragmento del buffer.
- String **toString** () → Devuelve el buffer convertido en cadena.
- void **setLength** (int newlen) → Cambia el tamaño.
- void **setCharAt** (int index, char ch) → Cambia un caracter.
- int **capacity** () → Capacidad máxima del buffer.
- void **ensureCapacity** (int mincap)

- int **reverse** () → Le da la vuelta al contenido.

7) StringTokenizer:

Sirve para sacar fragmentos de una cadena con delimitadores:

```
StringTokenizer st = new StringTokenizer(cadena, strDelimitadores);
String fragmento = st.nextToken();
```

8) Properties:

Sirve para listar las propiedades del entorno o para manipularlas también, aunque no es algo muy fácil de usar.

9) Runtime:

Es la clase con información del interprete de Java para poder obtener una referencia no hemos de hacer new, ya que el constructor es privado, sino:

```
Runtime r = Runtime.getRuntime();
```

Los applets y otros fragmentos no seguros de código provocarán que se lance una **`SecurityException`** al llamar a los métodos de esta clase. Algunos de ellos son:

- int **totalMemory** () → Memoria total de la máquina virtual de Java.
- int **freeMemory** () → Memoria libre disponible.
- Process **exec** (String cmd[[]]) → Lanza en un nuevo hilo un comando que pretendíamos ejecutar en la consola del SO.

10) System:

Se usa para acceder a los dipositivos de I/O, el reloj del sistema, etcétera. Para la I/O tenemos los siguientes métodos:
+ static PrintStream **err**; // Salida de error estandar.
+ statis InputStream **in**; // Entrada estandar.
+ static PrintStream **out**; // Salida estandar.

Los PrintStream tienen los métodos print(), println() o write(). Y los InputStream tienen read() y readln() entre otros. También con System podemos llamar a **`exit([número])`**, para salir en cualquier lugar de la aplicación. O podemos cambiar el controlador de seguridad con setSecurityManager() o simplemente consultarlo con getSecurityManager().

Capítulo 6 - Ficheros en Java

1) La clase File:

Esta es la clase principal para tratar temas de ficheros y para declararlo hay tres formas:
- File (String ruta)
- File (String directorio, String fichero)
- File (File directorio, String fichero)

Y una vez creado tenemos que contemplar que con ficheros se pueden lanzar una **`IOException`** si algo va mal. Además en esta clase hay algunos métodos como:

- String getName () → Da el nombre del fichero.
- String getPath () → Da el directorio donde está el fichero.
- boolean exists () → Indica si el fichero existe.
- boolean canRead () → Indica si tiene permiso de lectura.
- boolean canWrite () → Indica si tiene permiso de escritura.
- int length () → Devuelve el tamaño del fichero.

2) Aplicaciones:

Pero con el objeto File a secas no hay mucho que hacer para manipular el contenido de un fichero. Java tiene montones de clases para manejar esto con streams. Los más básicos para el manejo son:

+ **FileOutputStream** → Para escribir en ficheros. Algunos de sus métodos son: write() y close().
+ **FileInputStream** → Para leer de ficheros. Entre sus métodos nos encontramos con: read() y close(). En el caso de read(), devuelve el número de bytes leidos.

En ambos casos al constructor hay que pasarle el fichero asignado en el objeto File, pasándole este objeto como argumento:

```
FileOutputStream fos = new FileOutputStream(miFichero);
```

Nota: La clase File tiene un campo estático llamado `separatorChar`, que sirve para saber cual es el carácter que delimita los nombres de los directorios, independientemente del sistema operativo que se esté usando.

3) Excepciones y seguridad:

La principal excepción que se suele lanzar en los temas de ficheros y flujos de entrada y salida, es la `IOException`. Otro tema importante es la seguridad, por ejemplo si intentamos en un applet o en un servlet manipular un fichero, se producirá un error de seguridad. Para eliminar este problema hay que usar la herramienta **PolicyTool**, para generar un fichero de configuración que otorgue nuevos privilegios a nuestra aplicación. Ejemplo (java.polizas):

```
grant {
        permission java.util.PropertyPermission "user.home", "read";
        permission java.io.FilePermission "${user.home}/textoe.txt", "write";
        permission java.io.FilePermission "${user.home}/textol.txt", "read";
}
```

4) Servlets:

Son programas web que a diferencia de los applets, estos son hijos de `HttpServlet` y sus principales métodos a sobreescribir son `doGet()` y `doPost()`. Uno se llama cuando la página se carga y el otro cuando desde un formulario html invocamos el método post.

5) Acceso aleatorio:

Para poder manejar de forma más flexible un fichero tenemos la clase `RandomAccessFile`, que nos permite abrir ficheros para escribir y leer donde queramos. Su constructor pide:

```
RandomAccessFile (File fichero, String modo);
```
Modo:
+ "r" Lectura
+ "w" Escritura
+ "rw" Ambas

Y tiene métodos como: **seek()**, **read()**, **write()**, **close()**, etcétera.

Capítulo 7 - Excepciones en Java

1) Manejar excepciones:

Con la POO se pensó en otro método más limpio para gestionar errores en los algoritmos y nacieron las excepciones, con lo que en vez de devolver un código de error, lo que se "suele" hacer ahora es:

```
throw NombreExcepción;
```

Esto es todo un logro para quitar ifs y tabulaciones interminables como ocurre en Pascal. Pero tiene su coste de rapidez y consumo de memoria (por ello no son muy populares en C++). Para gestionar esto tenemos:

```
try {
    // Código sosprechoso...
} catch (NombreExcepción objeto) {
    // Código de gestión del fallo...
} catch (Exception e) {
    // Código general para capturar cualquier excepción...
} finally {
    // Un código que siempre querramos ejecutar haya fallo o no...
}
```

2) Generar excepciones:

En Java la clase **Throwable** tiene dos hijas llamadas **Error** y **Exception**, todo esto forma el conjunto básico de objetos que pueden ser lanzados en nuestras aplicaciones. Error representa errores irrecuperables en el programa, vamos que no tienen solución, de hecho salvo en un caso (ThreadDead), cierran la aplicación. Para poder lanzar en un método una excepción hay que indicarlo en su cabecera, en la sección **throws**:

```
void leer () throws IOException, MiException { ... }
```

Con esto ya podremos usar throw para lanzar una excepción desde el método. Toda excepción contiene un mensaje que se puede obtener con **getMessage()**. Para crear una nueva clase de excepción solo hay que heredar de alguna:

```
class MiException extends Exception {}
```

Por último si ocurre una excepción que no es atrapada por ningún bloque catch del método, esta será lanzada a las funciones superiores hasta llegar al método de entrada del programa, si no hay ningún bloque catch que la capture. Si en el "main" no fuera tampoco capturada el programa finaizaría y el interprete de Java mostraría el error.

Capítulo 8 - Tareas y multitarea

1) Ejemplo de programa multi-hilo:

```
class Test extends Thread {
	private String Nombre;
	private int Retardo;
	public Test (String n, int r) {
		this.Nombre = n;
		this.Retardo = r;
	}
	public void run () { //Método "main" del hilo.
		try {
			Thread.sleep(this.Retardo);
		} catch (InterruptedException e) { ; }
		System.out.println("Fin " + this.Nombre + " " + this.Retardo);
	}
}

public class Multihilo {
	public static void main (String[] args) {
		Test t1, t2, t3;
		t1 = new Test("Hilo 1", (int) (Math.random() * 2000));
		t2 = new Test("Hilo 2", (int) (Math.random() * 2000));
		t3 = new Test("Hilo 3", (int) (Math.random() * 2000));
		t1.start();
		t2.start();
		t3.start();
	}
}
```

2) Creación y control de hilos:

Si queremos crear hilos hemos de crear una clase que derive de **`Thread`**, reimplementando algunos métodos. Dentro de Thread hay los siguientes métodos estáticos:

- Thread **currentThread** () → Da el hilo que se ejecuta actualmente.
- void **yield** () → Ordena un cambio de contexto para parar el hilo actual e iniciar la ejecución del siguiente.
- void **sleep** (long ms) →Suspende la ejecución del hilo unos milisegundos.

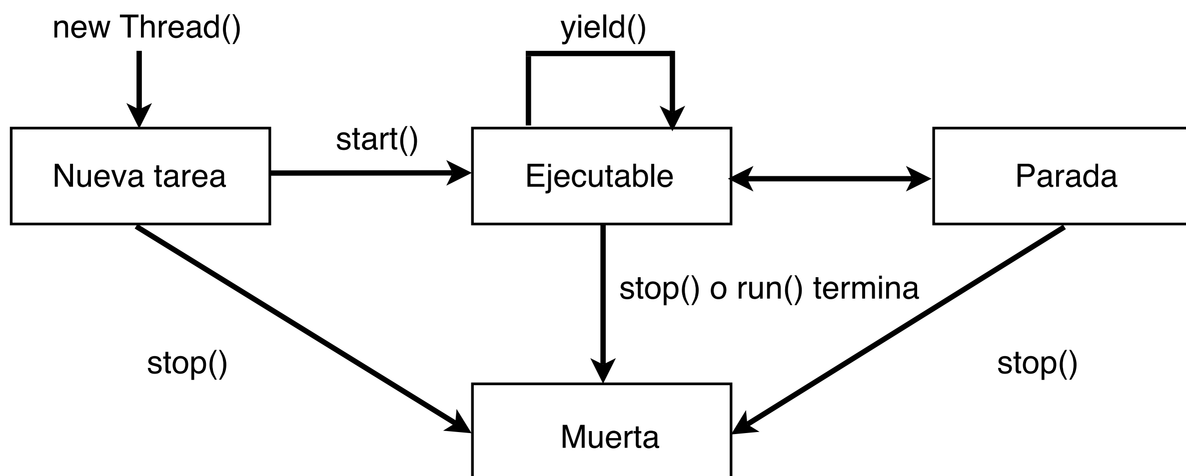
Y como métodos normales que se heredan:

- void **start** () → Este método invoca la creación de un nuevo hilo y una vez creado invoca la función run().
- void **run** () → Es el único método que hay en la interfaz Runnable y hay que sobreescribirlo para que el hilo haga algo.
- void **stop** () → Detiene la ejecución de un hilo y la destruye.

- void **suspend** () → Para el hilo pero no lo destruye, para poder ser reanudado en un futuro con la función resume ().
- void **resume** () → Reanuda la ejecución de un hilo suspendido.
- boolean **isAlive** () → Indica si el hilo está activo o pausado.
- void **setPriority** (int level) → Cambia la prioridad de un hilo. En Thread están MIN_PRIORITY = 1, NORM_PRIORITY = 5, MAX_PRIORITY = 10, como mínimos y máximos a indicar.
- int **getPriority** () → Devuelve la prioridad del hilo.
- void **setName** (String name) → Le pone un nombre al hilo.
- String **getName** () → Devuelve el nombre del hilo.

Otro tema es la creación de grupos de tareas con la clase **`ThreadGroup`** y también con la función **`setThreadGroup()`** de la clase Thread o en alguno de los constructores de la última.

3) Estados de una tarea:



En caso de hacer una llamada fuera de lugar (un suspend sin haber hecho un start), se lanzará una **`IllegalThreadStateException`**. Heredados desde Object están **`notify()`** y **`notifyAll()`**, que sirven para enviar una señal que permita a un hilo salir de la función **`wait()`** dentro del método de algún objeto. Por último en Java 2 el método stop() de Thread, parece que carece de implementación por lo que habrá de dotarla de ella el programador.

4) Comunicación entre tareas:

Para garantizar el buen uso y acceso de la memoria compartida, en programación hay métodos como los semáforos o los monitores. Para implementarlos en Java, necesitamos una forma de garantizar la exclusión al llamar una función y en Java esa forma consiste en definir un miembro del objeto como **`synchronized`**. Con ese modificador esta garantizada la exclusión, pero hay más, si un hilo ya ha activado el semáforo y llega a otro hilo, este tendrá que esperar. La mejor forma es llamar a **`wait()`**, que suspenderá la ejecución del hilo a la espera de algo. Ese algo en este ejemplo sería que el semáforo quede libre, pero porque hace falta algo más, está **`notify()`**, que al llamarla se enviará una señal al objeto para mirar que está atrapado en wait() y soltarlo.

Capítulo 9 - Delegación de eventos

1) Modelo de delegación:

En un entorno visual de botones y ventanas el usuario genera **eventos** y estos son recibidos por los `Listener` que son interfaces que hemos de implementar con clases nuestras, que si no necesitamos implementar todos los métodos de la interfaz, tenemos los `Adapter` que son clases de las que podemos heredar y que tienen todos los métodos de la interfaz "implementados" sin contenido. Una vez tengamos la clase que dicta que hacer frente un evento, se la añadimos al control que queramos. Con lo que las cosas quedan:

```
<Nombre del evento>{Event | Listener | Adapter }
control.add<Nombre del evento>Listener(objeto);
```

2) Eventos:

+ De bajo nivel:
  + java.util.EventObject
    + java.awt.AWTEvent
      + java.awt.event.ComponentEvent → Componente redimensionado o desplazado.
      + java.awt.event.FocusEvent → Perdida o ganancia del foco por un control.
        + java.awt.event.InputEvent
          + java.awt.event.KeyEvent → Pulsación del teclado.
          + java.awt.event.MouseEvent → El ratón se ha movido o pulsado.
        + java.awt.event.ContainerEvent
        + java.awt.event.WindowEvent
+ De alto nivel:
  + java.util.EventObject
    + java.awt.AWTEvent
      + java.awt.event.ActionEvent → Ejecución de un comando.
      + java.awt.event.AdjustmentEvent → Ajuste de un valor.
      + java.awt.event.ItemEvent → Cambio del estado de un item.
      + java.awt.event.TextEvent → Cambio de valor de un texto.

3) Receptores de eventos:

+ **Listener:** java.util.EventListener
  - Bajo nivel (java.awt.event): ComponentListener, ContainerListener, FocusListener, KeyListener, MouseListener, MouseMotionListener, WindowListener.
  - Alto nivel (java.awt.event): ActionListener, AdjustmentListener, ItemListener, TextListener.
+ **Adapter:**
  - Bajo nivel (java.awt): ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, MouseMotionAdapter, WindowAdapter.
  - Alto nivel (java.awt): ActionAdapter, AdjustmentAdapter, ItemAdapter, TextAdapter.

4) Fuentes de eventos:

Todo control (botón, ventana, etcétera) es hija de `Component` y esta permite registrar los siguientes listeners:

java.awt.Component → add<evento>Listener
evento = Component, Focus, Key, Mouse, MouseMotion.

Otros controles de java.awt con listeners de bajo nivel son:

`Container`	→ addContainerListener
`Dialog`	→ addWindowListener
`Frame`	→ addWindowListener

Y respecto los eventos de alto nivel:

`Button`	→ addActionListener
`Choice`	→ addItemListener
`Checkbox`	→ addItemListener
`CheckboxMenuItem`	→ addItemListener
`List`	→ addActionListener, addItemListener
`MenuItem`	→ addActionListener
`Scrollbar`	→ addAdjustmentListener
`TextArea`	→ addTextListener
`TextField`	→ addActionListener, addTextListener

5) Eventos generados por el usuario:

A la hora de completar los programas puede ocurrir que querramos crear un nuevo tipo de control. Para ello tendremos que heredar del objeto Component, pero eso por sí solo no recoge más que los eventos básicos, si queremos ampliar ese aspecto:
+ Habrá que tener una variable con la lista de listeners:

```
ActionListener listaActions;
```

+ Habrá que tener una función para registrar los listeners:

```
public void addActionListener (ActionListener li) {
    listaActions = AWTEventMulticaster.add(listaActions, li);
}
```

+ Por último habrá que ser capaces de generar el evento:

```
public void createActionEvent () {
    listaActions.actionPerformed(new ActionEvent(this,
        ActionEvent.ACTION_PERFORMED, strID));
}
```

También podemos crear nuevos tipos de eventos:
+ Primero se crea una clase que hereda de EventObject.
+ Segundo se crea una interfaz que hereda de EventListener.

Finalmente la clase **`Toolkit`** tiene métodos para conseguir la cola de eventos del sistema, con lo que podemos generar los eventos que querramos, en las condiciones que precisemos.

6) Eventos en Swing:

+ La librería Swing trae una lista nueva de **listeners**: Ancestor, Caret, CellEditor, Change, Document, Hyperlink, InternalFrame, ListData, ListSelection, Menu, PopMenu, TableColumnModel, TableModel, TreeExpansion, TreeModel, TreeSelection, UndoableEdit.
+ Respecto a los **eventos**: Ancestor, Caret, Change, Document, Hyperlink, ListData, ListSelection, Menu, PopMenu, TableColumnModel, TableModel, TreeExpansion, TreeModel, TreeSelection, UndoableEdit.
+ Y luego hay otros elementos como: EventListenerList, InternalFrameAdapter.

Capítulo 10 - AWT

1) Componentes:

La clase padre de todo control es `Component` y de esta surgen: Button, Canvas, Checkbox, Choice, Label, List, Scrollbar, TextComponent (TextArea, TextField), Container (Panel, Window (Dialog, Frame)).

Cada componente se comporta de un modo en particular y sus principales métodos según el control son:
+ **Button:** addActionListener, removeActionListener, getLabel, setLabel.
+ **Choice:** add, select, addItemListener.
+ **CheckboxGroup:** getSelectedCheckbox.
+ **Checkbox:** getName, getLabel.
+ **List:** add, setMultipleMode, select, addActionListener, getSelectedItem.
+ **TextField:** addActionListener, setText, getText, getSelectedItem.
+ **TextArea:** addTextListener, append, setText, getText.
 (SCROLLBARS_{BOTH | NONE | HORIZONTAL_ONLY | VERTICAL_ONLY})
+ **Label:** Métodos de alineación del texto.
+ **Canvas:** minimumSize, preferredSize, addNotify, paint, setBackground.
+ **Scrollbar:** getValue, addAdjustmentListener. (VERTICAL, HORIZONTAL)

2) Contenedores:

+ **Frame:** setLayout, add, setSize, setVisible, addWindowListener.
+ **Dialog:** setTittle, show, setSize, add.
+ **Panel:** setLayout, setBackground, add.

3) Menús:

+ **MenuComponent:** La clase padre de todos los controles de menú.
+ **Menu:** Es el control que almacena los menús típicos desplegables.
+ **MenuItem:** addActionListener, setEnabled, isEnabled.
+ **MenuShortcut:** Sirve para definir una tecla de acceso rápido (ver KeyEvent).
+ **MenuBar:** Es la barra de menús y almacena los objetos Menu.
+ **CheckboxMenuItem:** addItemListener.
+ **PopupMenu:** Almacena items de menú para hacer un menú emergente.

4) Layouts:

+ **FlowLayout:** Es el layout que suele estar por defecto. Y los va colocando por orden (de izquierda a derecha). Aunque con la función `void setBounds (Rectangle area)` de cualquier control que se pueda añadir a un contenedor (botones, cajas, barras, etcétera), podemos indicar el tamaño que deseamos para este. En el constructor podemos indicar la alineación y el espacio de separación. Esas propiedades se pueden cambiar (si lo cambiamos en el objeto creado, hemos de pasarle al frame el layout con setLayout() otra vez y llamar a validate(), para efectuar los cambios), por ejemplo la separación es con `setHgap` y `setVgap`, y se obtienen con `getHgap` y `getVgap`.
+ **BorderLayout:** La distribución de este esquema es de 5 huecos, que a la hora de usar en la función `add()` de contenedores como Frame, hemos de indicar el control y con una cadena lo siguiente: "`North`", "`West`", "`Center`", "`East`" o "`South`". En este esquema, en el constructor podemos indicar el Vgap y el Hgap, y por supuesto cambiarlo luego.
+ **GridLayout:** En este esquema en el constructor indicamos un número de filas y uno de columnas, pues este crea una "rejilla", que a medida que vamos metiendo controles en un contenedor, van asignandose de izquierda a derecha, de arriba a abajo. Con `setRows` y `setColumns` se pueden cambiar esas propiedades.
+ **GridBagLayout:** Este esquema usa la clase `GridBagConstraints` para definir el tamaño y configuración en la rejilla, que toman los controles. Antes de hacer add con el control, hay que llamar a setConstraints(control, gbc), desde el objeto GridBagLayout creado.
+ **CardLayout:** Se suele usar cuando es necesaria una zona de la ventana que permita colocar distintos componentes sobre ella.
+ **Posicionamiento absoluto:** Si hacemos setLayout(null), podremos con el método setBounds de los controles, indicar donde ponerlos.
+ **BoxLayout:** Incorporado con Swing, permite colocar un control en una posición x, y e indicar el tamaño que ocupen.
+ **OverlayLayout:** Incorporado con Swing, y se dimensiona para contener el control más grande y superpone cada uno sobre los otros.

5) Imprimir con AWT:

En el objeto `Toolkit` el método `getPrintJob()` es la puerta al manejo de la impresora en este tema. Muchos de los controles tienen un método `printAll()` que necesita un objeto `Graphics`, que nos devolvería el objeto `PrintJob` con `getGraphics()`.

Capítulo 11 - Swing

1) javax.swing.*:

Con Java 1.2 se introdujo Swing, una nueva librería para manejar la GUI. El cambio desde AWT en el mejor de los casos es poner una J delante del nombre de la clase. Algunos controles son: JFrame, JDialog, JPanel, JButton, JCanvas, JCheckBox, JLabel, JList, JTextField, JTextArea. Pero por ejemplo JFrame ya no tiene el método add, sino `getContentPane().add(control [, orientación])`. El estilo visual de esta GUI se llama Metal.

(Nota: Ver el Swing Tutorial de Sun para más información.)

2) Diferencias remarcables:

+ **Bordes:** JComponent tiene un método llamado **`setBorder()`**, para modificar el aspecto de este mediante las clases: TitledBorder, EtchedBorder, LineBorder, MatteBorder, BevelBorder, SoftBevelBorder, CompoundBorder.
+ **Etiquetas:** El JLabel es un control muy elemental para mostrar texto no editable por el usuario. Con la clase **`Icon`** podemos cargar un **`ImageIcon`** y añadirlo a la etiqueta.
+ **Botones:** **`JButton`** es hija de **`AbstractButton`**, igual que **`JToggleButton`**, **`JCheckBox`** y **`JRadioButton`**.
+ **Grupos de botones:** Para agrupar botones se utiliza la clase **`ButtonGroup`**.
+ **Listas y cagas combinadas:** Para crear una lista tenemos **`JList`**, pero hemos de añadirlo en un **`JScrollPane`**, para que tenga barra de desplazamiento. A parte de texto, también se pueden añadir botones, cajas de texto y gráficos. **`JComboBox`** sirve de sustituto de Choice.
+ **Texto:** En Swing tenemos **`JTextField`**, **`JPasswordField`**, **`JTextArea`** (que para tener barras de desplazamiento requiere de un JScrollPane), **`JTextPane`** (que permite texto de diferentes fuentes, colores y tamaños). Para manejar el tema del estilo tenemos el **`DefaultStyleDocument`**, que es un tipo que **`StyleContext`** genera devolviendo objetos de tipo **`Style`**. Estos objetos de tipo Style nos sirven para la función de JTextPane **`setCharacterAttributes()`**. Para crear un objeto Style con StyleContext hay que llamar a **`addStyle()`** y con el objeto **`StyleConstants`** se pueden modificar las propiedades del texto.
+ **Tool tips:** Las clases que heredan de JComponent obtienen el método **`setToolTipText(String mensaje)`**, para poder mostrar un cartelito de ayuda sobre cada control.
+ **Iconos:** Con el objeto **`Icon`** podemos añadir una imagen a los JLabel y los controles que derivan de AbstractButton, con **`setIcon()`**. Para cargar un Icon se crea un nuevo objeto **`ImageIcon`**.
+ **Menús:** Tenemos los objetos JMenuBar, JMenu, JMenuItem, JCheckBoxMenuItem, JRadioButtonMenuItem.
+ **Menús popup:** Para hacer funcionar un **`JPopupMenu`** hay en JPanel que llamar a **`enableEvents()`** y luego en el **`MouseEvent`** preguntarle **`isPopupTrigger()`**, para llamar al método **`show()`** del popup (ver el Swing Tutorial para ver la última versión).
+ **Barras de progreso:** Swing tiene la clase **`JProgressBar`** y **`JSlider`**, para tener una barra de progreso y un deslizador.
+ **Árboles:** Para esto se usa el control **`JTree`**, pero siendo un control tan sofisticado hay clases de apoyo como **`DefaultMutableTreeNode`**, para hacer menos dramática la situación. JTree no necesita que se declare un JScrollPane, porque lo lleva incorporado al crearse.
+ **Tablas:** Para crear esto tenemos **`JTable`**, que maneja su contenido mediante objetos **`TableModel`**, interfaz para manejar el asunto, que está ya implementada en **`AbstractTableModel`**, y se recomienda heredar de esta. La clase con los datos que creemos, se la tenemos que pasar al constructor de JTable. Y este control necesita un JScrollPane para tener barras de desplazamiento. Hay otro método para los datos y es crear dos array de tipo String, uno de una dimensión (para el nombre de las columnas) y otro de dos dimensiones (para los datos), y pasárselos al constructor de JTable.
+ **Pestañas:** Este control se crea con **`JTabbedPane`** y con el método **`addTab(String tabName, JPanel panel)`** se añade una nueva pestaña. Con **`setSelectedIndex()`** se activa la pestaña para ser vista (javax.swing.border.*).

+ **Selector de ficheros:** Para abrir un cuadro de diálogo de selección de ficheros se usa la clase **`JFileChooser`** y para cuadros de diálogo de confirmación se emplea el método estático **`JOptionPane.showConfirmDialog()`**.
+ **Teclado:** La clase **`JRootPane`** tiene un método para registrar una tecla a este con **`registerKeyboardAction()`**, que necesita de un objeto de tipo **`KeyStroke`**. Los eventos van al método actionPerformed de la interfaz ActionListener.
+ **Paneles desplazables:** Para añadir barras de desplazamiento se usa **`JScrollPane`**, pero si queremos partir la pantalla en dos existe **`JSplitPane`**. Puede ser vertical u horizontal, y tiene **`setLeftComponent()`** y **`setRightComponent()`** para añadir paneles. JSplitPane se agrega como si fuera un panel. JScrollPane usa **`getViewport()`** para llamar a **`add()`** y a **`addChangeListener()`**, lo primero para añadir el control y lo segundo para actualizar los tamaños de los JScrollPane con el evento Change.

Capítulo 12 - Gráficos

1) Las coordenadas:

Como en toda API 2D la esquina superior izquierda es el punto 0, 0 y la otra punta el x, y. A la hora de pintar en un componente hay que saber su alto y anchura, pero además de **`getWidth()`** y **`getHeight()`** hemos de calcular el borde con:

```
Insets getInsets ();
```

Esto nos devuelve el grosor de cada borde del control.

2) Pintado en AWT:

Hay dos formas de que se actualice un control, por el sistema:
- El control se hace visible en pantalla por primera vez.
- El control cambia de tamaño.
- El control estaba parcialmente oculto y vuelve a ser visible.

O que la propia aplicación controle ese aspecto.
+ **El método paint():**

```
public void paint (Graphics g) { ... }
```

Esta es la función que pinta el componente y lo hace con el objeto **`Graphics`**, que el control configura antes de llamar a paint, pasándole el foreground color, la fuente, las coordenadas y el tamaño de la región de pintado del control. Por eso no se llama nunca a paint directamente sino a **`repaint()`** y sus variantes:

```
void repaint ([int x, int y, int width, int height]);
void repaint (long tm [, int x, int y, int width, int height]);
```

+ **Componentes Lightweight:**
Si creamos una nueva versión de un Container, hemos de llamarlo primero a super.paint(g) y luego los hijos.

3) Pintado en Swing:

+ **Soporte de Doble Buffer:**

Por defecto es algo que está activado, pero desde JComponent se incluyen funciones para manipularlo:
- boolean isDoubleBuffered ()
- void setDoubleBuffered (boolean o)

+ **Opacidad:**
Es una propiedad de todo heredero de JComponent, que por defecto está a true, para que se pinte el control entero.
- boolean isOpaque ()
- void setOpaque (boolean o)

+ **Redibujo "optimizado":**
Swing le cuesta controlar la superposición de componentes y existe desde JComponent un método llamado `isOptimizedDrawingEnabled()`, que si devuelve true, indica que no tiene ningún control solapado.

+ **Los métodos paint:**
Los controles "ligeros" (los que contienen a otros), cuando hacen un paint(), llama a unas funciones en un orden:
```
void paintComponent (Graphics g)
void paintBorder (Graphics g)
void paintChildren (Graphics g)
```
Por eso se recomienda solo reescribir paintComponent() y no paint para el correcto funcionamiento del control.

+ **Repintado e interfaz de usuario:**
Cuando creamos un nuevo control a partir de JComponent, con sobreescribir paint() funciona bien, pero heredando ya de otros controles con aspecto, su ciclo de pintado consiste en:
paint() → paintComponent() → if(ui != null) ui.update() → ui.paint()
Por esto parece lógico sobreescribir paintComponent() y llamar a super.paintComponent(g).

+ **Repintado síncrono:**
Para repintar un componente podemos llamar al método de JComponent `repaint()` y esperar a que se digne en hacerlo. O podemos directamente llamar a `paintInmediately()`, para que actualice en el momento, pero hay que tener cuidado con esto pues podría saturar considerablemente la ejecución del programa.

+ **La clase RepaintManager:**
Se encarga de gestionar las peticiones de repaint(), de la forma más eficiente que puede. Se puede conseguir la instancia creada de este y modificar sus propiedades.

4) La clase Graphics:

Esta clase representa el contexto gráfico de una aplicación, el modo de acceder al buffer de pantalla. Hay clases que funcionan sobre Graphics para realizar cosas específicas, como **`Rectangle`** o **`Polygon`**. Siendo Graphics una clase abstracta no puede instanciarse:
+ **Métodos generales:**

- **clearRect** (int, int, int, int) → Borra una región con el color de fondo del control establecido con setBackground().
- **copyArea** (int, int, int, int, int, int) → Copia una región (indicada por los 4 primeros parámetros) a las coordenadas que le marquemos en los dos últimos parámetros.
- **create** () → Devuelve una copia del objeto Graphics.
- **dispose** () → Cuando creamos un nuevo objeto Graphics, debemos llamar a esta función para liberar sus recursos, una vez que no lo necesitemos más.
- **finalize** () → Elimina el contexto gráfico cuando no queda ninguna referencia, parecido a dispose.
- **getColor** () → Devuelve el color actual en un objeto `Color`.
- **setColor** (Color) → Cambia el color actual para el pintado.
- **setPaintMode** () → Cambia la forma de pintar del contexto.
- **setXORMode** (Color) → Esta función cambia el modo de pintar algo en pantalla aplicándole un xor a cada pixel que mostremos en pantalla. Para quitar este modo hay que llamar a setPaintMode().
- **translate** (int, int) → Cambia el punto de origen del contexto.

+ **Métodos para pintar figuras:** (Ver documentación para los parámetros.)
  - **drawLine** () → Dibuja una linea.
  - **drawPolyline** () → Dibuja una secuencia de lineas.
  - **drawRect** () → Dibuja un rectángulo.
  - **fillRect** () → Dibuja un rectángulo con relleno.
  - **drawRoundRect** () → Dibuja un rectángulo de esquinas redondeadas.
  - **fillRoundRect** () → Dibuja un rectángulo de esquinas redondeadas con relleno.
  - **draw3DRect** () → Dibuja un rectángulo 3D.
  - **fill3DRect** () → Dibuja un rectángulo 3D con relleno.
  - **drawOval** () → Dibuja una elipse.
  - **fillOval** () → Dibuja una elipse con relleno.
  - **drawArc** () → Dibuja un arco.
  - **fillArc** () → Dibuja un arco con relleno.
  - **drawPolygon** () → Dibuja un objeto Polygon.
  - **fillPolygon** () → Dibuja un objeto Polygon con relleno.

+ **Métodos para pintar texto:**
  - **drawString** (String, int, int) → Pinta una cadena de texto en la pantalla.
  - **drawChars** (char[], int, int, int) → Pinta una cadena de texto en la pantalla.
  - **getFont** (), **setFont** (Font) → Para manipular la fuente actual en uso o sus propiedades tenemos estas funciones.
  - **getFontMetrics** ([Font]) → Sirven para obtener un objeto de tipo FontMetrics con las medidas de la fuente actual o la que le indiquemos.

+ **La clase Font:**

  Para crear una fuente en el constructor hay que pasarle el nombre de esta, el estilo y el tamaño. Para conocer las fuentes disponibles en el sistema hemos de invocar al método `getAllFonts()`, de la clase `GraphicsEnvironment`. El método deriveFont() devuelve la fuente con el estilo y el tamaño modificado si así se lo indicamos.

+ **La clase FontMetrics:**

Esta almacena las medidas de una fuente dada. Entre sus métodos hay: getSize(), getHeight(), stringWidth(), etcétera.

+ **Métodos de clipping:** (Ver parámetros en la documentación.)
- clipRect () → Delimita la zona de pintado.
- getClip () → Devuelve la zona de pintado.
- getClipBounds () → Devuelve la zona de pintado de los bordes.
- setClip () → Cambia la zona de pintado.

+ **La clase Color:**

Es la clase que representa el tema de un color y sus componentes RGB. De forma estática hay unas constantes para los colores básicos, evitando al usuario tener que crear un objeto Color para ese caso. Entre sus métodos se encuentran:
- **getRed** (), **getGreen** (), **getBlue** () → Devuelve un componente.
- **getRGB** () → Devuelve el color en formato 0xFFRRGGBB.
- **brighter** () → Crea una versión más brillante del color.
- **darker** () → Crea una versión más oscura del color.
- **decode** (String) → Decodifica la cadena devolviendo el color en un int.

+ **Las imagenes:**

Graphics para pintar una imagen tiene `drawImage()`, que requiere un objeto de tipo `Image`. Esta clase es abstracta, con lo que para cargar una imagen se necesita llamar a `getImage()` desde la clase `Applet` o `Toolkit`. Image tiene diversos métodos:
- **getScaledInstance** (int, int, int) → Crea una versión escalada de la imagen. (SCALE_{AREA_AVERAGING | FAST | SMOOTH | REPLICATE | DEFAULT })
- **flush** () → Libera los recursos tomados por la imagen.
- **getGraphics** () → Crea un contexto gráfico para pintar en una imagen.
- **getHeight** (ImageObserver) → Alto de la imagen.
- **getWidth** (ImageObserver) → Ancho de la imagen.
- **getProperty** (String, ImageObserver) → Devuelve una propiedad.
- **getSource** () → Devuelve una referencia a los pixels de la imagen.

Otra forma de crear una imagen es con `createImage()`, de Component o Toolkit. Java trae la clase `MediaTracker`, que sirve para almacenar y gestionar colecciones de imagenes. `ImageObserver` es una interfaz que los componentes de AWT y Swing implementan. Otras clases de Java para temas relacionados con el pintado avanzado de imagenes son: ColorModel, IndexColorModel, DirectColorModel, FilteredImageSource, ImageFilter, MemoryImageSource, PixelGrabber.

5) <u>Nota final</u>:

Recordemos al pintar en paint(), que los componentes tienen bordes que se obtienen sus tamaños con `getInsets()`, devolviendo un objeto de tipo `Insets` (left, top, right, bottom como propiedades). Por ello se recomienda al pintar, llamar a:

```
g.translate(getInsets().left, getInsets().top);
```

Para cambiar el origen de coordenadas del componente. Luego para obtener el ancho y el alto sería:

```
alto = getHeight() - getInsets().top - getInsets().bottom;
ancho = getWidth() - getInsets().left - getInsets().right;
```

Y así tendremos las auténticas propiedades reales del area de pintado del componente.

Capítulo 13 - Comunicaciones en red (java.net.*)

1) La clase InetAddress:

Es una clase para traducir una url en una ip. Si metemos una dirección url inválida, lanzará una **UnknownHostException**.

2) La clase URL:

Es una clase de alto nivel para acceder a recursos en internet. Además también se usan para esto URLEncoder y URLConnection. **URL** almacena la dirección de internet y el protocolo. **URLEncoder** codifica cadenas de tal forma que no tengan caracteres no válidos para URL. Y **URLConnection** establece la conexión al lugar apuntado por URL (url.openConnection()), y con URL podemos abrir un stream de entrada:

```
BufferedReader paginaHtml = new BufferedReader(new
        InputStreamReader(url.openStream()));
```

Mientras accedemos a propiedades de más alto nivel con URLConnection del recurso apuntado por URL.

3) La clase Socket: (java.io.*, java.util.*)

Hay dos formas de afrontar una aplicación de red:

+ **TCP:** El cliente crea un **Socket** conectándose al **ServerSocket** del servidor, este acepta la petición y crea un socket para atender al cliente de forma individual.
+ **UDP:** El cliente crea un **DatagramSocket** y cada vez que quiere mandar algo, construye un **DatagramPacket** y lo envía. El servidor tiene un DatagramSocket asignado a un puerto en particular, recibe el paquete y crea otro DatagramSocket para atender al cliente.

De ahí que con el Socket se hagan cosas para TCP. Al crear uno hemos de indicar la ip y el puerto de destinos:

```
Socket soc = new Socket("www.miserver.es", 4000);
```

La ip también se podría poner con el formato "192.168.1.1". En caso de no crearse se lanzaría UnknownHostException o una IOException en cuanto se intentara usar. Para poder enviar algo hay que abrir un stream de salida, con cualquier clase **Writer**:

```
PrintWriter pw = new PrintWriter(new OutputStreamWriter(
        soc.getOutputStream()), true);
```

Y para leer necesitaríamos un stream de entrada, usando cualquiera de las clases **Reader**:

```
BufferedReader br = new BufferedReader(new InputStreamReader(
        soc.getInputStream()));
```

Y cuando ya no fueramos a hacer nada más con el Socket, lo cerraríamos:

```
soc.close();
```

4) La clase ServerSocket:

Se crea indicando solo el puerto con un entero (int):

```
ServerSocket ss = new ServerSocket(puerto);
```

Y luego a esperar las peticiones de conexión con el método:

```
Socket accept ();
```

Normalmente se crean hilos para atender a los clientes.

5) La clase DatagramPacket:

Esta clase almacena los datos de un paquete a enviar:

```
DatagramPacket env = new DatagramPacket(msj, tam, ip, puerto);
//DatagramPacket(byte[] msj, int tam, InetAddress ip, int puerto);
```

O para recibirlos:

```
DatagramPacket rec = new DatagramPacket(buffer, tamaño);
```

Hay que tener en cuenta que un paquete no puede llegar a ocupar 64KB (como mucho 65535 bytes). Si al recibir un paquete, nos llegan más datos de los que soporta el buffer, se lanzará una **`IllegalArgumentException`**. Esta clase tiene por métodos:

- **getAddress** () → La dirección destino o fuente.
- **getPort** () → Puerto del destino o fuente.
- **getData** () → Buffer de datos del paquete.
- **getLength** () → Tamaño del buffer.

También tiene por cada get expuesto un set.

6) La clase DatagramSocket:

Para crear un socket UDP sería:

```
DatagramSocket dsoc = new DatagramSocket([puerto]);
```

Para enviar:

```
dsoc.send(paquete);
```

Para recibir:

```
dsoc.receive(paquete);
```

Y para cerrarlo:

```
dsoc.close();
```

J2ME

Índice

Capítulo 1 - Introducción 30
Capítulo 2 - Lo más básico 31
Capítulo 3 - MIDP 2.0 33
Capítulo 4 - Almacenamiento con RMS 35

Capítulo 1 - Introducción

1) Restricciones y cambios frente a J2SE:

Dadas las limitaciones de los dispositivos móviles J2ME no tiene:
+ Soporte para números reales (float y double).
+ Soporte para el método finalize ().
+ Todas las clases de excepciones de J2SE.

Pero además de quitar algunas librerías en J2ME se añadieron:
+ javax.microedition:
  + .midlet → Base del programa.
  + .lcdui → Interfaz "gráfica".
  + .rms → Almacenamiento.

2) Hola mundo:

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class HolaMundo extends MIDlet implements CommandListener {
	private Command exitCommand;
	private Display display;
	private Form screen;
	public HolaMundo () {
		display = Display.getDisplay(this);
		exitCommand = new Command("Salir", Command.EXIT, 2);
		screen = new Form("HelloWorld");
		screen.append(new StringItem("", "Hola mundo..."));
		screen.addCommand(exitCommand);
		screen.setCommandListener(this);
	}
	public void startApp () throws MIDletStateChangeException {
		display.setCurrent(screen);
	}
	public void pauseApp () {}
	public void destroyApp (boolean incondicional) {}
	public void commandAction (Command c, Displayable s) {
		if(c == exitCommand) {
			destroyApp(false);
			notifyDestroyed();
		}
	}
}
```

3) Un poco de anatomía:

Todo MIDlet tiene 2 estados:

startApp() → **Ejecución** → pauseApp() → **Pausado**
↓
destroyApp()

Capítulo 2 - Lo más básico

1) Las librerías básicas:

De J2SE se conserva parcialmente los paquetes:

java. → lang, util, io.

Además de las clases Timer y TimerTask. En cuanto a los paquetes de J2ME (javax.microedition): midlet, lcdui, io, rms.

2) Algunas aclaraciones del Hola mundo:

Con **`getDisplay(this)`** se le indica que el programa contenido en el MIDlet es lo que tiene que mostrar. La clase **`Command`** sirve para registrar un tipo de evento de la aplicación, que tras ser creado se le pasará a addCommand(). La función setCommandListener() sirve para establecer la clase que recibe los eventos creados por los comandos registrados.

3) La interfaz a bajo nivel: (MIDP 1.0)

La API tiene una clase **`Canvas`** de la que se puede heredar para gestionar el pintado de la aplicación, en vez de usar un objeto Form. La creación sería parecida:

```
// Declaración: private Juego screen;
screen = new Juego();
screen.addCommand(exitCommand);
screen.setCommandListener(this);
```

Y para definir una clase derivada sería:

```
class Juego extends Canvas {
      public void paint (Graphics g) {
            g.setColor(255, 255, 255);
            g.fillRect(0, 0, getWidth(), getHeight());
      }
}
```

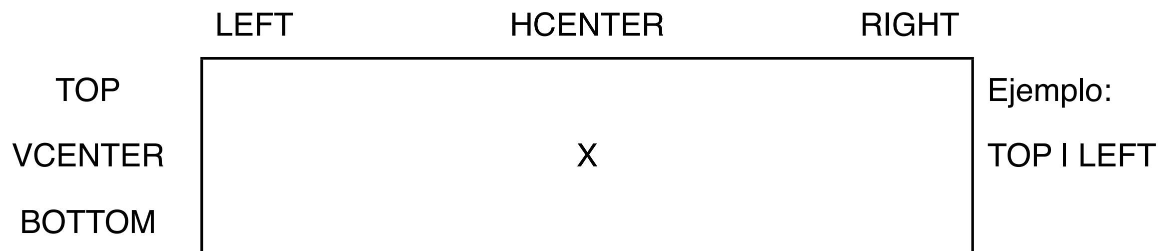
El objeto **`Graphics`** tiene muchos métodos para gestionar con qué color pintar y con qué formas, con relleno (fill) o sin él (draw). Entre ellas están: Line, Rect, RoundRect, Arc. Para el color está setColor() y setGrayScale().

4) Texto e imágenes:

Pero un juego necesita poder pintar texto e imagenes. Para el texto tenemos en Graphics:

void **drawString** (String texto, int x, int y, int ancla);

El "ancla" sirve para variar el origen de coordenadas y Graphics tiene una serie de constantes que se pueden complementar con el operador | (or binario):



También existe la constante Graphics.BASELINE, que posiciona el origen en la base de la linea de texto. También es útil cambiar la fuente con:

void **setFont** (Font fuente);

Para conseguir una fuente en la clase Font:

static Font **getFont** (int espaciado, int estilo, int tamaño);
+ tamaño: SIZE_SMALL, SIZE_MEDIUM, SIZE_LARGE.
+ estilo: STYLE_{PLAIN | ITALIC | BOLD | UNDERLINED }
+ espaciado: FACE_{SYSTEM | MONOSPACE | PROPORTIONAL }

Todas esas constantes son de Font. Lo siguiente son las imagenes, que para crearlas se hace:

void **drawImage** (Image img, int x, int y, int ancla);
void **drawRegion** (Image src, int xsrc, int ysrc, int wsrc, int hsrc, int transform, int xdst, int ydst, int anchor);
+ transform: Sprite.TRANS_{NONE | ROT90 | ROT180 | ROT270 | MIRROR | MIRROR90 | MIRROR180 | MIRROR270 }

Nota: Los recursos (imagenes, sonidos, etcétera) se guardan en la carpeta "res".

5) Lectura del teclado:

La clase `Canvas` tiene unos métodos específicos para capturar la entrada por el teclado del móvil, que podemos sobreescribir:

public void **keyPressed** (int keyCode) {}
public void **keyReleased** (int keyCode) {}
public void **keyRepeated** (int keyCode) {}

Una vez dentro hay que recoger la tecla que se supone pulsada:

int action = **getGameAction**(keyCode);

De ahí obtenemos alguna de las siguientes constantes:
+ KEY_NUM# (# = 0..9) → Teclas numéricas.
+ KEY_POUND → Tecla almohadilla.
+ KEY_STAR → Tecla asterisco.
+ GAME_# (# = A, B, C, D) → Teclas especiales de juego.
+ UP, DOWN, LEFT, RIGHT → Arriba, abajo, izquierda, derecha.
+ FIRE → Disparo.

6) Threads:

Para crear un hilo primero necesitamos una clase que tenga un método run() y que implemente la interfaz `Runnable`:

```
public class Hilo implements Runnable {
	public void run () {
		//...
	}
}
```

Una vez tenemos esto, lo siguiente es en algún método:

```
Thread hilo = new Thread(new Hilo());
hilo.start();
```

La clase Thread además trae el método **stop()**, para terminar la ejecución de un hilo, y la función estática **sleep(int time)** para dormir unos milisegundos la ejecución del juego.

7) El game loop:

Una versión simple del bucle de juego sería el siguiente, que iría en el método run() del nuevo hilo que creamos para gestionar la lógica del juego, separada del hilo de la JVM donde llama a paint():

```
public void run () {
	iniciar();
	while(true) {
		logica();  // Lógica del juego.
		repaint(); // Actualizar la pantalla.
		serviceRepaints();

		try { // Un descanso ligero...
			Thread.sleep(time);
		} catch (InterruptedException e) {
			System.out.println(e.toString());
		}
	}
}
```

8) Sonidos:

El soporte bajo MIDP 1.0 es nulo por ello en la 2.0 se añadió un poco de soporte para este tema.

Capítulo 3 - MIDP 2.0

1) Sonidos:

El formato soportado en J2ME es el wav, para cargar uno tan solo hay que hacer:

```
InputStream in = getClass().getResourceAsStream("/sonido.wav");
Player sonido = Manager.createPlayer(in, "audio/x-wav");
```

Al crearlo podrían ser lanzadas **IOException** o **MediaException**. Y para reproducir el sonido sería:

```
try {
    sonido.start();
} catch (MediaException me) {}
```

2) Música:

También podemos reproducir nótas musicales, llamadas tonos:

```
try {
    Manager.playTone(ToneControl.C4, 100, 80);
    // El primer argumento es la nota.
    // El segundo argumento es la duración en ms.
    // El tercer argumento es el volumen.
} catch (Exception e) {}
```

Para el silencio es ToneControl.SILENCE, pero esta forma podría resultar engorrosa, así que la otra forma consiste en tener primero:

```
byte secuencia = {
    ToneControl.VERSION, 1,
    ToneControl.TEMPO, tempo,
    // Crear un bloque
    ToneControl.BLOCK_START, 0,
    nota, duración, ...,
    ToneControl.BLOCK_END, 0,
    // Reproducir un bloque
    ToneControl.PLAY_BLOCK, 0,
    ...
};
```

La secuencia musical que reproduciremos con:

```
Player p = Manager.createPlayer(Manager.TONE_DEVICE_LOCATOR);
p.realize();
ToneControl c = (ToneControl) p.getControl("ToneControl");
c.setSequence(secuencia);
p.start();
```

3) El GameCanvas:

Esta nueva clase permite que el juego tenga doble buffer y que el pintado esté en el mismo hilo sin llamar a repaint():

```
public class Juego extends GameCanvas implements Runnable {
    //...
    private boolean done;
    public void run () {
        long start, end; int duration;
        Graphics g = getGraphics();
        while(!done) {
            start = System.currentTimeMillis();
```

```
                    // Lógica → int keyStates = getKeyStates();
                    this.input();
                    // Pintado → flushGraphics();
                    this.render(g);
                    end = System.currentTimeMillis();
                    duration = (int) (end - start);
                    if(duration < tiempoFrame) {
                        try {
                            Thread.sleep(tiempoFrame - duration);
                        } catch (InterruptedException ie) {
                            done = true;
                        }
                    }
                }
            }
            //...
        }
```

Los nuevos métodos que trae esta clase son:

+ Graphics **getGraphics** () → Devuelve el contexto de pintado.
+ void **flushGraphics** () → Actualiza la pantalla.
+ int **getKeyStates** () → Devuelve las teclas pulsadas y para ello el entero que recibimos le aplicamos un and lógico (&), junto a las nuevas constantes de tecla, para saber si está pulsada. Las nuevas constantes son #_PRESSED (# = FIRE, DOWN, LEFT, RIGHT, UP, GAME_# (# = A, B, C, D), KEY_NUM# (# = 0..9), KEY_POUND, KEY_STAR).

Capítulo 4 - Almacenamiento con RMS

1) Abrir y cerrar un RecordStore:

static RecordStore **openRecordStore** (String nombre, boolean crear);
void **closeRecordStore** ();
static void **deleteRecordStore** (String nombre);

El nombre indica el "fichero" que queremos abrir, con crear a true, si el "fichero" no existe se crea uno nuevo.

2) Añadir registros:

int **addRecord** (byte[] dato, int offset, int numbytes);
+ offset: Posición en el "fichero" donde se meterá el dato.
+ numbytes: Número de bytes que ocupa dato.

Este método puede lanzar un **`RecordStoreException`**.

3) Leer registros:

byte[] **getRecord** (int id);

Al añadir un registro se nos devuelve un entero, ese es el "id" para referenciarlo posteriormente. Pero para saber cuales hay, existe otro método:

```
RecordEnumeration regsId = null;
try {
    regsId = fichero.enumerateRecords(null, null, false);
    while(regsId.hasNextElement())
        System.out.println(regsId.nextRecordId());
} catch (Exception e) {}
```

El método getRecord() también puede lanzar una RecordStoreException.

4) Borrar registros:

void **deleteRecord** (int id);

También puede lanzar una RecordStoreException.