PROGRAMACIÓN

**Temario**

**Módulo 1. Contexto e introducción al curso**

* 1. ¿Qué es la programación y en qué sitios está presente?
  2. Distintas modalidades y carreras profesionales
  3. Concepto general de Programación Estructurada
  4. Concepto general de Programación Orientada a Objetos
  5. Programación Estructurada vs Programación Orientada a Objetos
  6. El proyecto CODE.ORG

**Módulo 2: POO. Conceptos básicos I**

2.1 Variables primitivas

2.2 Métodos

2.3 Clases

2.4 Arrays y otros componentes

2.5 Funciones de cadena

2.6 Estructuras de control de flujo

2.7 Condicional if

2.8 Condicional switch

2.9 Bucle while

2.10 Bucle for clásico

2.11 Bucle foreach

2.12 Los errores que se pueden producir en un programa y su control

2.13 Programación básica de Applets

**3. Módulo 3. POO. Conceptos básicos II**

* 1. Instancias y métodos constructores
  2. Interfaces
  3. Polimorfismo
  4. Herencia
  5. Encapsulación. Niveles de acceso
  6. Modificador de acceso static
  7. Ejecución de constructores en la Herencia. Uso de super y this.

**4. Módulo 4. Las actuales Tecnologías y sus Lenguajes de Programación**

* 1. Plataforma JAVA-J2EE
  2. Plataforma .NET
  3. Tecnologías Móviles

**MÓDULO 1. CONTEXTO E INTRODUCCIÓN AL CURSO**

**1.1 ¿Qué es la programación y en qué sitios está presente?**

Podríamos definir la programación como aquella parte de la informática encargada de dar instrucciones al ordenador para realizar una serie de tareas. Todo este conjunto de instrucciones en sí es lo que se denomina **programa.**

Entre las tareas que puede desarrollar un programa se encuentran:

* La manipulación de datos (Borrar datos, modificar datos…)
* Cálculos numéricos
* Facilitar los problemas y sistemas de gestión
* Realizar funciones multimedia (Animación de gráficos [videojuegos], edición de imágenes, sonido y videos…)
* Etc…

Hoy en día la programación se encuentra presente en nuestro que hacer diario ,por ejemplo si queremos reservar una entrada de cine a través de internet por detrás se encuentran los programas que realizan esta tarea, cuando vamos al dentista y nos realizan una radiografía esto es controlado por programas informáticos, cuando realizamos una trasferencia de dinero en un banco o bien a través de internet también son procesos que están manejados por programas. En definitiva cada día más, sucede que la programación y lo que son sus unidades centrales( los programas) se hacen más presentes en nuestras vidas y nos ayudan a mejorar la interactividad con todo lo que nos rodea.



Imgm1\_1

La medicina al ser una ciencia que está en continuo movimiento de avance , está cada día más estrechamente ligada a la informática siendo esta una gran herramienta de apoyo y ayuda hasta el punto de que actualmente se están diseñando programas específicos para poder realizar operaciones asistidas y dirigidas a través de internet.

Existen varios lenguajes JAVA, C++, COBOL, C#, entre otros, cada uno de ellos cumple una función específica y está orientado a diferentes sectores dentro de la informática.



Imgm1\_2

Quizás el lenguaje actualmente con más futuro y más utilizado sería el lenguaje JAVA, dadas sus características es un lenguaje con el que se puede programar casi todo y está orientado fundamentalmente a internet y a la web lo cual en la actualidad lo convierte en el lenguaje *“de moda”*

**Conceptos importantes acerca de los lenguajes de programación.**

* **CÓDIGO FUENTE**

Conjunto de líneas de texto realizado en un determinado lenguaje informático que a modo de instrucciones están contenidas en un programa que permiten ejecutar ese mismo programa.

* **LENGUAJES DE PROGRAMACIÓN COMPILADOS**

Aquellos que precisan de un software compilador para transformar el código fuente completo en código máquina en caso de que no tenga error. En caso de que detecte un error el compilador nos alerta de ello para solventarlo.

La CPU del ordenador terminará ejecutando el código máquina que el compilador ha generado proveniente del código fuente original.

Existe un compilador para cada tipo de lenguaje existente.

Los lenguajes compilados son más rápidos que los interpretados.

* **LENGUAJES DE PROGRAMACIÓN INTERPRETADOS**

Aquellos que por medio de un software interprete va cogiendo línea a línea el código fuente y lo va llevando a la CPU para que lo ejecute.

Si se produce un error en unas de las líneas la ejecución del programa se detiene ya que el software intérprete no es capaz de supervisar y detectar los errores.

Entre los lenguajes más conocidos están JavaScript, VisualScript

Los lenguajes script son aquellos que están estrechamente relacionados con las páginas web

**JAVA** es un lenguaje de programación que no es ni compilado ni interpretado al 100% ya que ambos procesos participan en él.

***Proceso de compilación***

|  |  |  |
| --- | --- | --- |
| Código Fuente ------->  Java | Compilador ------->  Java | ByteCodes  (archivos).class |

***Proceso de interpretación***

|  |  |  |
| --- | --- | --- |
| ByteCodes ------->  .class | JVM --------------->  (Java Virtual Machine) Intérprete de ByteCodes | CPU |

* **IDE (Integrated Development Environment)**

Entorno de desarrollo gráfico para escribir código fuente en un determinado lenguaje.

Tiene una serie de instrumentos gráficos que nos ayudan a la realización del programa.

Es capaz de detectar los errores en tiempo de compilación.

Algunos ejemplos son: NetBeans. Eclipse, Unity

* **Compiladores J.I.T. (Just In Time)**

Se utilizan para acelerar la ejecución de los ByteCodes

|  |  |  |
| --- | --- | --- |
| ByteCodes ------->  .class | Compilador J.I.T---------->  (para cada plataforma existente en el mercado) | (ejecutable).exe (para una determinada plataforma) |

**1.2 Distintas modalidades y carreras profesionales**

En nuestros días y al igual que casi todo la programación ha sufrido un proceso de cambio y de modernización adaptándose a los cambios y avances tecnológicos en todas las ramas (ingeniería, medicina, arquitectura, eventos deportivos, ocio, etc…).

****

Imgm1\_3

En los años 80 y 90 la programación más común se denominaba programación estructurada y actualmente en nuestros días es aún utilizada por el sector Banca en la mayoría de sus procesos internos, siendo una tarea ”difícil” el hacer que cambien a otro tipo de programación más moderna como la programación orientada a objetos, ya que son tantos procesos y todos funcionando a la perfección que realizar cambios en toda su estructura cuando menos provocaría una cierta psicosis. Así que se acogen al primer principio del programados *“si funciona, no lo toques”.*

Pese a tratarse de un tipo de programación algo más antigua al ser tan de vital importancia el sector Banca hace que lenguajes de programación estructurada como el COBOL nunca mueran.

Otra de las modalidades actualmente tan en auge es todo lo relativo a la programación WEB. Lenguajes como Java Script se han convertido en un referente a la hora de proporcionar dinamismo a las página web en el lado del cliente ( páginas HTML).

JAVA también es un referente en la programación web del lado del servidor , lenguajes como JSP, Servlets son muy utilizados por las grandes empresas.



Imgm1\_4

Dentro de la modalidad de programación web cabe también destacar el lenguaje PHP que está muy difundido en la comunidad de programadores, y aunque se trata de un lenguaje relativamente más antiguo, su auge y mejoras hacen que permanezca en primera línea.

Otra de las modalidades es la programación para dispositivos móviles tanto para sistemas operativos android como para ios.

Lenguajes como Java( una vez más) para desarrollos android o lenguajes como swift (para IOS) son los más comunes en este tipo de desarrollos.



Imgm1\_5

Dentro de las carreras profesionales que podríamos desarrollar si optamos por el mundo de la programación y todo su entorno serían las siguientes:

* Programador (Java ,Cobol, C, C++ , C#,etc..)
* Analista
* Analista/Programador
* Desarrollador WEB
* Analista de Sistemas
* Desarrollador de APPs para dispositivos móviles (Android, IOS)
* Desarrollador de Videojuegos y plataformas interactivas (UNITY)
* Jefe de proyecto

**

Imgm1\_6

Todas estas modalidades por supuesto, aplicadas a los números campos de la ciencia y de la tecnología. Ya cada uno a nivel personal debería de autoencuadrarse en el sector que más le atraiga ya bien sea el sector médico, sector ocio con una gran cantidad de negocios como cine, televisión, videojuegos, sector turismo viajes , vuelos , barcos , etc…

**1.3 Concepto general de Programación Estructurada**

Cualquier programa a nivel funcional constaría de tres elementos: Entrada / Proceso / Salida.

|  |  |  |
| --- | --- | --- |
| ENTRADA -------> | PROCESO ---------->  Sentencias (Instrucciones) de control flujo del programa | SALIDA |

Es decir, a nivel funcional un programa transforma unos datos de entrada a unos de salida mediante un proceso. Todos los programas se pueden representar con un pseudo lenguaje que se puede denominar diagramas de flujo o pseudocódigo.

* **Diagrama de flujo**

Conjunto de símbolos mediante los cuales podemos representar cualquier sentencia de programación. Son independientes de cualquier lenguaje de programación.

|  |  |
| --- | --- |
| |  | | --- | | ***Simbolos fundamentales:***  *Inicio y fin de proceso.*  *Realizar un proceso( una operación*  *matemática por ejemplo).* |   *Entrada Y/O salida de datos.*  *Decisión (la respuesta puede ser Si/ NO)*    *Salida por pantalla* |

* **Pseudocódigo**

Representar todas las instrucciones o sentencias de las que consta nuestro programa representado mediante un “lenguaje llano”.

También es independiente de cualquier lenguaje de programación.

Utilizaremos el castellano para escribirlo.

|  |  |
| --- | --- |
|  | *Si nombre=’Luis*  *mostrar en pantalla ‘luis es el profe”*  *Fin\_Si* |

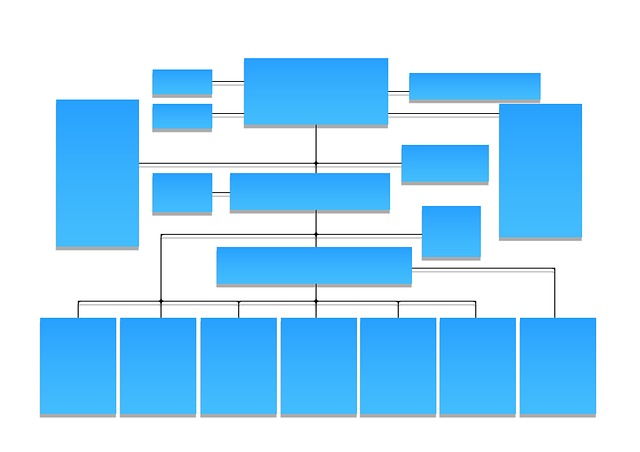
|  |  |
| --- | --- |
|  | *Desde i=0 Hasta i=10*  *Imprimir variable i*  *Fin\_Desde* |

|  |  |
| --- | --- |
|  | *Inicio*  *Declaración:*  *nombre*  *Inicialización:*  *nombre ← ‘luis’*  *Proceso:*  *Mostrar nombre*  *Fin* |
|  |  |

* **Programación Estructurada**

En la programación estructurada cada programa debemos entenderlo como una sola unidad funcional.

El programa está estructurado en varias zonas, pero siempre hablamos de una única unidad secuencial. El programa se procesa de forma secuencial, de arriba a abajo.



Imgm1\_7

En el paradigma de la programación estructurada vamos a tener tres elementos: Secuencias / Estructuras alternativas / Estructuras repetitivas

|  |  |  |
| --- | --- | --- |
| SECUENCIAS------> | ESTRUCTURAS--------->  ALTERNATIVAS | ESTRUCTURAS  REPETITIVAS |

Ejemplos de programación estructurada son RPG, Cobol. Lenguajes utilizados por los bancos en su gran mayoría.

**1.4 Concepto general de Programación Orientada a Objetos**

El programa consta de varias unidades funcionales todas ellas independientes entre sí.

Cada unidad funcional está formada por uno o varios objetos.

La P.O.O trata de llevar los objetos del mundo real al del software.

Un **objeto** de programación es una entidad software perteneciente a una clase que va a contener las propiedades y métodos descritas en esa clase.

Una **clase** es un plantilla que contiene propiedades y métodos que van a heredar todos los objetos pertenecientes a esa clase.

Mediante un proceso de abstracción debemos ser capaces de identificar propiedades y métodos que son comunes a todos los objetos que pertenezcan a esa clase.

Si los objetos del mundo real tienen propiedades y métodos los objetos del mundo del software también.

Para acceder a las propiedades y métodos de un objeto del mundo del software su utiliza la “notación punto”:

objeto.propiedad

objeto.metodo()

Los métodos los vamos a representar siempre con dos paréntesis al final de su nombre, por ejemplo el método acelerar de la clase Coche se representará:

**Acelerar()**

|  |  |
| --- | --- |
| OBJETO MUNDO REAL: COCHE  PROPIEDADES:  Marca  Modelo  Color  Matrícula  MÉTODOS:  Arrancar()  Acelerar()  Frenar() | OBJETO MUNDO SOFTWARE  **PROPIEDADES**  **+**  **MÉTODOS** |

Las aplicaciones cliente-servidor se pueden hacer en C, C++, Visual Basic, C#...

Las de escritorio tienen en común que se usan ventanas Windows. Las aplicaciones en Internet cliente-servidor no existían, porque a través de internet no se pueden comunicar ventanas Windows porque tienen una codificación que no puedes ser transferida a través de un protocolo Web.

Por eso, en 1985, James Gosling crea un proyecto basado en una idea: sacaron un lenguaje basado en C (el lenguaje se llamaba OAK) capaz de manipular cualquier dispositivo electrónico de mano (pda´s, electrodomésticos…) que estuviesen todos manejados por un software universal.

Posteriormente trataron de llevar esa idea al mundo de los pc. Así sacaron la primera versión de JAVA: JAVA 1.0 (Android se basa en Java Micro Edition).

Crearon JVM, la JAVA VIRTUAL MACHINE (es un software necesario para que pueda funcionar cualquier aplicación java en una máquina ).

Existen IDE(entornos de Desarrollo Integrado) para JAVA, como NetBeans y Eclipse (entornos para desarrollar en JAVA).Estos 2 son los más importantes.

JAVA se creó exclusivamente para internet en un principio , porque sus programas si se pueden trasmitir por esta red ( protocolo HTTP).

Otros lenguajes y tecnologías no lo permitían.

**Los 3 grandes bloques de JAVA**

* **J2SE - Java 2 Standard Edition**
  + **Sintaxis**
  + **Fundamentos**
  + **Errores**
  + **GUI (SWING, AWT)**
  + **Ficheros**

Dentro de J2SE vamos a tener dos tipos de programas:

* **Aplicaciones:**  son programas Java que se van a ejecutar en un entorno BASH (que es una consola como la de los sistemas operativos, p.ej. MSDOS)
* **Applets:** son programas Java insertados en una página HTML, vienen con un sistema de seguridad llamado **SandBox** (caja de arena), porque al final te estás descargando un programa y puede contener código maligno que por ejemplo formateé tu disco duro.

CARACTERÍSTICAS DE SANDBOX Security Model:

* Los Applets no pueden ejecutar ninguna operación de entrada-salida (lectura-escritura en tu disco duro, gracias al modelo que implementan los navegadores=SandBox)
* Los Applets no se pueden conectar a ninguna Url que no sea aquella de la cual proceda(no puede abrir un socket(dirección ip + puerto) de comunicación con una máquina que no sea de la cual proceda), salvo que tu Applet esté certificado, es decir, que sea un Applet TRUSTED, que certifican entidades certificadoras como VERISIGN,ATT ,etc..
* **J2EE -Java 2 Enterprise Edition**
  + **JDBC (Java para bases de datos)**
  + JSP (Java Server Pages)
  + JPA
  + SERVLETS (Programas que se ejecutan en el servidor y generan HTML)
  + RMI (Remote Method Invocation)
  + JNDI (JAVA Name and Directoy Interface)
  + SAX (Simple Api for XML)

Java ha mejorado todas estas Api para generar Frameworks basados en MVC (Model View Controller = Modelo Vista Controlador) que separa por capas las aplicaciones, como Hibernate, Struts y Spring.

Estas tres últimas tecnologías son altamente demandadas por las grandes empresas.

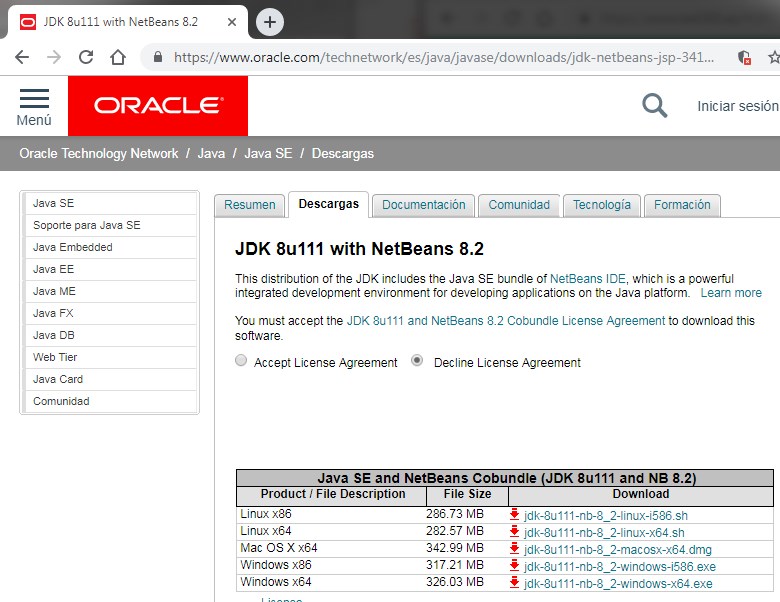
**Descarga del JDK y el entorno integrado (IDE) NetBeans**

Entramos en Google y tecleamos JDK + NetBeans



Imgm1\_8

Hacemos click en el primer enlace y entramos ya en la página de Oracle para realizar la descarga.



Imgm1\_9

Aceptamos la licencia.

De los enlaces siguientes podemos descargar el de Windows 64 si tenemos una arquitectura de Windows de 64 bits(lo más normal).

Una vez descargados instalaremos tanto el JDK y después el NetBeans.

El proceso es bastante sencillo.

**Tipos de programas JAVA**

**Aplicaciones:** se entiende aquellos programas que se ejecutan en un entorno BASH, que es un entorno en el cual está presente el sistema operativo por detrás. En el caso del Windows el sistema operativo es MS-DOS y la aplicación Símbolo del sistema



Imgm1\_10

**Configurar Windows para poder compilar y ejecutar aplicaciones JAVA desde la consola de MS-DOS:**

* Explorador de Windows -> Botón derecho sobre “Este equipo” -> Propiedades -> Configuración Avanzada del Sistema -> Variables de entorno:
* Variables de usuario para usuario:

Crear Nueva: Nombre de la variable: classpath

Valor de la variable:

C:\Program Files\Java\jdk1.8.0\_111\lib;.;

* Variables del sistema:

Editamos la variable path del sistema y añadimos una nueva entrada:

Nueva (añadir):

C:\Program Files\Java\jdk1.8.0\_111\lib\bin

* En el disco duro (c:/) creamos una carpeta que se llame “java”
* Abrimos el editor de notas (notepad) y escribimos:

*class Ejemplo {*

*public static void main(String[ ] args)*

*{*

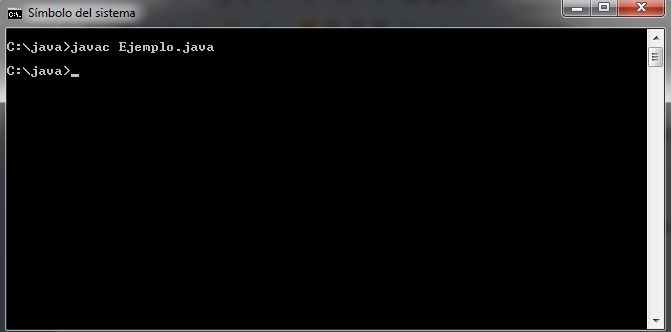
*System.out.println("Hola Mundo");*

*}*

*}*

Y lo guardamos en la carpeta “java” que hemos creado con el nombre **Ejemplo.java**

* Para comprobar que funciona volvemos a la aplicación Símbolo del sistema, escribimos:

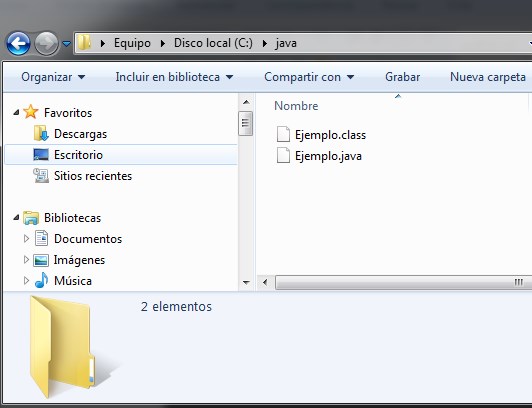


Img1m1\_11

Donde:

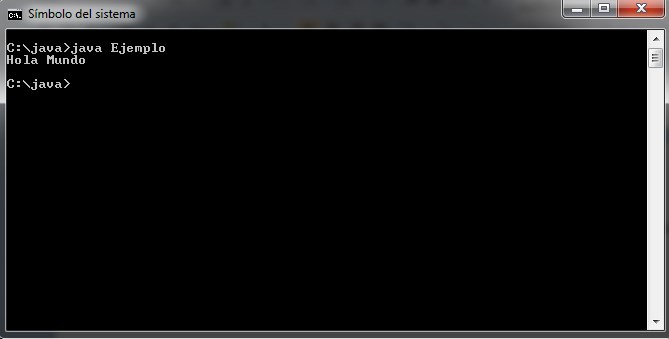
C:\java> javac Ejemplo.java → compila el archivo java

Al compilar este archivo y en caso de que no tenga errores en el proceso de compilación, generará el fichero con el mismo nombre pero con extensión .class (Ejemplo.class).



Imgm1\_12

C:\java> java Ejemplo2 → Ejecuta el archivo .class que ha resultado de la compilación



Imgm1\_13

|  |
| --- |
| **IMPORTANTE:** *Todo este proceso anteriormente descrito es para compilar y ejecutarun archivo JAVA en caso de que no dispongamos de un IDE tipo NetBeans o similar.* |

**Applets:** es un programa JAVA que va insertado dentro de una página HTML

|  |  |  |
| --- | --- | --- |
| **Html**   |  | | --- | | **Applet** | | <applet  code=”fichero.class” width=”200” height=”200”>  </applet > |

**Appleviewer:** es un sumilador para poder ejecutar applets sin necesidad de tener un navegador. Va incluido dentro de las herramientas del JDK.

Pese a que se siguen utilizando por entidades muy importantes Google ha desestimado los applets de su navegador.

Aún así estamentos gubernamentales siguen utilizando los applets en sus aplicaciones web , tal es el caso del Ministerio de Hacienda, la policía nacional , etc…, por ejemplo en las aplicaciones para poder obtener certificados digitales y poder operar desde casa, estos organismos nos recomiendan utilizar el navegador FireFox debido al uso de applets de java, o también para realizar la renovación del dni o pasaporte sucede lo mismo, necesitamos un browser que admita applets.

El sistema de seguridad de los applets es el Sandbox Security Model

**Características de SANDBOX Security Model**

* Los Applets no pueden ejecutar ninguna operación de entrada-salida (lectura-escritura en tu disco duro, gracias al modelo que implementan los navegadores = SandBox)
* Los Applets no se pueden conectar a ninguna Url que no sea aquella de la cual proceda(no puede abrir un socket(dirección ip + puerto) de comunicación con una máquina que no sea de la cual proceda), salvo que tu Applet esté certificado, es decir, que sea un Applet TRUSTED, que certifican entidades certificadoras como VERISIGN

**Estructura de un programa en JAVA**

Todos los programas van dentro de la definición de una clase.

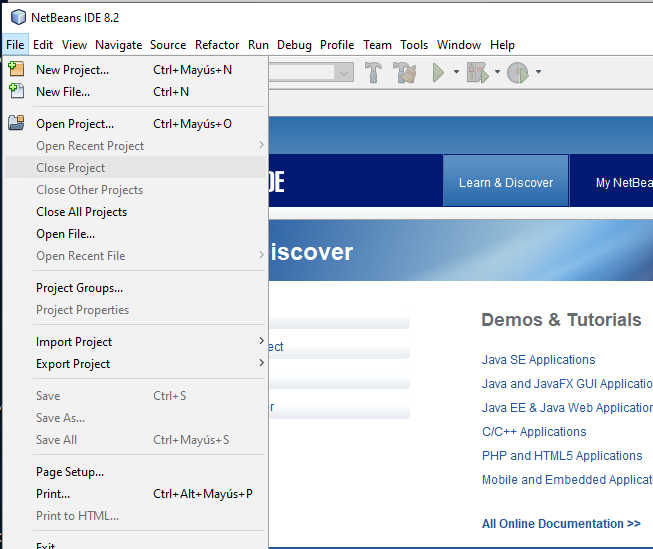
|  |
| --- |
| *class Ejemplo {*  *}* |

Hay una clase que siempre es la principal que es la que lleva el método main o función main

|  |
| --- |
| *class Ejemplo {*  ***public static void main (String arg [ ] ) {***  ***}***  *}*  La función main es obligatoria dentro de un programa java |

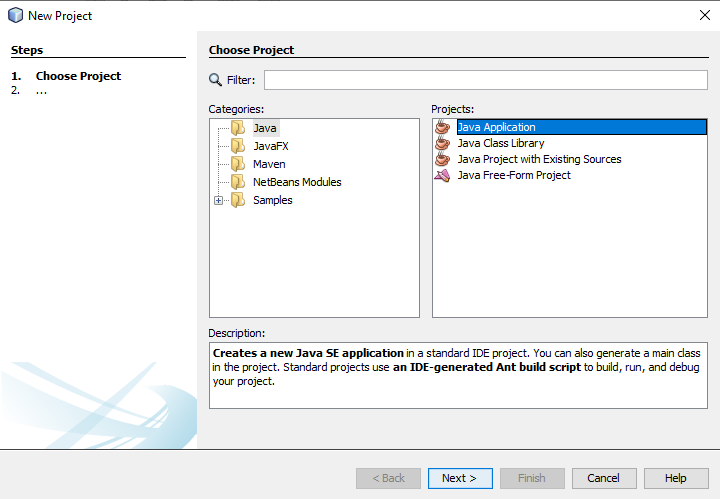
**Creación de un programa en JAVA con NETBEANS**

Abrir la aplicación y seguir estos pasos:



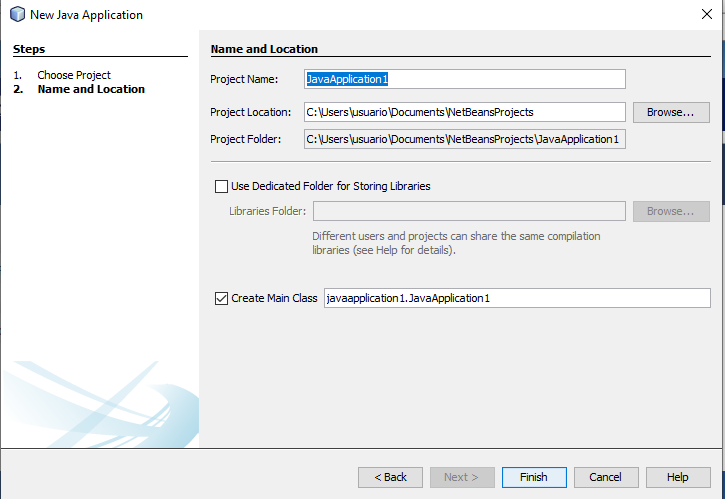
Imgm1\_14

Elegimos la opción del menú File🡪New Project…



Imgm1\_15

Marcamos de la lista de la derecha **JAVA Application** y le damos al botón de NEXT

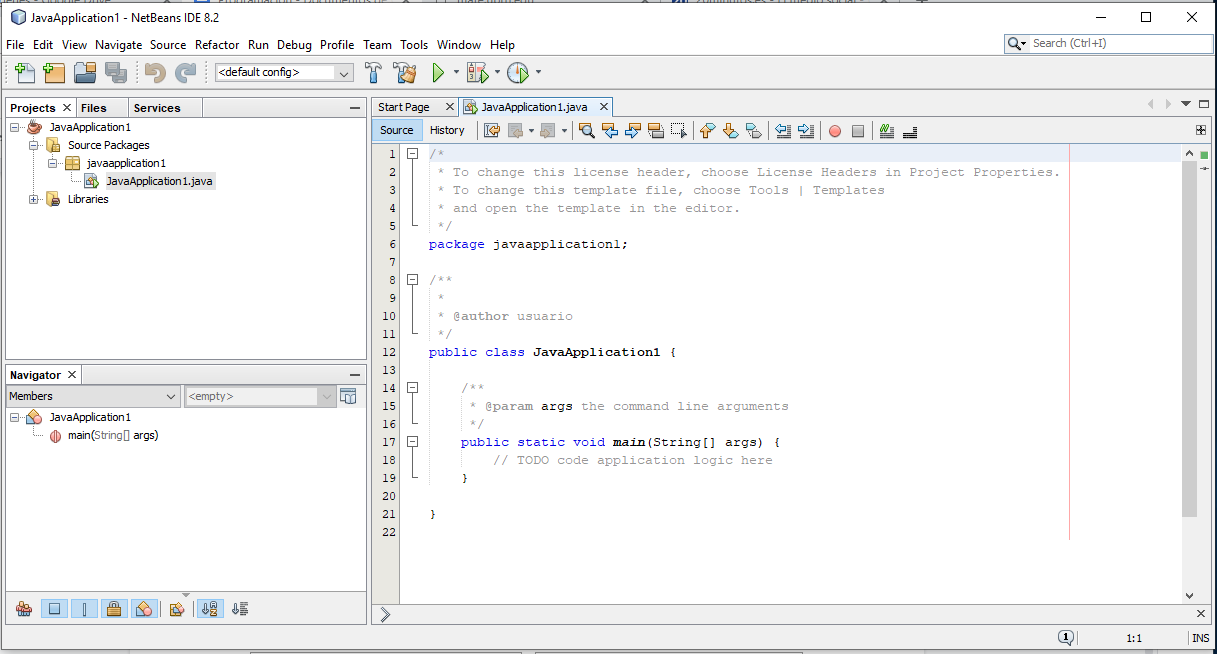


Imgm1\_16

Elegimos un nombre y una ubicación para nuestro proyecto.

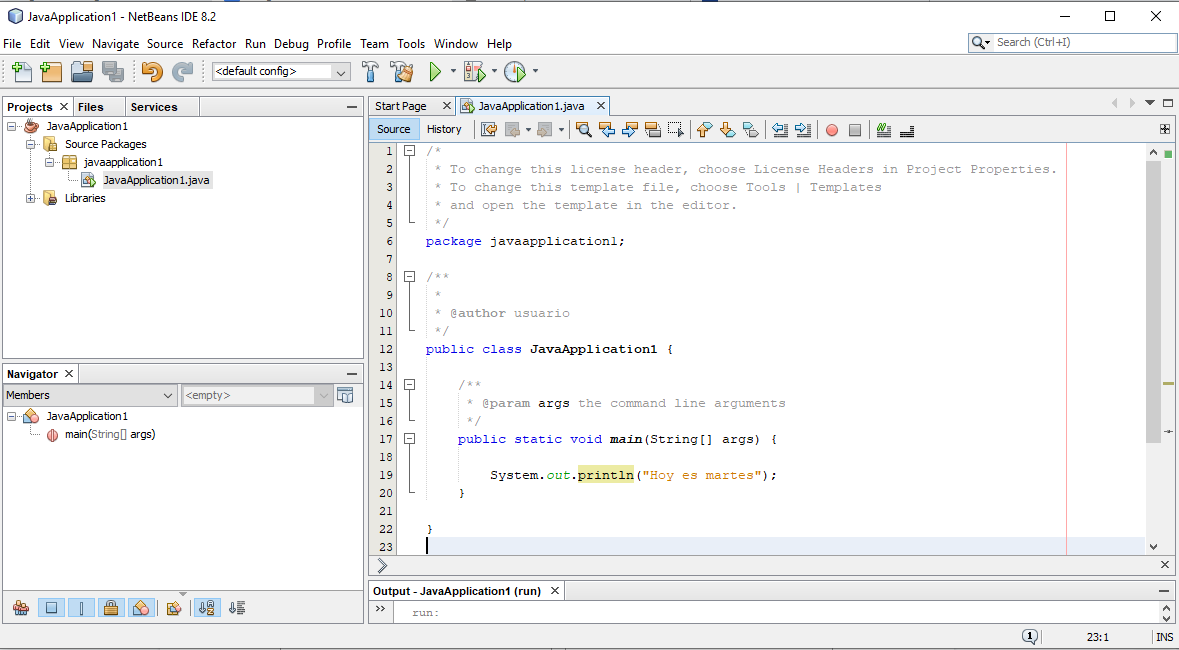
Por defecto Netbeans lo guarda enla ruta:

**C:/users/usuario/documents/NetBeansProjects**



Imgm1\_17

Y el propio NetBeans nos genera una clase por defecto, podemos usar esa misma o crear nosotros una nueva.



Imgm1\_18

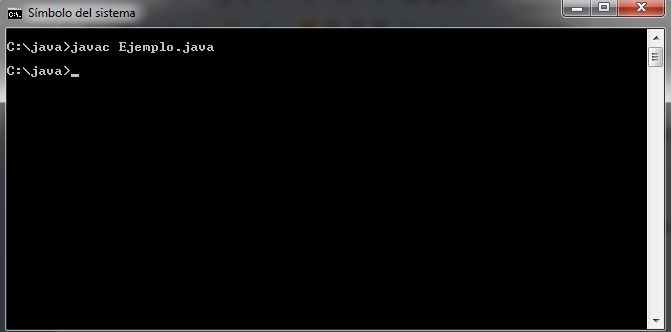
En este ejemplo visualizamos por pantalla la frase:

“Hoy es martes”

Utilizando la instrucción

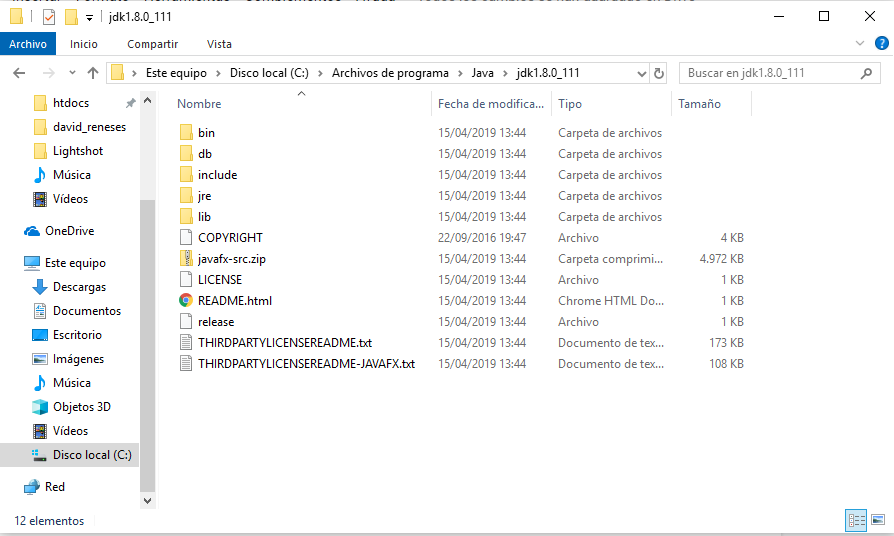
**System.out.println(“Hoy es martes”);**

El JDK ( java development kit) es el kit de desarrollo de JAVA, con este kit no necesitamos un IDE avanzado tipo NetBeans o Eclipse, y es de gran utilidad cuando no lo tenemos a nuestro alcance yta que con el JDK se puede hacer también todo , lo malo es que la parte visual no es nada agradecida ya que ejecutamos y compilamos desde el símbolo del sistema.



Imgm1\_19

**Estructura JDK**

****

Imgm1\_20

Nos situamos en esta ruta:

**C:\Program Files\Java\jdk1.8.0\_111**

La versión puede variar según la que nos hayamos descargado de la página de Oracle

Cuando realizamos las descargas.

**CARPETA lib**

* Contiene todas las librerías de clases que vienen por defecto en JAVA.
* Las librerías están organizadas en packages (paquetes).
* Un package equivale a una carpeta.

**CARPETA bin**

* Contiene todas las herramientas del JDK
* Entre otras las más importantes el compilador (javac.exe) y el intérprete (java.exe) de JAVA

**CARPETA jre (java runtime environment)**

* Es el núcleo de JAVA, es lo que hace que JAVA se pueda instalar en cualquier máquina.
* Cualquier máquina que tenga instalado el jre es capaz de poder ejecutar cualquier aplicación JAVA.

**1.5 Programación Estructurada vs Programación Orientada a Objetos**

Siempre recuerdo la anécdota en la cola del banco donde la señora que tenía delante de mi al decirle el cajero cuando le llegó su turno *“Señora no puede hacer trasferencias porque se ha caído el servidor”* y yo atónito ví como levantaba su paraguas y si no la paro le hubiese dado un buen paraguazos al pobre cajero.

Yo enseguida le dije a la señora que lo que quiso decir aquel hombre es que el ordenador central había dejado de funcionar con lo que no se podía hacer ningún tipo de operación. Y es que al tratarse de un Banco todo su sistema cliente/servidor basado en lenguajes Cobol ( estructurados) tenían la pega de que si deja de funcionar el servidor ( ordenador central) automáticamente dejarán de poder utilizarse los clientes( nosotros y nuestras operaciones en el banco).



Imgm1\_21

En la programación orientada a objetos los sistemas requeridos se denominan **Sistemas Distribuidos** ello significa que nuestra aplicación pueden estar distribuidos en varias máquinas , no solamente en una, de tal modo que si en algún momento una de ellas deja de funcionar (se caé) el resto seguirá funcionando y solamente una pequeña parte de la aplicación será la que deje de estar operativa y no toda, como sucedía en el caso de los sistemas cliente/servidor.

Por resumir , la programación estructurada siempre la vamos a ver como una unidad funcional en la que aunque podríamos tener subrutinas independientes dentro del mismo programa , no llegarían a considerarse 100% independientes y siempre dependerían del programa central.

En la programación orientada a objetos los programas sí constituyen unidades funcionales independientes , reutilizables y distribuibles entre plataformas.

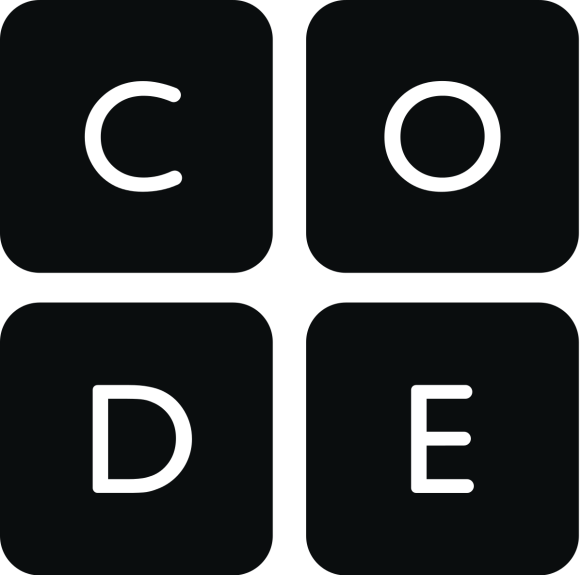
**1.6 El proyecto CODE.ORG**

El proyecto Code.Org es una iniciativa para que todas las personas incluidas niños, adolescentes, jóvenes y mayores de cualquier edad aprendan a programar basándose en que la programación en sí nos ayuda en nuestras vidas en muchísimas facetas tanto en la organización de nuestras ideas, quehaceres diarios, planificación de casi cualquier cosa, manera de tomar decisiones , etc…

Nos hace ver que realmente la programación está presente en casi cualquier cosa que nos rodea.

Si aprendemos a programar, nos facilitará la realización de múltiples tareas cotidianas.

Se trata de un proyecto promovido por asociaciones en las que están muchas personas famosas como jugadores de basquet de la NBA, actores, deportistas de élite en general, artistas famosos, y personas que no tienen nada que ver con el mundo de la informática.



Imgm1\_22

Puedes visitar el video promocional en la siguiente dirección de youtube:

<https://www.youtube.com/watch?v=sDk1pTDPROI>

En la página oficial de CODE ORG (<https://code.org/>) puedes crear y aportar tu propia línea de código.

**MÓDULO 2. POO CONCEPTOS BÁSICOS I**

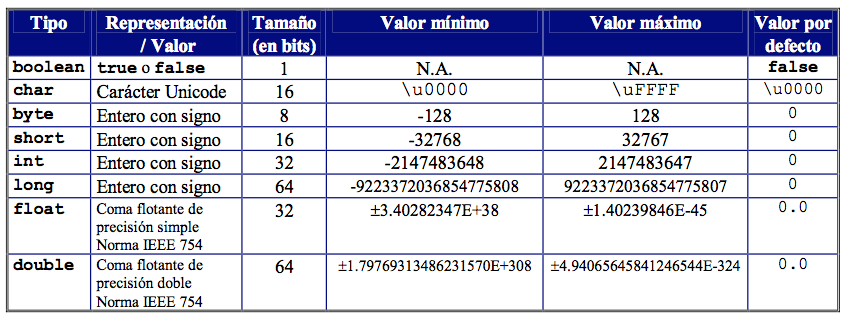
**2.1 Variables primitivas**

Es una posición de memoria cuyo valor puede variar a lo largo de la ejecución de un programa.

Ese valor está representado por un nombre y un tipo.

En programación lo que lleva un signo = es una expresión (ej. var i = 3) **literal**

Lo que va a la derecha del = se llama literal en todos los lenguajes de programación. En JAVA, al ser un lenguaje multiplataforma tiene una serie de bugs, entre los cuales está que cuando detecta un literal numérico directamente lo convierte en int. Si queremos compilar hay que utilizar un **casting** que es reforzar el tipo de variable que declaramos añadiéndole una F, por ejemplo, en el caso de un número **float**

****

**VARIABLES PRIMITIVAS (en función del tipo de almacenamiento)**

* **NUMÉRICAS**
  + ENTEROS:
    - **byte** (1 byte) desde -27 a +27 -1. Desde -128 a +127

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Valor 1 es - | 26 | 25 | 24 | 23 | 22 | 21 | 20 |
| Valor 0 es + | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

* + - **short** (2 bytes) desde -215  a +215 -1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Valor 1 es - | 214 | 213 | 212 | 211 | 210 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 |
| Valor 0 es + | 16.384 | 8.162 | 4.096 | 2.048 | 1.024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

* + - **int** (4 bytes) desde -231 a+231 -1
    - **long** (8 bytes) desde -263 a+263 -1
  + REALES:
    - **float** (4 bytes),
    - **doble** (8 bytes)
* **BOOLEANOS**
  + BOOLEAN:
    - **boolean** v= true
    - **boolean** v= false
* **CARÁCTER**
  + CHAR: **char** letra= ‘a’(siempre comilla simple)

**VARIABLES OBJETO**

* **String** – en JAVA se consideran objetos que pertenecen a la clase String y siempre va con comillas dobles
* **Array** – en JAVA se consideran objetos que pertenecen a la clase Array

Sintaxis JAVA

* Todas las sentencias acaban en **;**
* Diferencia mayúsculas y minúsculas
* Existen las mismas sentencias que en JavaScript (if, for, while…)
* Los **Array** y los **String** en JAVA son objetos.
* JAVA no utiliza ASCII (1 byte por carácter), utiliza UNICODE (2 byte por carácter)

**En JAVA existen tres tipos de variables en función de su posición dentro del código:**

* **Variables de instancia**: son variables que van dentro de la clase, fuera de los métodos y que para poder utilizarlas hay que crear un objeto. Y se accedería a ellas con el **nombre del objeto** y el punto.

|  |
| --- |
| *public class Ejemplo {*    *int x;*    *public static void main(String[] args) {*    *Ejemplo1 e=new Ejemplo1();*  *e.x=3;*  *}*    *}* |

* **Variables de clase**: van dentro de la clase, fuera de los métodos pero **NO HACE FALTA CREAR UN OBJETO PARA ACCEDER A ELLAS.** Las declaramos con **static**

|  |
| --- |
| public class Ejemplo {    static int x;    public static void main(String[] args) {    x=3;  }    } |

* **Variables locales:** van dentro de los métodos y hay que inicializarlas para poder usarlas.

|  |
| --- |
| *public class Ejemplo {*    *public static void main(String[] args) {*    *int x=3;*  *System.out.println (x);*  *}*    *}* |

**2.2 Métodos**

Un método es un conjunto de instrucciones que solamente se van a ejecutar cuando son llamadas o invocadas desde un punto del programa.

En java los métodos tienen este formato:

**[modificador] tipo\_devuelto nombre ([parámetro1,parámetro2, parámetro3…] )**

**{**

**Acciones;**

**}**

Lo que va entre corchetes es siempre opcional

**[modificador**] puede ser public, friendly( en este caso no ponemos nada delante), protected o prívate .

Lo veremos más delante de momento serán public.

**[tipo\_devuelto]** puede ser cualquier tipo de dato int,char,String, etc..

Si es void significa que el método no devuelve nada

**([parámetro1,parámetro2, parámetro3…]** Parámetros que puede recibir el método, son opcionales.

Ejemplo:

public int suma (int x, int y)

{

return ( x+y );

}

En este ejemplo el método suma es de tipo int (devolverá un número entero) y recibe 2 parámetros x e y también de tipo entero.

La palabra reservada **return** es para indicar que va a devolver un valor, en este caso la suma de x + y.

En el caso de las clases principales de nuestros programas tenemos el método main (también llamada función main) con el siguiente formato:

public static **void** main (String arg[])

{

Acciones;

}

Obsérvese que es de tipo void ( no devuelve nada).

El resto de los parámetros los veremos más adelante para entender bien su significado.

**2.3 Clases**

Las clases son el esqueleto de nuestros programas.

Son estructuras independientes y pueden interactuar unas con otras.

Dentro de las clases nos encontraremos propiedades(variables) y métodos.

Todos los programas java van dentro de la definición de una clase. Ya veremos más adelante la definición exacta de clase . Además dentro de una aplicación java siempre es obligatorio definir el método main al igual que ocurre en el lenguaje C .

Con lo que ya tenemos una idea aproximada de lo que sería el esqueleto de un programa java :

Nombre de una clase ( lo que sería el nombre del programa ) , abriríamos una llave ( { ) , definiríamos el método main con todos sus parámetros y finalmente cerraríamos la llave ( } ).

Si por ejemplo quisiésemos definir el programa Prueba1 lo haríamos de la siguiente forma :

**class Prueba1 {**

**public static void main (String args [ ] )**

**{**

**}**

**}**

La definición del método main es tal y como aparece obligatoriamente , más adelante indicaremos lo que significa cada palabra reservada que lo acompaña .

En cuanto a más aspectos de la sintaxis java , indicaremos que al final de cada sentencia es obligatorio el punto y coma ( ; ) , además el lenguaje java es *case sensitive* , es decir , diferencia entre mayúsculas y minúsculas .

En los siguientes capítulos se hará un estudio más profundo de toda la sintaxis de este lenguje.

**2.4 Arrays y otros componentes**

Los arrays son una lista de elementos de cualquier tipo o clase que ocupan posiciones contiguas en la memoria y además se pueden referenciar a través de un índice..Todos los elementos han de ser del mismo tipo.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |

### 0 1 2 ............ 8

### Array de 9 posiciones que van desde la posición 0 a la posición 8

### Declaración del array.

Existen dos formas:

Tipo\_array NombreArray [ ];

Tipo array [ ] NombreArray;

En la definición de un array observamos que nunca se indica el número de elementos de que consta.

Ejemplos:

String Palabras[ ];   
int numeros[ ];   
String[ ] Palabras;   
int[ ] numeros;

### Creación de un array .

Hay de nuevo dos formas de hacerlo:

Tipo\_Array[ ] NombreArray = new Tipo\_Arra[tamaño];

, donde tamaño es el numero de elementos del array.

Aquí en la creación del array es obligatorio indicar el número de elementos de que consta el array.

Ejemplo:

int numeros=new int [10]; // Array de enteros que va desde la //posición 0 a la 9

La segunda forma de crear un array es creándolo y dándole valores iniciales:

Tipo\_Array[ ] NombreArray = {valor1,valor2, ...};

, los valores entre llaves deben ser del mismo tipo; del tipo Tipo\_Array.

Ejemplo:

char letras []={‘a’,’b’,’c’}; // Array de carácteres de 3 posiciones

### Acceso al array.

El acceso para asignarle un valor a un objeto se hace como sigue:

NombreArray [posición] = valor;

La posición del array comienza desde la posición 0.

Ejemplo:

int numeros[]=new int[3];

numeros [0]=1;

numeros [1]=3

numeros [2]=6

### Arrays multidimensionales.

Java no soporta arrays mutidimensionales pero si podemos declarar un array de arrays consiguiendo el mismo efecto:

Int coordenadas[ ] [ ] = new int [2][2];   
 coordenadas [0][0] = 1;  
coordenadas [0][1] = 2;  
coordenadas [1][0] = 3;  
coordenadas [1][1] = 4;

|  |  |
| --- | --- |
| 1 | 2 |
| 3 | 4 |

Si la dimensión es 2 al array se le denomina matriz.

**Obtener la longitud del array**

Todos los objetos array tienen una variable de instancia llamada **length ,** que contiene el número de elementos presentes en el mismo de este modo podríamos en un bucle for recorrer el contenido del array desde la posición 0 hasta la última de la siguiente manera:

int numeros []=new int[10];

for(int i=0;i<numeros.length;i++)

{ ...

...

}

En este caso se hace uso de la variable length para obtener el número de elementos del array , lo cual es siempre más seguro.

**Compobación de los límites del array.**

A diferencia de C++ , en Java , los accesos no permitidos al array se comprueban por el sistema en tiempo de **ejecución** y cualquier intento de emplear un índice fuera del rango permitido provocaría una excepción*.* ( Es un error en tiempo de ejecución que en java se denominan excepciones y serán tratadas en un capítulo aparte ) .

que puede ser controlada desde el propio código. Cada excepción java o error en tiempo de ejecución es representada por una clase java siendo el error un objeto de dicha clase.

En el caso de tratar de acceder a una posición que excede el límite del array la clase que representa este error se llama *ArrayIndexOutOfBoundException* , estamos por ejemplo accediendo a la posición 7 de un array que solo tiene 5 posiciones.

Dentro de la definición del método main :

public static void main (String args[ ] ){}

String args[ ] corresponde a la definición de un array denominado args , que es de tipo String, o sea almacena datos en formato cadena , y es el array donde se almacenan los datos que se le pasan al programa por línea de comandos .

A la hora de ejecutar el programa :

C:>java Prueba 6 ↵

El 6 en este caso se almacenaría automáticamente en la posición 0 del array args . Pero se almacenaría como la cadena “6” ya que acabamos de decir que el array es de tipo cadena de caracteres . Para poder operar con él en caso de querer realizar una operación aritmética habría que convertirlo a número entero con la función :

Integer.parseInt(args [0]); // args [0] hace referencia a la primera posición del array args.

Si queremos pasar más de un parámetro al programa desde la línea de comandos , se pondrían uno a continuación del otro seguido por espacios en blanco.

Ejemplo:

C:>java Prueba 4 hola 56 luis ↵

Array args

|  |  |  |  |
| --- | --- | --- | --- |
| “4” | “hola” | “56” | “luis” |

0 1 2 3

El Array args es el único array en java cuya dimensión es indefinida .

**2.5 Funciones de cadena**

Java define además otros tipos que no están en la anterior clasificación de tipos primitivos o elementales de datos

**El tipo String:** Para las cadenas de caracteres , pero en java son consideradas como objetos de la clase String.

También en java los arrays son considerados como objetos. ( Esta es otra característica importante del lenguaje java).

Ejemplo:

String cadena =”Hola”;

Aunque en otros lenguajes sería un tipo básico en java se considerará un objeto de la clase String con lo que tendrá propiedades y métodos . Es importante ir quedándonos con el concepto de que dentro de una clase java lo que vamos a tener siempre son propiedades y métodos únicamente, solo en algunos casos que veremos también a lo largo de este curso veremos que se pueden incluir otras clases también.

En java no sucede como en C , no podemos “apuntar” a cada posición dentro de un String , solamente a la cadena entera. Para poder manipular cada posición de un String tendríamos la clase StringBuffer que nos permite acceder igual que en C a cada una de las posiciones como si de un array se tratara.

Los Strings u objetos de la clase String se pueden crear explícitamente o implícitamente. Para crear un String implícitamente basta poner una cadena de caracteres entre comillas dobles. Por ejemplo, cuando se escribe

System.out.println("Curso de Java");

Java crea un objeto de la clase String automáticamente.

Para crear un String explícitamente es decir como si fuese un objeto de la clase String ,( operador *new*) escribimos:

String cadena=new String("Curso de Java");

También se puede escribir, alternativamente:

String cadena="Nuestro primer programa";

Para crear un String vacío o nulo ( cadena vacía) se puede hacer de estas dos formas:

String cadena="";

String cadena=new String();

Un String nulo es aquél que no contiene caracteres, pero es un objeto de la clase String. Sin embargo,

String cadena;

está declarando un objeto cadena de la clase String, pero aún no se ha creado ningún objeto de esta clase.

Una cadena en definitiva es un objeto de tipo String, por lo tanto tiene propiedades y métodos

**MÉTODO length**: devuelve un dato que representa la longitud de la cadena.

|  |
| --- |
| String cadena=”Hoy es lunes”  For (int i=0; i<cadena.lenght; i++)// |

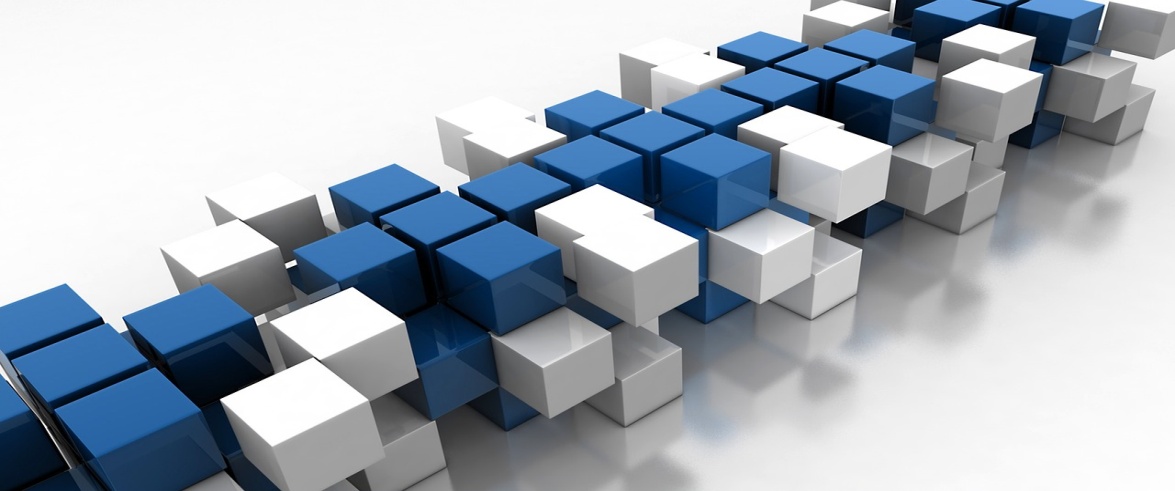
**MÉTODO charAt** (int posición): devuelve el carácter que está en “posición”

|  |
| --- |
| String cadena=”Hoy es lunes”  Char letra = cadena.chartAt(3); // *Devolvería el espacio en blanco   existente entre “hoy es”* |

**MÉTODO toUpperCaser** ( ): convierte a mayúsculas

**MÉTODO toLowerCaser** ( ): convierte a minúsculas

**2.6 Estructuras de control de flujo**

****

Imgm2\_1

Hay dos grandes grupos o bloques:

**ALTERNATIVAS:**

**Simple:** Si (condición) 🡪 Si se cumple la condición se cumple la Acción/es

Acción/es

Fin\_Si

**Compuesta:** Si (condición)

Acción1/es

Si No

Acción2/es

Fin\_Si

**Múltiple:** Caso (valor variable) 🡪 Sólo se ejecutará un caso de todos los posibles

Caso1 : Acción1/es

Caso2 : Acción2/es

…..

CasoN : AcciónN/es

Fin\_Caso

**REPETITIVAS (bucles):**

**Conocemos número repeticiones:**

Desde (inicio hasta fin)

Fin\_Desde

**No conocemos número repeticiones**

Mientras (condición) 🡪 Mientras se cumple condición se ejecuta acción

Acción/es

Fin\_Mientras 🡪Se ejecutará de 0 a N veces

Hacer 🡪Se ejecutará de 1 a N veces

Acción/es

Fin\_Hacer(condición) 🡪 Si se cumple la condición vuelve a hacer

**2.7 Condicional if**

**Alternativa simple**: Se cumple la acción si la condición es true

|  |
| --- |
| *if (condición)*  *{*  *Acción/es;*  *}*  *}* |

|  |
| --- |
| *public class Ejemplo {*    *int numero;*    *public static void main(String[] args) {*    *Ejemplo e=new Ejemplo ();*  *e.numero=5;*    *if (e.numero==5){*  *System.out.println("El número es igual que 5");*  *}*    *}*  *}* |

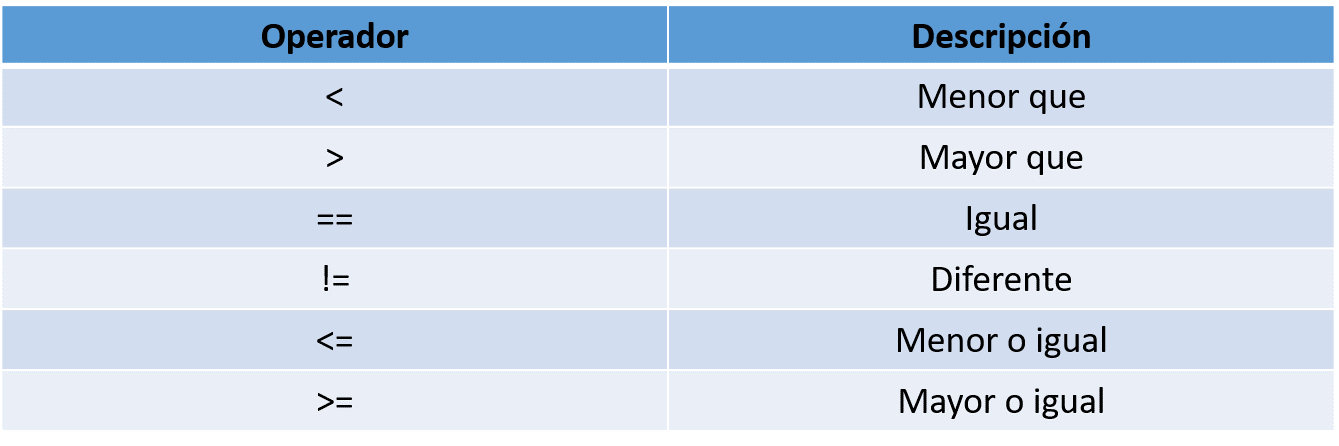
**Alternativa compuesta**: Si se cumpla la condición se ejecuta la acción 1 o conjunto de acciones y si no (else) se ejecuta la acción 2 o conjunto de acciones.

|  |
| --- |
| *if (condición)*  *{*  *Acción1;*  *} else {*  *Acción2;*  *}* |
|  |

|  |
| --- |
| *public class Ejemplo {*    *int numero;*    *public static void main(String[] args) {*    *Ejemplo e=new Ejemplo ();*  *e.numero=5;*    *if (e.numero>5){*  *System.out.println("El número es mayor que 5");*  *} else {*  *System.out.println("El número es igual que 5");*  *}*    *}*  *}* |

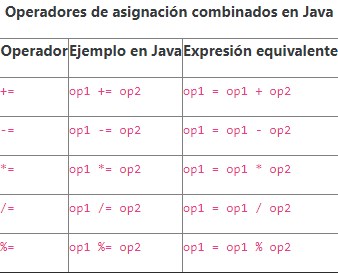
**OPERADORES RELACIONALES**

Son aquellos operadores que se utilizan en las condiciones.



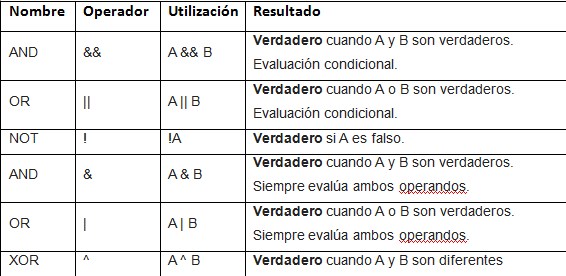
Imgm2\_2

**OPERADORES DE ASIGNACION**

****

Imgm2\_3

**OPERADORES BOOLEANOS**

****

Imgm2\_4

**2.8 Condicional switch**

La instrucción switch es una instrucción de múltiples opciones evaluando una variable. Proporciona una forma sencilla de enviar la ejecución a diferentes partes de la sentencia en función del valor de la expresión. El default solo lleva break si no es la última opción. El break es salir de la sentencia



|  |
| --- |
| Imgm2\_5  *public static void main (String [] args) {*    *int variable = 1;*    *switch (variable) {*  *case 1: System.out.println("Numero 1");*  *break;*  *case 2: System.out.println("Numero 2");*  *break;*  *case 3: System.out.println("Numero 3");*  *break;*  *case 4: System.out.println("Numero 4");*  *break;*  *default : System.out.println("error");*  *}*  *}* |
|  |

EJEMPLO:

Dada la siguiente frase:

“Hoy Es Lunes VEintunuevE dE Abril"

Contar el número de vocales, el número de consonantes y el número de letras e que contiene.

Ha de contar tanto e minúscula como E mayúscula.

|  |
| --- |
| *public class CuentaVocales {*  *public static void main(String arg[])*  *{*  *String cadena="Hoy Es Lunes VEintunuevE dE Abril";*  *int num\_blancos=0, num\_vocales=0,num\_cons=0, num\_es=0;*  *char letra=' ';*  *cadena=cadena.toLowerCase();*  *for(int i=0;i<cadena.length();++i)*  *{*  *letra=cadena.charAt(i);*  *switch(letra)*  *{*  *case 'a':num\_vocales ++;*  *break;*    *case 'e':num\_vocales ++;num\_es ++;*  *break;*  *case 'i':num\_vocales ++;*  *break;*  *case 'o':num\_vocales ++;*  *break;*    *case 'u':num\_vocales ++;*  *break;*  *case ' ':num\_blancos ++;*  *break;*    *default:num\_cons ++;*  *}//switch*    *}//for*  *System.out.println("Número de vocales= " + num\_vocales) ;*  *System.out.println("Número de consonantes= " + num\_cons) ;*  *System.out.println("Número de e's= " + num\_es) ;*    *}//main*          *}//clase* |
|  |

**2.9 Bucle while/ do-while**

**WHILE**: Estructura repetitiva

No conocemos el número de repeticiones

Si la condición se cumple se ejecuta y si nose cumple **no se ejecuta ni una sola vez.**

|  |
| --- |
| ***while (condición)***  *{*  *Acción;*  *}*  *// Fin while* |

**DO WHILE**: Estructura repetitiva

No conocemos el número de repeticiones

**Entra en el bucle una vez y hace una ejecución** y después verifica si la condición se cumple.

En caso afirmativo volverá a entrar en el bucle y así sucesivamente hasta que se deje de cumplir la condición

|  |
| --- |
| ***do***  *{*  *Acción;*  *}* ***while (condición)*** |

EJEMPLO ESTRUCTURA DO WHILE:

EJEMPLO ESTRUCTURA WHILE:

Realizar un programa java que nos pida una palabra por teclado continuamente hasta que introduzcamos la palabra “coche”.

Mientras fallemos sacará el mensaje:

**"!ERROR! , Introduce otra palabra:"**

Si acertamos sacará la frase:

**"!BIEN! , por fin acertaste!!!"**

*public class AdivinaPalabra {*

*public static void main(String[] args) {*

*Scanner sc=new Scanner (System.in);*

*String palabra="";// palabra secreta*

*System.out.println("Introduce una palabra:");*

*palabra=sc.nextLine();//leemos la palabra por teclado*

*while(! palabra.equals("coche"))*

*{*

*System.out.println("!ERROR! , Introduce otra palabra:");*

*palabra=sc.nextLine();//leemos la palabra por teclado*

*}*

*System.out.println("!BIEN! , por fin acertaste!!!");*

*}*

*}*

**2.10 Bucle for clásico**

Es una estructura repetitiva en la cual conocemos exactamente el número de repeticiones.

|  |
| --- |
| ***for (inicialización; condición; incremento)***  *{*  *Acción;*  *}* |

|  |
| --- |
| *public class Ejemplo {*  *public static void main(String[] args) {*  ***for (int i=0; i<5; i=i+1)*** *{*  *System.out.println("valor de i="+i);*  *}*  *}*  *}*  ***Nota: i=i+1 se puede sustituir por i++*** |

EJEMPLO: Realizar un programa que imprima los números pares comprendidos entre 0 y 20 incluyendo el 20.

|  |
| --- |
| *public class Ejemplo {*  *public static void main(String[] args) {*  *for (int i=2; i<=20; i+=2) {*  *System.out.println("valor de i="+i);*  *}*  *}*  *}*  ***Nota: i+=2 equivale a i=i+2*** |

EJEMPLO: Calcula la suma de los números pares comprendidos entre 1 y 20

|  |
| --- |
| *public class Ejemplo{*  *public static void main(String[] args) {*  *int suma=0;*  *for (int i=2; i<=20; i=i+2) {*  *suma += i;*  *}*  *System.out.println("Total = "+suma);*  *}*  *}*  ***Nota: suma += i; es como suma = suma + i*** |

EJEMPLO: Imprimir en modo decreciente los números impares del 15 al 0

|  |
| --- |
| *public class Ejemplo {*  *public static void main(String[] args) {*  *for (int i=15; i>=0; i-=2) { //i+=2 esquivale a i=i-2*  *System.out.println("valor de i="+i);*  *}*  *}*  *}* |

EJEMPLO: Imprimir los pares entre 0 y 20 utilizando el operador %

|  |
| --- |
| *public class NumerosParesConOperadorRestoDivisionEntera {*  *public static void main (String [] args){*  *for (int divisor=1; divisor<=20; divisor+=1)*  *if (divisor % 2 ==0)*  *System.out.println( divisor );*  *}else{*  *System.out.println("");}*  *}* |

EJEMPLO: Imprimir tabla de multiplicar del 9

|  |
| --- |
| *public class TablaMultiplicar{*  *public static void main (String [] args) {*  *int multiplicador=9;*  *for (int multiplicando=0; multiplicando<11; multiplicando++)*  *System.out.println("9 x "+multiplicando +" = "+multiplicador\*multiplicando);*  *}*  } |

**2.11 Bucle foreach**

En las últimas versiones de Java se ha introducido una nuevo estilo del bucle del for tradicional, al que se denomina “for extendido” o “for each”.

Esta forma de uso del for, que ya existía en otros lenguaje ( c # por ejemplo), facilita el recorrido de objetos existentes en una colección sin necesidad de definir el número de elementos a recorrer. La sintaxis que se emplea es:

|  |
| --- |
| for ( Tipo nombreVariableTemporal : nombreColección ) {  sentencias;         } |

En ningún momento se usa la palabra clave each que se usa en otros lenguajes, aunque al for muchas veces se le nombre como for each. Para saber si un for es un for extendido o un for normal vamos a definir la sintaxis que se usa.

La interpretación que se puede hacer de la sintaxis del for extendido es:

**“Para cada elemento del tipo *Tipo* que se encuentre dentro de la colección *nombreColección* ejecuta las sentencias que se indican”.**

La variable local-temporal del ciclo almacena en cada paso el objeto que se visita y sólo existe durante la ejecución del bucle y desaparece después(es una variable local). Debe ser del mismo tipo que los elementos a recorrer.

Ejemplo

|  |
| --- |
| public void listarNombres\_Apellidos () {          for (String i: listaNombresApellidos) {          System.out.println (i); //Muestra cada uno de los nombres y apellidos dentro de //listaNombresApellidos      }    } |

En este tipo de bucles podemos darle un nombre más descriptivo a la variable temporal, por ejemplo:

 public void listarNombres\_Apellidos () {

for(String nombre\_ape:listaNombresApellidos)

{

System.out.println (nombre\_ape);

}

}

**2.12 Los errores que se pueden producir en un programa y su control**

Dentro de un programa se pueden producir dos tipos de errores:

* Errores en tiempo de complilación , por ejemplo se me olvida poner un punto y coma al final de una instrucción, pongo una letra mayúscula en vez d eminúscula en un nombre( java hace diferencia entre mayúsculas y minúsculas) , etc….El caso es que el programa **no compila con lo que no se generará el fichero .class**
* Errores en tiempo de ejecución, llamados **Excepciones,** y en este caso el programa sí compila pero a la hora de ejecutarlo se produce un error, por ejemplo si en algún momento del programa dividimos un número por cero, esto en matemáticas en infinito, pero en programación es un error , y el programa se detendría. Lo que debemos lograr es que el programa nos lance un aviso pero que no se detenga, es decir que continúe su ejecución.

**Clasificación de las Excepciones**

### Una excepción Java representa un error en tiempo de ejecución

En Java existen dos tipos de errores:

* Errores de compilación.
* Errores de ejecución; se dividen en dos tipos:
* Errores Lógicos.
* Excepciones: son situaciones anómalas que se producen en tiempo de ejecución y que los detecta la JVM.

Para la generación de excepciones crea un objeto con la descripción del error y se le envía al programa. Si el programa no recoge la excepción se produce una parada en la ejecución del programa. El programa puede recoger la excepción y tratarla para que no se pare el programa.



Imgm2\_6

### Los bloques try/catch/finally.

Una vez visto el manejo de excepciones de un modo practico, conviene fijar las ideas repasando de un modo estructurado la estructura try/catch/finally.

Veamos la estructura :

try {

// código a ejecutarse y que puede producir

// algún tipo de excepción

}

catch(Excepcion1 e1) {

// aquí se ejecutaría el código necesario si dentro

// de try ocurriera la excepción denominada Excepcion1

}

catch(Excepcion2 e2) {

// aquí se ejecutaría el código necesario si dentro

// de try ocurriera la excepción denominada Excepcion2

}

...

finally {

// aquí se ejecutaría SIEMPRE el código que nos

// interese, como pueda ser limpieza de variables ...

}

El bloque **try** constituye un conjunto de sentencias como otras pero que pueden lanzar excepciones que a continuación serán tratadas.

El bloque o los bloques **catch** va a manejar cada uno el tipo de excepción al que hacen referencia si esta ocurre en el bloque try. Su argumento debe ser del tipo **Throwable** o una subclase de este. La excepción la cogerá la primera cláusula catch cuyo argumento sea del mismo tipo que la excepción ocurrida.

Puede haber todos los bloques catch que nos sean necesarios, igualmente puede no haber ninguno. Por ello, podemos deducir que no debe haber un bloque catch por cada tipo de excepción que pueda ocurrir.

### La cláusula finally.

Puede darse el caso que tengamos que hacer una determinada acción necesariamente, para asegurarnos que se va a realizar conviene crear una excepción, pero es conveniente usar una cláusula dentro de la sentencia try que esta diseñada con este propósito, es la clausula **finally**:

algunaClaseArchivo f = new algunaClaseArchivo();

if (f.open("/a/file/name/path")) { // instrucción para abrir un fichero

try {

algunMetodoExcepcional();

}

**finally** {

f.close(); // instrucción para cerar un fichero

}

}

Lo que hay dentro de la cláusula **finally** se va a ejecutar siempre pase lo que pase .De este modo lo que hacemos es poner la instrucción de cierre del fichero dentro de una claúsula finally , así suceda lo que suceda tendemos la seguridad de que el fichero se va a cerrar.

Esta forma de trabajar se comporta mejor que la siguiente:

algunaClaseArchivo f = new algunaClaseArchivo();

if (f.open("/a/file/name/path")) {

try {

algunMetodoExcepcional();

}

catch (Throwable t) {

f.close();

throw t;

}

}

La cláusula **finally** se usa generalmente para la limpieza del sistema; entiéndase cierre de ficheros, liberación de recursos,... Una propiedad importante del bloque finally es que siempre se va a ejecutar aunque solo se ejecutase una porción de try y se saltase con un break o return.

El orden establecido de ejecución de la estructura completa es: ejecutar try y si ocurre alguna excepción se salta a los catch, una solución es que se coja aquí la excepción y se trate y otra es pasársela para que la coja el nivel superior subiendo en la jerarquía. Después de esto siempre se ejecutara el bloque finally.

Una utilidad de la pareja try/finally y saliéndonos de las excepciones esta en usar try como etiqueta para un conjunto de sentencias y finally como bloque de liberación de recursos después de estas que se ejecutara siempre haya o no dentro del try breaks, continues o returns.

### Uso de excepciones.

Una vez diseñado el programa para hacer lo previsto en principio, hay que pasar a definir el manejo de los casos en que se produzca un error en tiempo de ejecución. Para ello se hará uso de las excepciones. La forma de tratar las excepciones que Java proporciona tiene un aspecto como el que sigue:

public class MiPrimeraClaseExcepcional {

public void unMetodoExcepional() throws MiPrimeraExcepcion {

//Código donde se puede producir MiPrimeraExcepción

}

*...*

}

Aquí indicamos que este método puede lanzar (producir) una excepción, que además se llama **MiPrimeraExcepcion.**

Hasta ahora, cuando diseñamos un método para que realice una serie de operaciones y nos devuelva una variable de un tipo determinado, estamos confiando en que nos dará siempre esa variable. Puede ocurrir, sin embargo, que pase algo anormal en el cuerpo de ese metodo y se produzca un **error**; si esto ocurre no tenemos la variable que devuelve el método y nuestro programa se bloquearía. Es aquí donde entran las excepciones, que servirán para que nosotros en todo momento tengamos control sobre el programa. Si nos falla la variable, lo sabremos y actuaremos al respecto pero el programa no se bloquea , no se para.

No hemos de olvidar que las excepciones son objetos de una clase y gozan de las ventajas de tener una jerarquía.

No todos los errores deben ser listados al declarar un método. Las instancias de la clase **Error** no deben ser capturadas

.

Throwable

RuntimeException

# Exception

Error

De Libreria

Las excepciones que deriven de la rama Error como hemos comentado no se capturan , son tales como NoClassDefFoundError (no se encuentra una clase) , NoSuchMethodError (no se encuentra la definición de un método ) , y en general son todas las que terminan con la palabra **“Error”. (También llamadas excepciones de la “maquina virtual”).**

Después está la rama exception y dentro de esta hay otras dos ramas .Las excepciones que derivan de RunTimeException que se denominan **“excepciones del sistema”** dentro de estas hay 5 que veremos a continuación , y las **“de librería”** que son excepciones que se encuentran en librerías java , como por ejemplo IOException que se encuentra en el paquete **java.io** o MalformedURLException que se encuentra en la librería java.net.

**Excepciones del Sistema:**

**1. ArithmeticException :** Se produce cuando sucede un error de tipo aritmético por ejemplo cuando tratamos de dividir un número por cero , el programa no entiende lo que es y se para.

**2.NullPointerException:** Se produce cuando tratamos de acceder a una propiedad de un objeto que no existe todavía para el pogama .Ejemplo:

String cadena=null:

int longitud=cadena.length(); //Error porque el objeto no existe aun.

**3.ClasCastException:** Se produce cuando no hacemos correctamente un casting entre clases.

**4. ArrayIndexOutOfBoundsException:** Se produce cuando accedemos a un aposición de un array superior al número de elementos del array.

**5. NegativeSizeArrayIndexException:** Se produce cuando accedemos a un aposición de un array inferior al elemento cero del array , es decir , a una posición negativa del array.

### Uso práctico de excepciones.

Estudiemos ahora como usar las excepciones. Recordemos el método que definimos anteriormente denominado unMetodoExcepcional():

public class MiPrimeraClaseExcepcional {

public void unMetodoExcepional() **throws** **MiPrimeraExcepcion** {

...

}

...

}

Vamos a tratar de llamarlo:

public void otroMetodoExcepcional() throws MiPrimeraExcection {

MiPrimeraClaseExcepcional mPCE = new MiPrimeraClaseExcepcional();

mPCE.unMetodoExcepcional();

}

Si no tratamos aquí la excepción **MiPrimeraExcepcion** al menos hemos de lanzarla (indicar)a los llamadores de nuestro método otroMetodoExcepcional() (lo hacemos mediante la sentencia **throws**) y hemos de hacerlo así porque estamos llamando al método unMetodoExcepcional() que puede producir dicha excepción.

Si ahora quisiéramos nosotros mismos manejar la excepción lo haríamos de una forma parecida a la que sigue (solo la manejamos nosotros):

public void metodoResponsable() {

MiPrimeraClaseExcepcional mPCE = new MiPrimeraClaseExcepcional();

**try {**

mPCE.unMetodoExcepcional();

**}**

**catch (MiprimeraExcepcion m) {**

// aquí haríamos las acciones pertinentes

// si se produjera la excepción MiPrimeraExcepcion

**}**

}

La traducción de la sentencia **try** podría ser:

intenta ejecutar el código de dentro de las llaves y si hay excepciones lánzalas que yo las uniré a los manejadores para que se ocupen de ellas.

Generalizando lo que hemos hecho podríamos tener tantas cláusulas **catch** al final de la sentencia try como necesitemos. Y cada una nos permitiría manejar una instancia de una de las clases listadas dentro de los paréntesis o de sus subclases (recordemos que las excepciones son clases).

**Lanzamiento de excepciones .Cláusula throw**

Para completar los casos, si quisiéramos manejar la excepción pero además pasársela a aquel que llame a nuestro método, escribiríamos:

public void metodoResponsable() throws MiprimeraExcepcion {

MiPrimeraClaseExcepcional mPCE = new MiPrimeraClaseExcepcional();

**try** {

mPCE.unMetodoExcepcional();

}

**catch** (MiPrimeraExcepcion m) {

// aquí haríamos las acciones pertinentes

// si se produjera la excepción MiPrimeraExcepcion

**throw m;** // relanzamos la excepción

}

}

Esto va a funcionar porque las excepciones pueden anidarse. Va a ser interesante operar de esta forma porque así nosotros estamos tratando para nosotros la excepción pero además se la pasamos a aquel que use nuestro método para que el realice sus oportunas acciones. El sistema de por si trata todas las excepciones que no son cogidas (tratadas desde catch) pero lo hace abortando el programa y mostrando un mensaje de error.

Nosotros también podemos lanzar excepciones si en algún punto del programa detectásemos que algo no funciona bien:

public class MiPrimeraClaseExcepcional {

public void unMetodoExcepional() throws MiprimeraExcepcion {

...

if (algoInusualOcurrio()) {

**throw new MiPrimeraExcepcion();**

// la ejecucion ya no llegaría aquí

}

}

...

}

Esta vez **throw** actúa de forma similar a **break**, salta sin ejecutar las líneas de código inmediatamente a continuación.

Las excepciones son una potente herramienta para dividir el amplio espacio de posibles errores en partes mas manejables. Se pueden construir cadenas como las del siguiente ejemplo:

try {

algunMetodoExcepcional();

}

catch (NullPointerException n) {

// Tratamiento

}

catch (RuntimeException r) {

// Tratamiento

}

catch (IOException i) {

// Tratamiento

}

catch (Exception e) {

Tratamiento que se realizará siempre en caso de que el código del programa no entre por ninguno de los catch anteriores , esto funciona como un comodín , siempre al final ponemos la clase más superior (Exception) y así nos aseguramos de que siempre se ejecutara algún catch en caso de error.

}

El listar primero las subclases e ir subiendo en la jerarquía es para que las clases más especificas traten las excepciones más especificas y que las más generales sean manejadas por la parte superior de la jerarquía.

Mediante estas cadenas y aprovechando que disponemos de una jerarquía también para las excepciones podemos crearnos nuestro propio árbol para estas y tratar de forma precisa cada excepción.

Falta, como no, aplicar esto pero utilizando interfaces para el caso en el que la jerarquía típica se quede corta. No entramos en ello de momento.

Los pasos para lanzar una excepción son los siguientes:

1. Se crea el objeto del tipo excepción.
2. Con la palabra reservada **throw** se lanza.

class Demothrow {

static void ProcDemo() {

try { **throw new NullPointerException (“Mi Error”);** }

catch (NullPointerException ex) {

System.out.println (“Captura en ProcDemo”);

**throw** ex;

}

} // fin ProcDemo

public static void main(String [] args) {

try {ProcDemo();}

catch(NullPointerException ex) {

System.out.println(“Capturada en main”+ex);

}

} // fin main

} // fin de clase

Si una función no captura una excepción, la excepción viaja por la pila de llamada a funciones hasta que una función la captura o finaliza la ejecución del programa:

main() calula() divide() // Error

try { calcula() }

catch() { . . . }

En un bloque try se pueden definir varios tipos de excepciones:

class MultiCatch {

public static void main(String [] args) {

try { int a = args.length;

int b = 42 / a; //AritmethicException

int[] c = {2, 8};

c[a] = 99; //IndexOutOfBoundsException

}

catch (AritmethicException ex) {

System.out.println (“No hay argumentos”+ex);

}

catch (IndexOutOfBoundsException ex) {

System.out.println (“Demasiado argumentos”);

}

catch (Exception ex) { // Para demás excepciones

System.out.println (“Error desconocido”+ex);

}

} // fin de main

} // fin de clase

### Limitaciones para el programador.



Imgm2\_7

A la hora de sobreescribir métodos, ahora hemos de tener ya en cuenta las excepciones y debemos coger (mediante catch) las excepciones necesarias.

Un ejemplo no valido podía ser:

public class MiClaseIlegal {

public String toString() {

algunMetodoExcepcional();

... // devuelve algún String

}

}

La superclase Object definió el método toString() sin cláusulas throw y nosotros debemos mantener esa restricción. Por eso no podemos llamar a algunMetodoExcepcional() que genera errores y excepciones y algunas de estos deben ser cogidas en su respectiva sentencia catch. Un ejemplo que si seria valido puede ser el siguiente:

public class MiClase {

public String toString() {

try {

algunMetodoExcepcional();

}

catch (IOException e) {

}

catch (MiPrimeraExcepcion m) {

}

... // devuelve algún String

}

}

Hemos cogido las dos excepciones pero no hemos hecho nada con ellas, esto es legal, pero no será siempre lo correcto. Habremos de añadir un comportamiento no trivial que hará nuestro programa más robusto.

**2.13 Programación básica de Applets y componentes gráficos**

****

Imgm2\_8

Los Applets de Java son programas java que se ejecutan dentro de una página HTML ( página web). Aunque Google Chrome ya los está dejando de implementar navegadores como Internet Explorer , Opera y Firefox los siguen utilizando .

Además estamentos públicos muy importantes los siguen implementando en funciones tales como reservar citas para renovar dni y pasaporte, certificados digitales para el Ministerios , etc…

A nivel pedagógico resultan ser programas muy instructivos a la hora de aprender el lenguaje java.

**Características de un Applet . Ciclo de vida**

Cabe primeramente diferenciar una aplicación de un Applet. Las aplicaciones Java funcionan por si solas usando el interprete java.exe. Sin embargo, los Applets Java deben ejecutarse desde un browser ( un navegador ) del WWW. Los Applets son llamados desde documentos HTML en el navegador.

Debido a que el Applets viajan a todas partes, si estos portaran virus, podrían dañar los sistemas de archivos de ordenadores clientes, la JVM define sobre los Applets algunas restricciones de seguridad (SANDBOX SECURITY MODEL):

1. Los Applets no pueden leer o escribir sobre el sistema de ficheros, excepto en directorios específicos.

2. No pueden comunicarse con otro servidor que no sea el que guarda el Applet. Esto a veces se puede configurar desde el browser.

3. Los Applets no pueden ejecutar programas del sistema de lectura.

4. Los Applets no pueden cargar programas nativos de la plataforma local.

Para crear un Applet hemos de crear una subclase de la clase Applet que está dentro del paquete java.applet . La sintaxis es:

**import java.applet.Applet;**

**public class nombreClase extends Applet {**

**...**

**}**

La clase original debe ser pública .

##### Applet

##### MiApplet

Para comenzar a trabajar con Applets, se necesita algo más que el main() usado en las aplicaciones. Un Applet necesita hacer algunas actividades asociadas a eventos importantes en el ciclo de vida del Applet y que están directamente relacionados con el funcionamiento del browser Así cada actividad, tiene asociado su método.

Por defecto, estos métodos que se encuentran en la clase Applet no hacen nada y seremos nosotros quienes los sobrescribamos y les digamos que hagan algo

Los principales métodos a tener en cuenta son:

### Inicialización.

Nada más el Applet se carga. Entre lo que se puede hacer aquí destacar:

a. Crear objetos que necesitemos.   
b. Inicializar valores.   
c. Cargar imágenes.   
d. Cargar fuentes.

El método a sobreescribir es:

**public void init() {**

**...**

**}**

El proceso de inicialización ocurre **sólo una vez** en la vida del Applet.

### Arranque.

Ocurre después de inicializar o después de una parada previa. Este proceso puede ocurrir muchas veces durante la vida de un Applet.

El método a sobreescribir es:

**public void start() {**

**...**

**}**

### Parada.

Ocurre cuando el usuario abandona la pagina que incluya este Applet o cuando llamamos nosotros a stop():

**public void stop() {**

**...**

**}**

### Destrucción.

Sirve para limpiar de memoria antes de abandónala desde el navegador. Normalmente no sobreescribiremos este método a menos que queramos liberar recursos específicos.

El método a sobreescribir es:

**public void destroy() {**

**...**

**}**

La diferencia entre destroy() y finalize() está en que finalize() es más genérico y sirve para un objeto cualquiera mientras que destroy() es propia del Applet.

### El método paint().

Define como un Applet dibuja algo en la pantalla. Este proceso puede ocurrir cientos de veces en la vida del Applet. El método a sobreescribir es:

**public void paint(Graphics g) {**

**...**

**}**

Dentro del código de nuestra Applet hemos de incluir:

**import java.awt.Graphics;**

El método paint( Gaphics g) recibe un objeto g de la clase Graphics. Este objeto se denomina contexto gráfico y se va a utilizar obligatoriamente siempre que queramos dibujar algo dentro del paint .

EJEMPLO:

public void paint(Graphics g)

{

g.drawString (“Hola desde java”, 10,10);

}

En este ejemplo dibujamos un String (Hola desde java ) en las coordenadas x=10 y=10 dentro del applet . (Después se verán los métodos más importantes para dibujar en el applet).

### Inclusión de un Applet en una página web. Paso de parámetros.

### La etiqueta <APPLET>.

Para incluir un Applet en una página web hay una etiqueta especifica; la etiqueta **<APPLET>.** Dentro del cuerpo de nuestra página web y allí donde nos interese, pondríamos el Applet como:

**<APPLET CODE="NombreApplet.class" WIDTH=200 HEIGHT=60>**

**Texto Alternativo**

**</APPLET>**

Detrás de la palabra **CODE** se pone entre comillas el nombre de la clase Applet que queremos. De este modo suponemos que el Applet (.class) se encuentra en el directorio donde está el documento HTML. En el nombre hay que incluir la extensión .class.

El **texto alternativo** sirve para sustituir al Applet si el navegador no entiende la etiqueta <APPLET>. Es interesante hacer uso de esta opción para usuarios de nuestros Applets que no dispongan de un browser lo suficiente potente.

**WIDTH y HEIGHT** serían respectivamente el ancho y el alto del applet en píxeles.

### Atributos de la etiqueta <APPLET>.

La etiqueta APPLET tiene varios atributos que se colocarían dentro de la misma etiqueta <APPLET ..... atributos> </APPLET>. Los atributos son:

### ALIGN.

Define como debe alinearse el Applet. Puede tener 9 valores. Para alinear en horizontal:

**ALIGN=LEFT** Sitúa el Applet al margen izquierdo de la pagina web y todo el texto que sigue a la derecha del Applet.

**ALIGN=RIGHT** Sitúa el Applet al margen derecho de la pagina web y todo el texto que sigue a la izquierda del Applet.

Si queremos que el texto siga debajo usaremos la etiqueta <BR> o su familia <BR CLEAR=opcion>, donde opción puede ser LEFT, RIGHT o ALL.

Para alinear en vertical usaremos:

**ALIGN=TEXTTOP** Alinea la parte superior del Applet con la parte superior del texto en línea.

**ALIGN=TOP** Alinea la parte superior del Applet con la parte superior del texto/imagen en línea.

**ALIGN=ABSMIDDLE** Alinea el centro del Applet con el centro del texto/imagen en línea.

**ALIGN=MIDDLE** Alinea el centro del Applet con la base del texto en línea.

**ALIGN=BASELINE** Alinea la parte inferior del Applet con la base del texto en línea.

**ALIGN=BOTTOM** Equivale a ALIGN=BASELINE.

**ALIGN=ABSBOTTOM** Alinea la parte inferior del Applet con la parte más baja del texto/imagen/Applet en línea.

### HSPACE y VSPACE.

**HSPACE** Espacio horizontal en píxeles entre un Applet y el texto adjunto. Es el valor que deja tanto a izquierda como a derecha.

**VSPACE** Espacio vertical en píxeles entre un Applet y el texto adjunto. Es el valor que deja tanto arriba como abajo.

En definitiva, es como si la porción ocupada por el Applet fuera:

Anchura: HSPACE+WIDTH+HSPACE

Altura: VSPACE+HEIGHT+VSPACE

### CODE y CODEBASE.

**CODE** sirve para indicar la clase del Applet si esta está en el mismo directorio que el archivo HTML en uso.

**CODEBASE** es similar, pero dando el URL de donde se encuentra el Applet bien absoluto o bien relativo al archivo HTML actual. Es decir, haríamos:

<APPLET CODE="nombreClase.class" CODEBASE="clases"

WIDTH=100 HEIGHT=100> Java Applet Here </APPLET>

### NAME y ALT.

**NAME** sirve para nombrar el Applet caso de tener varios y tener que trabajar con ellos.

**ALT** se utiliza opcionalmente para indicar un texto alternativo por si no se carga el Applet (es lo mismo que el texto substitutivo antes comentado).

### La etiqueta <EMBED>.

Actualmente los únicos archivos ejecutables embebidos en documentos HTML son los Applets de Java y los controles Actives de Microsoft. Debido a que pronto surgirán nuevos ejecutables en nuevos lenguajes de programación, estos para estar embebidos en los HTML tendrían su nueva etiqueta al igual que Java tiene <APPLET>. Para unificar criterios se define una etiqueta genérica denominada **<EMBED>** para representar a cualquier recurso que pueda ser cargado dentro del documento HTML.

La etiqueta <EMBED> hereda todas las propiedades que hasta ahora tenía la etiqueta <APPLET> y por ello tiene la misma sintaxis y los mismos atributos que esta. Solo cambia que los atributos CODE y CODEBASE se unen en un único atributo **SCR**. Si tenemos por ejemplo:

**<APPLET CODE="nombreClase.class" CODEBASE="clases"**

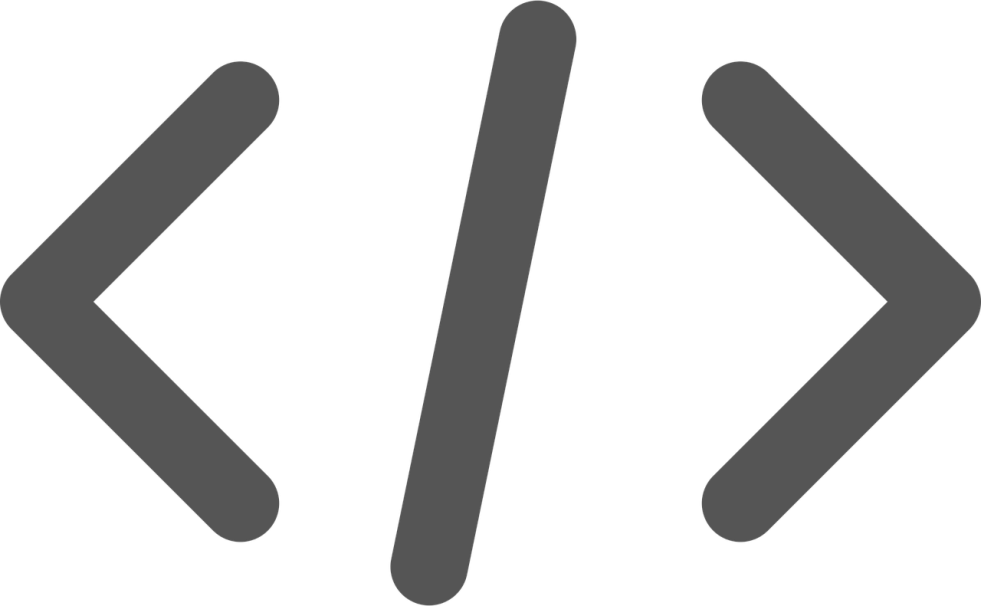
**WIDTH=100 HEIGHT=100> Java Applet </APPLET>**

Utilizando la etiqueta <EMBED> escribiríamos:

**<EMBED SCR="clases/nombreClase.class"**

**WIDTH=100 HEIGHT=100> Java Applet </EMBED>**

### Pasando parámetros a Applets.



Imgm2\_9

A las Applets también se le pueden pasar parámetros, para ello es necesario:

1. Una etiqueta especial en el archivo HTML.

2. Código en nuestro Applet para tratar estos parámetros.

Los parámetros en un Applet van en dos partes: un **nombre** y su **valor**. La etiqueta especial del lenguaje HTML es **<PARAM>.** Su uso es:

<APPLET CODE="nombreClase" WIDTH=100 HEIGHT=100>

<PARAM NAME=”nombre” VALUE="Luis">

<PARAM NAME=”ciudad” VALUE="Madrid">

*...*

</APPLET>

Los parámetros se le pasan al Applet en su inicialización, en el método init() y accederemos al valor de los parámetros pasados al applet dentro del programa a través del método **getParameter().**

Este método requiere un argumento que es un String representando el nombre del parámetro y devuelve el valor del mismo que será otro String.

Si no nos interesara un String, haríamos una conversión de tipo. Así en nuestro caso, en init(), para coger los parámetros anteriores se haría:

public void init() {

String nombre = getParameter("nombre");

String ciudad = getParameter("ciudad");

...

}

Hemos de tener presente que el lenguaje es case-sensitive.

Si el parámetro que esperamos no está especificado en el fichero HTML, el método getParameter() devuelve null, por ello es interesante y ante una respuesta null, que nuestro programa utilice un valor por defecto:

if (nombre == null*)* {

nombre = "Luis";

}

Si por ejemplo queremos hacer una conversión de String a entero:

int tamaño;

String s = getParameter("tamaño");

if (s == null) {

tamaño = 12; // valor por defecto

}

else { tamaño = Integer.parseInt(s); }

### Ejecutar Applets. La clase Graphics

### Para desarrollar y ejecutar un applet primeramente realizamos el fichero java con el código del applet por ejemplo:

### import java.applet.Applet;

### import java.awt.\*;

### public class MiApplet extends Applet

### {

### public void init( )

### {

### setBackground (Color.red);//pone el fondo del applet en rojo

### }

### public void paint(Graphics g)

### {

### g.drawString(“Hola desde java...”,20,20); // Escribimos en el applet

### }

### }

### Después de construir el fichero .java se compila como siempre con javac:

### C:> javac MiApplet.java

### Después construimos el archivo MiApplet.htm e insertamos la clase ( fichero .class) en el atributo code:

### <html>

### <body>

### <applet code="MiApplet.class" width="200" height="200">

### </applet>

### </body>

### </html>

### Por último abrimos el Internet Explorer (browser) y abrimos la página MiApplet.htm:

### 

Imgm2\_10

### Este es el resultado final.

### También es posible poder visualizar el applet sin un navegado , con la ayuda del appletviewer la herramienta de visualización de applets incluida en el JDK.

### Lo haríamos de la siguiente forma:

### C:>appletviewer MiApplet.htm

### La salida sería similar:

### 

Imgm2\_11

**La clase Graphics**

Para dibujar algo en nuestro Applet, no tenemos que crear siempre un objeto de esta clase, sino que vamos a hacer uso del objeto Graphics de la función paint(), y dibujando sobre este objeto, el resultado aparecerá en nuestro Applet. Este objeto es pasado desde el navegador al propio Applet

Para hacer uso de la clase Graphics en nuestro Applet deberemos incluir al principio de nuestro Applet, como ya dijimos el paquete que contiene a esta clase:

**import java.awt.Graphics;**

El origen de coordenadas del sistema de gráficos tiene su origen (0,0) en la esquina superior izquierda. Los valores positivos crecen hacia abajo y hacia la derecha. Los píxeles tienen asociados valores enteros, nunca fraccionarios.

**+X**

**( 0,0 )**

**+Y**

Veamos ahora algunas clases para dibujar: líneas, rectángulos, polígonos, arcos, ...

### Líneas.

Basta usar el método **drawLine().** Se le pasan 4 argumentos, las coordenadas del punto origen x1, y1 y las del punto destino x2 e y2.

public void paint(Graphics g) {

g.drawLine(x1,y1,x2,y2);

...

}

### Rectángulos.

Existe la posibilidad de dibujar tres tipos de rectángulos:

a. Rectángulos planos.   
b. Rectángulos redondeados (esquinas redondeadas).   
c. Rectángulos tridimensionales (forma sombreada).

Para cada una de estas formas, disponemos de dos métodos, una que lo dibuja y otro que lo dibuja y rellena. Los parámetros que se le pasan a estos métodos son las coordenadas (x,y) de la esquina superior izquierda, la anchura (w) y la altura (h).

a. Rectángulos planos:

**drawRect(x,y,w,h); Lo dibuja**

**fillRect(x,y,w,h); Lo dibuja y lo rellena**

b. Rectángulos redondeados:

**drawRoundRect(x,y,w,h,hp,vp);**

**fillRoundRect(x,y,w,h,hp,vp);**

,hp y vp marcan el tamaño del recorte en horizontal y vertical respectivamente , es decir la curvatura de las esquinas redondeadas

**vp**

**hp**

c. Rectángulos tridimensionales:

**draw3DRect(x,y,w,h,eff);**

**fill3DRect(x,y,w,h,eff);**

El argumento eff es un boolean que indica si el efecto del rectángulo es hacia afuera (true) o hacia dentro (false).

EJEMPLO:

import java.applet.Applet;

import java.awt.\*;

public class MiApplet extends Applet

{

public void init( )

{

setBackground (Color.orange);//pone el fondo del applet en naranja

}

public void paint(Graphics g)

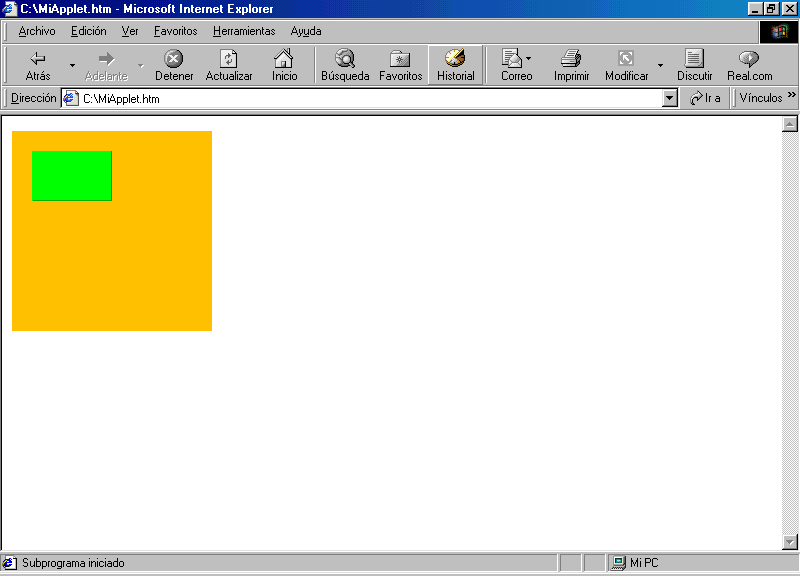
{

g.setColor(Color.green);//rellenamos d ecolr verde el rectángulo

g.fill3DRect(20,20,80,50,true); // Escribimos en el applet

}

}  
Salida:



Imgm2\_12

### Polígonos.

Para dibujar un polígono necesitamos de un set de puntos (x,y). El método dibuja líneas de punto a punto siguiendo el orden dado.

Como ocurría en los rectángulos, podemos dibujar y también rellenar:

**drawPolygon();**

**fillPolygon();**

El método drawPolygon() no cierra el polígono, así si queremos que se cierre hemos de incluir el primer punto como ultimo punto también. El método fillPolygon() si une el primer punto con el ultimo.

Los argumentos se le pueden pasar de dos formas dado que tenemos el método sobrecargado:

a. Tres argumentos: array con las coordenadas x, array con las coordenadas y, valor entero con el numero de puntos totales.

b. Un argumento: un objeto polígono, perteneciente a la clase Polygon.

Un objeto polígono se puede crear con la directiva new:

Polygon poli = new Polygon();

Polygon poli = new Polygon(vectorx,vectory,numPuntos);

y podemos añadirle puntos al polígono:

poli.addPoint(xNueva,yNueva);

### Ovalos.

Para dibujar elipses y círculos. Los parámetros que se le pasan vienen a ser los mismos que para dibujar rectángulos planos:

**drawOval(x,y,ejehoriz,ejevert);**

**fillOval(x,y,ejehoriz,ejevert);**

Se dibuja el óvalo que está circunscrito en el rectángulo que hemos definido.

### Arcos.

Para dibujar arcos utilizaremos la siguiente función:

**drawArc(x,y,w,h,ang,grad);**

**fillArc(x,y,w,h,ang,grad);**

donde

x ordenada esquina superior izquierda   
y abcysa esquina superior izquierda   
w anchura   
h altura   
ang ángulo de comienzo del arco   
grad grados que dibuja hasta parar.

El método fillArc() rellena como si el arco se tratara de una porción de una tarta. Para los contar los grados debemos tomar una circunferencia de referencia y situamos el valor 0 donde un reloj marca las 3 y los grados aumentan en sentido antihorario.

**90º**

**180º**

**0º**

**270º**

Ejemplo de un arco con relleno:

import java.applet.Applet;

import java.awt.\*;

public class MiApplet extends Applet

{

public void init( )

{

setBackground (Color.blue);//pone el fondo del applet en naranja

}

public void paint(Graphics g)

{

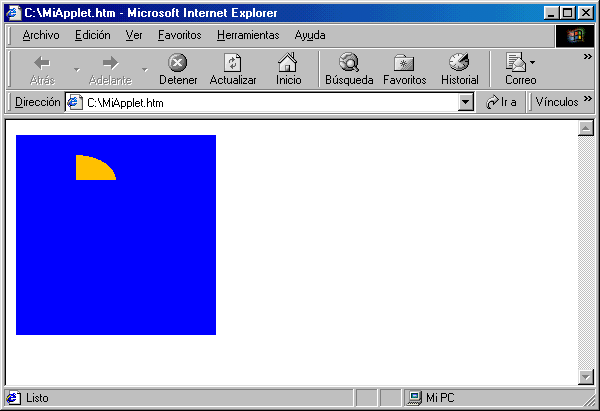
g.setColor(Color.orange);//rellenamos d ecolr verde el rectángulo

g.fillArc(20,20,80,50,0,90); // Escribimos en el applet

}

}

Salida:



Imgm2\_13

Finalmente y en relación con los gráficos, es posible copiar una porción de un gráfico o borrarlo. Los métodos son:

**copyArea(x,y,w,h,distx,disty);**

**clearRect(x,y,w,h);**

Donde x,y,w,h son los parámetros del área rectangular y donde distx y disty son los movimientos en x e y hacia donde vamos a copiar el área dada. Es decir, las nuevas coordenadas de la esquina superior izquierda del rectángulo son:

xnueva = x + distx

ynueva = y + disty

El método clearRect() dibuja un rectángulo con el color de fondo. Si queremos borrar el Applet completa podemos hacer uso de la función size() de la siguiente forma:

**g.clearRect(0,0,size().width,size().height);**

**size().width:** Devuelve el ancho del Applet

**size().height:**Devuelve el alto del Applet

### Texto y fuentes.

Un objeto fuente se caracteriza por su nombre, su estilo y su número de puntos.

**Fuentes**: "TimesRoman", "Courier", "Helvetica", ...

**Estilos:** Font.PLAIN, Font.BOLD, Font.ITALIC, ...

**Tamaño:** 12, 24, 36, ...

Los estilos se pueden combinar y crear un nuevo estilo:

Font.BOLD + Font.ITALIC Texto en negrita y cursiva

Para usar una fuente debemos cargar el paquete: **import java.awt.Font;**

Para asigna la fuente al Applet usamos el método de la clase Graphics setFont() que recibe un objeto de tipo Font.

EJEMPLO:

import java.awt.Font;

...

...

public void paint(Graphics g)

{

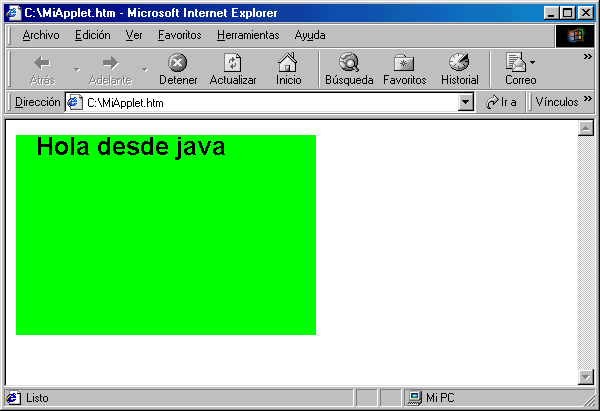
Font f=new Font(“Arial”,Font.BOLD,25);

g.setFont(f); // Para asignar la fuente al Applet

g.drawString(“Hola desde java”,20,20);

}

}  
Salida:



Imgm2\_14

### Dibujando caracteres y cadenas.

Usaremos los métodos:

**drawChars();**

**drawString();**

public void paint(Graphics g) {

Font f = new Font("TimesRoman", Font.BOLD, 24);

g.setFont(f);

g.drawString("Esta es una fuente grande. ",x,y);

...

}

Los dos valores numéricos x e y indican la posición donde comienza a dibujar la cadena de caracteres.

El método drawChars funciona análogamente, pero se la pasa un array de caracteres. Tiene 5 argumentos:

g.drawChars(array,intFirst,intLast,x,y);

Los nuevos valores intFirst e intLast indican el primer y el último elemento del array que se va a dibujar.

### Obteniendo información sobre una fuente.

Existen una serie de métodos que nos proporcionan información sobre las fuentes:

|  |  |  |
| --- | --- | --- |
| **Método** | Objeto | Acción |
| getFont() | Graphics | devuelve el objeto fuente actual |
| getName() | Font | devuelve el nombre de la fuente: String |
| getSize() | Font | devuelve el tamaño de la fuente: int |
| getStyle() | Font | devuelve el estilo:  0 si es PLAIN  1 si es BOLD  2 si es ITALIC  3 si es BOLD ITALIC |
| isPlain() | Font | devuelve true si la fuente es plain |
| isBold() | Font | devuelve true si la fuente es bold |
| isItalic() | Font | devuelve true si la fuente es italic |

Existe otra clase denominada **FontMetrics** que da información más específica sobre una fuente. Para crear un objeto de este tipo haremos:

Font f = new Font("TimesRoman", Font.BOLD, 24);

FontMetrics fmetrics = getFontMetrics(f);

g.setFont(f);

Algunas de las funciones de esta nueva clase son:

|  |  |
| --- | --- |
| **Método** | **Significado** |
| stringWidth(string) | devuelve la anchura total del string en píxeles |
| charWidth(char) | devuelve la anchura total del carácter en píxeles |
| getAscent() | devuelve la altura desde la base en píxeles |
| getDescent() | devuelve la altura desde la base a bajo en píxeles |
| getLeading() | devuelve la diferencia mínima entre dos líneas |
| getHeight() | altura total: descendiente más ascendiente. |

### Color.

Usando la clase **Color** podremos escribir y dibujar en varios colores. Por ello para dibujar en color crearemos una instancia de la clase Color. Esta clase define ciertos colores almacenados en variables pero si queremos otros colores de la paleta menos populares podemos hacerlo igualmente. Los colores más populares son:

|  |  |
| --- | --- |
| **Nombre** | **Valor RGB** |
| Color.white  Color.black  Color.lightGray  Color.gray  Color.darkGray  Color.red  Color.green  Color.blue  Color.yellow  Color.magenta  Color.cyan  Color.pink  Color.orange | 255,255,255  0,0,0  192,192,192  128,128,128  64,64,64  255,0,0  0,255,0  0,0,255  255,255,0  255,0,255  0,255,255  255,175,175  255,200,0 |

Si queremos cualquier otro color o de forma más general haremos:

Color c = new Color(R,G,B);

Los valores de R, G y B varían entre 0 y 255.

Para dibujar en un cierto color, en el método paint() , por ejemplo, haremos referencia a este como sigue:

g.setColot(Color.green);

g.setColor(c);

Además podemos fijar los colores del fondo y del plano principal (texto e imágenes) utilizando los métodos:

**setBackground(Color.white);**

**setForeground(Color.black);**

Además de configurar los colores, podemos leer sus valores:

**getColor();** Devuelve el color

**getBackground();** Devuelve el color de fondo

**getForeground();** Devuelve el color del primer plano

### Ventanas AWT y componentes

Una ventana Java es básicamente, un conjunto de componentes anidados. Estos componentes pueden ser más ventanas, barras de menú, botones y/o contenedores que incluirán otros componentes.

Estos son el grupo de componentes que usaremos dentro del AWT:

1. **Contenedores**. Son componentes genéricos del AWT que pueden contener a otros componentes (incluso containers). La forma más común es el panel (es una región diferenciada donde se incluyen componentes).

2. **Lienzo**s. Son simplemente superficies de dibujo. Son útiles para dibujar imágenes y otras operaciones gráficas.

3. **Componentes UI**. Pueden incluir botones, listas, menús desplegables, checkboxes y otros elementos de un interfaz de usuario.

4. **Componentes de construcción de una ventana**. Incluye ventanas, frames, barras de menú y diálogos.

La jerarquía de clases del AWT queda definida en la siguiente figura:

**Component**

**Canvas**

**Container**

**Panel**

**Applet**

**Window**

**-Frame**

**-Dialog**

**TextComponent**

**- TextField**

**UI Components (Ejemplo Button)**

### Los componentes básicos del interfaz de usuario.

El componente básico es el componente UI. Podemos crearlos y añadirlos a nuestro Applet sin conocer nada sobre contenedores ni paneles. Como nuestro Applet es ya en si un contenedor (deriva de Panel) puede contener componentes AWT y entre estos , los componentes UI.

PANEL

APPLET

El modo de trabajar con estos componentes es sencillo:

1. Creamos el componente. Se hace en el método init().

2. Sumamos el componente al panel que queramos.( add)

3. Le asignamos una localización. Por defecto se colocaría en el centro del panel y se aplica la regla de añadir los nuevos componentes de izquierda a derecha y en filas (uno detrás de otro).

4. Asignarle una función/método al evento asociado a la activación del componente.

Estudiemos ahora cada componente por separado:

### Etiquetas.

Son cadenas de texto que sirven para nombrar o etiquetar los otros componentes. La ventaja respecto al texto propiamente es que no tenemos que repintarlos nosotros y además se alinean en el panel.

Para crear una etiqueta usaremos uno de los siguientes constructores:

**Label()** crea una etiqueta vacía, su texto se alinearía a la izquierda.

**Label(String)** crea una etiqueta con el string que le pasamos alineado a la izquierda.

**Label(String, int)** crea una etiqueta con el string que le pasamos y la alinea según el valor que le pasamos. Los alineamientos posibles son 3. Así la variable int será: Label.RIGHT, Label.LEFT o Label.CENTER.

La fuente que escribe el contenido de la etiqueta depende de la fuente que hayamos configurado con setFont(). Ejemplo:

**add( new Label("Texto centrado", Label.CENTER));**

Una vez definida una etiqueta podemos utilizar métodos predefinidos en la clase Label como son:

|  |  |
| --- | --- |
| **Método** | **Acción** |
| getText() | Devuelve un string con el contenido de esta . |
| setText(String) | Cambia el texto de la etiqueta a este. |
| getAlignment() | Devuelve el alineamiento:  0 si Label.LEFT  1 si Label.CENTER 2 si Label.RIGHT |
| setAlignment(int) | Cambia el alineamiento según el int dado. |

### Botones.

Elemento que sirve para realizar una acción cuando es pulsado. Para crear un botón usaremos uno de estos constructores:

**Button()** crea un botón vacío.

**Button(String)** crea un botón con el string como etiqueta.

Una vez creado el botón podemos usar estos métodos con él:

|  |  |
| --- | --- |
| **Método** | **Acción** |
| getLabel() | devuelve su etiqueta. |
| setLabel(string) | asignamos una nueva etiqueta al botón. |

El tamaño del botón lo da la fuente de texto que lo etiqueta.

### Checkboxes.

Son interfaces de usuario que pueden estar activadas o desactivadas. Son pequeñas cajas cuadradas( casillas de verificación) que se pueden marcar o desmarcar.

Las checkboxes pueden ser utilizadas de dos formas:

1. Checkboxes **no exclusivas**. Cualquiera de un grupo puede estar seleccionada o no.

2. Checkboxes **exclusivas**. Una sola checkbox de un grupo puede estar activada. Se llaman radio buttons o también checkbox groups.

En este apartado veremos solo las checkboxes no exclusivas. Para su creación utilizaremos uno de los siguientes constructores:

**Checkbox()** crea una checkbox vacía.

**Checkbox(String)** crea una checkbox con la etiqueta que le pasamos.

**Checkbox(String, null, boolean)** crea la checkbox con la etiqueta que le pasamos y que estará activada/desactivada si el boolean vale true/false. El valor null es característico de estos componentes.

Para los checkboxes también se definen algunos métodos de utilidad:

|  |  |
| --- | --- |
| **Método** | **Acción** |
| getLabel() | devuelve su etiqueta. |
| setLabel(string) | asignamos una nueva etiqueta al botón. |
| getState() | devuelve true/false según este o no activada. |
| setState(boolean) | fija el estado al boolean que le pasemos. |

EJEMPLO:

import java.awt.\*;

import java.applet.Applet;

public class Cajas extends Applet

{

Label l1=new Label("Comedia");

Label l2=new Label("Oeste");

Label l3=new Label("Terror");

Checkbox c1=new Checkbox();

Checkbox c2=new Checkbox();

Checkbox c3=new Checkbox();

public void init()

{

setBackground(Color.yellow);

add(l1);

add(c1);

add(l2);

add(c2);

add(l3);

add(c3);

}

}

Código HTML:

<html>

<body>

<applet

code="Cajas.class"

width=200

height=200>

</applet>

</body>

</html>

Salida:



Imgm2\_15

### RadioButtons.

Son un grupo de checkboxes exclusivas. Para crearlos utilizaremos el siguiente constructor mediante un ejemplo:

**CheckboxGroup cbg = new CheckboxGroup();**

**add(new Checkbox("Rojo",cbg,true));**

**add(new Checkbox("Azul",cbg,false));**

**add(new Checkbox("Verde",cbg,false));**

**add(new Checkbox("Amarillo",cbg,false));**

Además podemos utilizar otros métodos predefinidos para los checkbox componentes del radio button:

|  |  |
| --- | --- |
| **Método** | **Acción** |
| getCheckboxGroup() | conocer el grupo del checkbox. |
| setCheckboxGroup() | cambiar el grupo de un checkbox. |
| getCurrent() | saber que checkbox es el activado. |
| setCurrent(Checkbox) | para activar un checkbox en concreto. |

EJEMPLO:

import java.awt.\*;

import java.applet.Applet;

public class RadiosButons extends Applet

{

Label l1=new Label("Forma de pago:");

public void init()

{

setBackground (Color.pink);

CheckboxGroup cbg = new CheckboxGroup();

add(l1);

add(new Checkbox("Visa",cbg,true));

add(new Checkbox("Metálico",cbg,false));

add(new Checkbox("Cheque",cbg,false));

}

}

Código HTML:

<html>

<body>

<applet

code="RadiosButons.class"

width=200

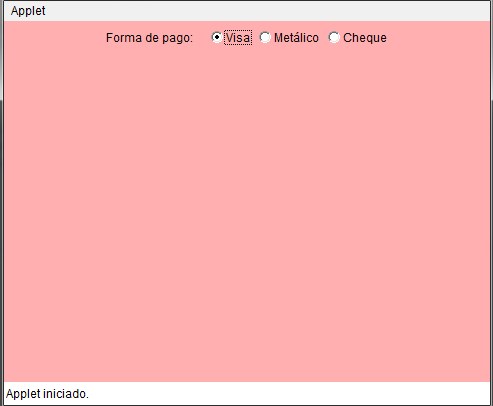
height=200>

</applet>

</body>

</html>

Salida:



Imgm2\_16

### Menús de selección.

Se trata de un menú desplegable que nos permiten seleccionar un ítem. Después el menú representa ese ítem en la pantalla.

Para crear un choice menú creamos un objeto de la clase Choice, luego le añadiremos los ítems y finalmente añadiremos el menú al panel:

**Choice c = new Choice();**

**c.addItem("Manzanas");**

**c.addItem("Naranjas");**

**c.addItem("Platanos");**

**c.addItem("Melocotones");**

**add(c);**

Estos menús solo permiten una selección por menú. Mas métodos para este componente son:

|  |  |
| --- | --- |
| **Método** | **Acción** |
| addItem(String) | añade el nuevo ítem con su etiqueta. |
| getitem(int) | devuelve el string del ítem indicado. [0,n-1] |
| countItems() | devuelve el numero de ítems del menú. |
| getSelectedIndex() | devuelve la posición del ítem seleccionado. |
| getSelectedItem() | devuelve el ítem seleccionado como string. |
| select(int) | selecciona el ítem que le pasamos. |
| select(String) | selecciona el ítem con el string pasado. |

**Campos de texto.**

Componentes que nos permiten entrar valores en forma de texto. Para crear una campo de texto, el constructor toma varias formas:

**TextField()** crea un campo de texto vacío de 0 caracteres de anchura.

**TextField(int)** crea un campo de texto vacío con el numero de caracteres que le pasamos.

**TextField(String**) crea el texto de tamaño 0, inicializado con el string que le pasamos.

**TextField(String, int**) crea el campo de texto con el string pasado y el tamaño int que le damos.

Después de crear el campo de texto lo sumamos al panel. Como ejemplo:

**TextField tf = new TextField("Hola desde Java",30);**

**add(tf);**

El campo de texto solo incluye el campo editable y no incluye ninguna etiqueta. Seremos nosotros los que incluyamos la nueva etiqueta por nuestra parte.

Los campos de texto solo ocupan una línea y están limitados a diferencia de las áreas de texto que hasta pueden tener barras de scroll.

Además podemos crear el campo de texto típico de las contraseñas que esconda el texto que se le entra. Para ello crearemos el campo de texto como tal y después teclearemos el setEchoCharacter('\*'). A modo de ejemplo:

**TextField tf = new TextField("Escribe algo",30);**

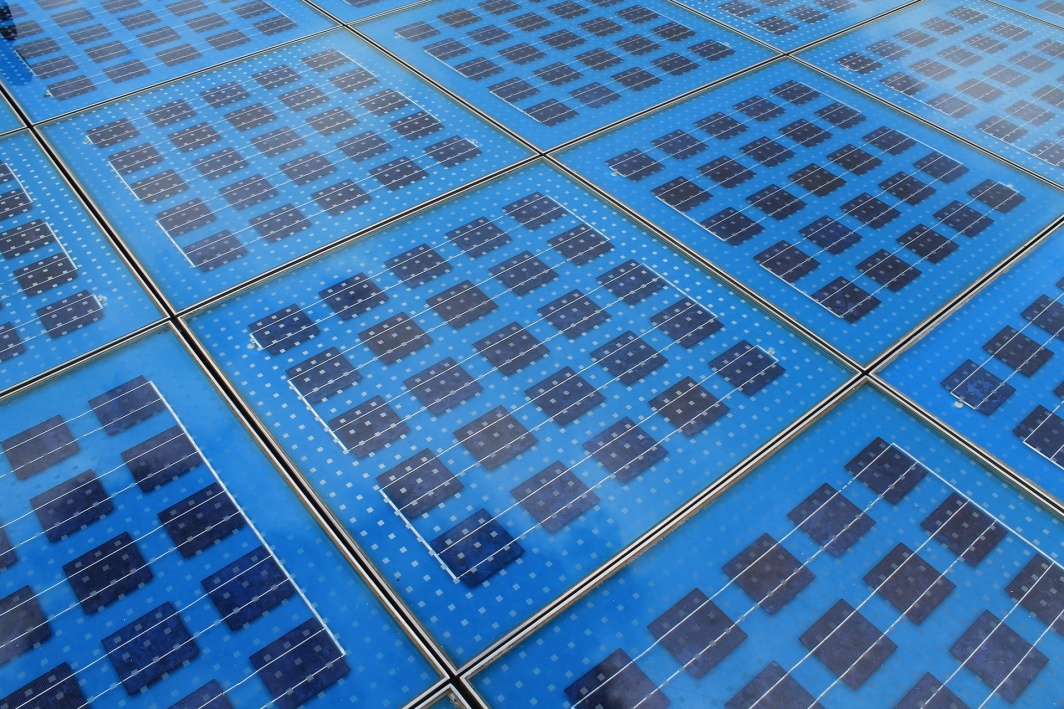
**tf.setEchoCharacter('\*');**

**add(tf);**

Los componentes campos de texto tienen más métodos predefinidos:

|  |  |
| --- | --- |
| **Método** | **Acción** |
| getText() | devuelve el contenido como un string. |
| setText(String) | pone el texto que pasamos dentro del campo. |
| getColumns() | devuelve la anchura del campo de texto. |
| select(int,int) | selecciona el texto entre las dos posiciones dadas como argumentos de entrada |
| selectAll() | selecciona todo el texto en el campo |
| isEditable() | devuelve true/false segun el texto sea editable o no. |
| setEditable(boolean) | true hace que el texto sea editable, false hace lo contrario. |
| getEchoChar() | devuelve el carácter utilizado como mascara. |
| echoCharIsSet() | devuelve true o false según el campo tenga un carácter mascara o no. |

### Paneles y Layouts.



Imgm2\_17

Queremos saber ahora como todos los componentes que hemos metido en el panel son arrancados y representados en pantalla. Java dispone de administradores de layout para que cada componente pueda ser mandado a pantalla.

No hemos de dibujar los componentes, el sistema AWT lo hace por nosotros, solo tendremos que dibujar las imágenes, gráficos, y animaciones que creemos.

Cada panel puede tener su propio administrador de layout. El AWT nos proporciona cinco básicos administradores de layout: **FlowLayout, GridLayout, GridBagLayout, BorderLayout y CardLayout.**

Para crear un administrador de layout utilizaremos el método setLayout() en el método init(). Por ejemplo:

**public void init() {**

**setLayout(new FlowLayout());**

**...**

**}**

Una vez el administrador de layout esta creado ya podemos comenzar a añadir componentes al panel. El orden de adición de los componentes es a menudo significante, dependiendo de cual administrador esta actualmente activo.

Veamos ahora los administradores de Layout.

### La clase Flowlayout.

Usando el flow layout los componentes se añaden al panel uno cada vez y fila a fila. Si un componente no ajusta en una fila se pasa a la siguiente. También realiza un alineamiento centrado por defecto de cada fila.

Para crear un layout básico con alineamiento centrado, usaremos la siguiente línea de código en nuestro método init():

**setLayout(new FlowLayout());**

, y si queremos otro alineamiento:

**setLayout(new FlowLayout(FlowLayout.LEFT));**

**setLayout(new FlowLayout(FlowLayout.RIGHT));**

y si queremos separaciones distintas de tres pixeles entre elementos que son los que hay por defecto haríamos (ejemplo para 10 pixeles en horizontal y en vertical):

**setLayout(new FlowLayout(FlowLayout.RIGHT), 10, 10);**

### El layout GridLayout.

El layout Grid ofrece un control más preciso a la hora de emplazar dentro del panel los componentes. Ahora se divide el panel en filas y columnas y cada uno de los componentes se sitúa en una celda en orden de izquierda a derecha y de arriba a abajo.

Para crear un layout de tipo Grid llamaremos en init() a:

**setLayout(new GridLayout(3,4));**

Si añadimos separaciones entre elementos en horizontal y vertical (10 y 15 respectivamente):

**setLayout(new GridLayout(3, 4, 10, 15);**

Mediante este ejemplo, se obtiene un Layout de 3 filas y 4 columnas con elementos separados 10 pixeles en horizontal y 15 en vertical. Con el GridLayout se obtiene siempre una rejilla con tamaños de celda iguales.

EJEMPLO:

import java.awt.\*;

import java.applet.Applet;

public class Grid extends Applet

{

Label l1=new Label("Celda 1");

Label l2=new Label("Celda 2");

Label l3=new Label("Celda 3");

Label l4=new Label("Celda 4");

public void init()

{

setLayout(new GridLayout(2,2));

add(l1);

add(l2);

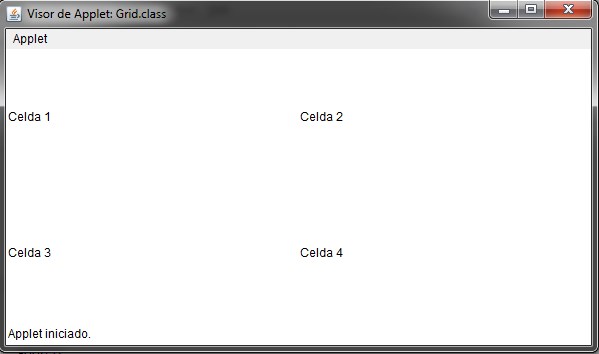
add(l3);

add(l4);

}

}

Salida:



Imgm2\_18

### El layout GridBagLayout.

Los layouts de la clase GridBagLayout son una variación de la clase GridLayout pero permiten variar la presentación de cada elemento en la rejilla. Estos usan una clase de ayuda denominada GridBagConstraints para indicar como debe ser formateado cada elemento en la rejilla.

El GridBagLayout es un administrador de layout más flexible que el GridLayout ya que nos permite alinear los componentes en vertical y horizontal en una rejilla de celdas de tamaño variable para cada elemento. Antes cada componente ocupaba una y sólo una celda, ahora el componente podra ocupar varias celdas y definirse así un tamaño más particular.

Cada componente se asocia a un objeto GridBagConstraints que se encarga de definir todos los parámetros de tamaño, posición, alineamiento y demás del componente. Para controlar estos parámetros daremos el valor de interés a las siguientes variables del objeto constraints:

**1. gridx:** Posición en x en la malla de celdas de la esquina superior de componente.

**2. gridy:** Posición en y en la malla de celdas de la esquina superior de componente.

El valor GridBagConstraints.RELATIVE (-1), que es el valor por defecto, especifica que el componente se ponga justo a la derecha (para gridx) o debajo (para gridy) del componente anterior.

**3. gridwidth:** Anchura del componente en número de celdas. Por defecto vale 1.

**4. gridheight:** Altura del componente en número de celdas. Por defecto vale 1.

Para gridwidth y gridheight los valores constantes siguientes significan:

GridBagConstranits.REMAINDER (0) especifica que este componente es el último de la fila (para gridwidth) o columna (para gridheight).

GridBagConstranits.RELATIVE (-1) especifica que este componente es el penúltimo de su fila (para gridwidth) o columna (para gridheight).

**5. fill:** Sirve para adecuar el tamaño del componente a su área de representación. Los valores de esta variable son:

GridBagConstranits.NONE (0) hace que el componente no se deforme. Es el valor por defecto.

GridBagConstranits.HORIZONTAL (2) ensancha el componente en horizontal para completar el área de representación en anchura. No modifica la alyura.

GridBagConstranits.VERTICAL (3) ensancha el componente en altura para completar el área que ocupa el componente en altura.No modifica la anchura del mismo.

GridBagConstranits.BOTH ensancha en altura y en anchura.

**6. ipadx:** Amplía la anchura del componente resultando la anchura anterior más 2\*ipadx pixeles.

**7. ipady:** Amplía la altura del componente resultando la altura anterior más 2\*ipady pixeles.

Estos parámetros modifican el tamaño sin deformar en sí el componente.

**8. insets:** Valor del margen del componente respecto de los ejes (imaginarios) de la malla.

**9. anchor:** Sirve para situar el componente dentro de su área de representación cuando está es mayor a lo que ocupa el componente. Los valores posibles son:

GridBagConstranits.CENTER (10)

GridBagConstranits.NORTH (11)

GridBagConstranits.NORTHEAST (12)

GridBagConstranits.EAST (13)

GridBagConstranits.SOUTHEAST (14)

GridBagConstranits.SOUTH (15)

GridBagConstranits.SOUTHWEST (16)

GridBagConstranits.WEST (17)

GridBagConstranits.NORTHWEST (18)

**10. weightx**: Número de componentes que caben en una fila.

**11. weighty:** Número de componentes que caben en una columna.

Por defecto, estas variables toman el valor 0 y el sistema con ello añade espacio extra entra la malla de celdas y los ejes del contenedor.

Los valores numéricos entre paréntesis son los valores que tienen dichas constantes que también podemos utilizar para no tener que teclear el nombre completo aunque perderemos claridad en el código fuente;

Como ejemplo de uso del GridBagLayout podemos propone el siguiente ejemplo:

Hemos definido primero un método llamado addComp(), que crea el objeto GridBagConstraints y asigna valores a las variables características antes comentadas para luego pasar a añadir el componente al GridBagLayout que le indiquemos. El hacerlo de este modo resulta práctico cuando vayamos a poner bastantes componentes en el layout y queramos características particulares para cada uno de ellos.

EJEMPLO:

void addComp(Panel panel, GridBagLayout gb, Component comp, int x, int y, int w, int h, int fill, int anchor) {

GridBagConstraints gbc = new GridBagConstraints();

gbc.gridx = x;

gbc.gridy = y;

gbc.gridwidth = w;

gbc.gridheight = h;

gbc.fill = fill;

gbc.anchor = anchor;

gbc.weightx = 1.0;

gbc.weighty = 1.0;

gb.setConstraints(comp,gbc);

panel.add(comp);

}

FlowLayout g = new FlowLayout();

setLayout(g);

GridBagLayout gb1 = new GridBagLayout();

Panel panel1 = new Panel();

panel1.setLayout(gb1);

TextField nombreTF = new TextField("Antonio Cabanyero Campos",45);

TextField telefonoTF = new TextField("964 217135",12);

addComp(panel1,gb1,new Label("Nombre:"),0,0,2,1,0,17);

addComp(panel1,gb1,nombreTF,2,0,4,1,0,13);

addComp(panel1,gb1,new Label("Telefono:"),0,1,2,1,0,17);

addComp(panel1,gb1,telefonoTF,2,1,2,1,0,13);

add(panel1);

Aquí hemos definido un FlowLayout como panel principal al que le iremos añadiendo paneles que podrán tener distintos layouts. En este caso sólo le añadimos un panel (panel1) que hemos definido como un GridBagLayout que contiene cuatro elementos: dos etiquetas y dos campos de texto.

### BorderLayouts.

Cuando añadimos un nuevo componente al panel que usa un layout de borde nosotros indicamos su emplazamiento geográfico como North, South, West, East o Center( la primera letra como se puede observar debe de ser mayúscula).

Los elementos del borde ocupan el espacio que necesiten y el central ocupa el espacio restante. Para crear un border layout hemos de teclear:

**setLayout(new BorderLayout());**

, después añadimos los elementos con un método add() algo especial pues tiene dos argumentos: un string indicando la posición y un campo de texto:

**add("North", new TextField("Titulo",50));**

**add("South", new TextField("Estado",50));**

También podemos crear un border layout con separación entre elementos:

**setLayout(new BorderLayout(10,10));**

EJEMPLO :

import java.awt.\*;

import java.applet.Applet;

public class Border extends Applet

{

Button b1=new Button ("Norte");

Button b2=new Button ("Sur");

Button b3=new Button ("Este");

Button b4=new Button ("Oeste");

Button b5=new Button ("Cento");

public void init()

{

setLayout(new BorderLayout());

add("North",b1);

add("South",b2);

add("East",b3);

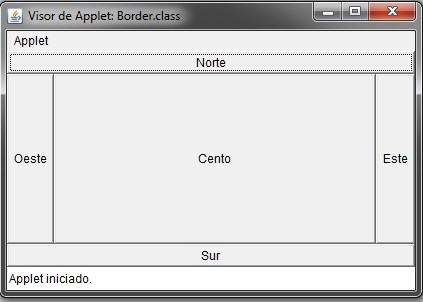
add("West",b4);

add("Center",b5);

}

}

Salida:



Imgm2\_19

### CardLayouts.

Si tratamos con Card layouts, los componentes del panel no son representados a la vez en el panel. Se representan uno cada vez dando la sensación de tener una diapositiva.

Normalmente cuando creamos un Card layout sus componentes suelen ser otros contenedores de elementos, normalmente paneles. Y podemos usar diferentes layouts para cada "card" individual. Cuando añadamos un nuevo "card" al panel podemos darle un nombre. Usando los métodos de la clase CardLayout podemos mover hacia atrás y hacia delante en la pantalla los diferentes "cards".

Un ejemplo de creación de un Card layout es el siguiente:

setLayout(new CardLayout());

Panel uno = new Panel();

add("Primero",uno);

Panel dos = new Panel();

add("Segundo",dos);

Panel tres = new Panel();

add("Tercero",tres);

show(this,"Segundo");

### Insets.

Los insets se utilizan para fijar los márgenes en horizontal y vertical entre los límites del panel y los elementos. La clase **insets** nos proporciona valores para los insets: top, bottom, left y right (superior, inferior, izquierda y derecha).

Para incluir un inset en cualquier layout sobreescribiremos el método insets() en nuestra clase:

**public Insets insets() {**

**return new Insets(vsup, vinf, vizq, vder);**

**}**

Donde vsup, vinf, vizq y vder son los enteros que queramos.

### Más componentes GUI.

Veamos algunos componentes UI más como: áreas de texto, listas scroll y lienzos.

### Areas de texto.

Son similares a los campos de texto pero tienen más funcionalidad para manejar mayores cantidades de texto. Las áreas de texto tienen scrollbars por defecto. Veamos cómo crear un área de texto:

**TextArea()** crea un área vacía de 0 filas y 0 columnas de caracteres de anchura.

**TextArea(int, int)** crea un área de texto vacía pero con el numero de filas y columnas dado.

**TextArea(String)** crea un área de texto conteniendo el string pero de 0 filas y 0 columnas.

**TextArea(String, int, int)** crea un área de texto con el string dado y de tamaño el numero de filas y columnas pasado.

Los componentes áreas de texto tienen métodos predefinidos:

|  |  |
| --- | --- |
| Método | **Acción** |
| getColumns() | devuelve el numero de columnas. |
| getRows() | devuelve el numero de filas. |
| insertText(String,int) | pone el texto que pasamos dentro del campo y en la posición que le pasamos, comienza por la posición 0. |
| replaceText(String,int) | cambia el texto que pasamos dentro del campo entre las posiciones que le pasamos. |
| setLineIncrement(int inc) | cambia el incremento de como de largo es el scroll al pulsar las flechas de estos. El valor por defecto es 1. |
| getLineIncrement() | devuelve el valor del incremento del scroll. |
| setPageImcrement(int inc) | fija el incremento de rango que por defecto es 10. |
| setPageImcrement() | devuelve el incremento del scroll de rango. |

### Listas scroll.

Es como un menú de selección pero no es desplegable, podemos definirla para elegir una o varias opciones, la lista puede ser larga y tener scrollbar.

Para crearlas:

**List()** crea una lista scroll vacía que permite solo una selección.

**List(int,boolean)** crea una lista con el número de líneas visibles que le pasamos. El valor boolean indica a true que es no-exclusiva o a false que es exclusiva.

Después de crear la lista scroll le añadiremos ítems mediante el método additem(). Como ejemplo puede valer el siguiente código:

List lst = new List(5,true);

lst.addItem("Asturias");

lst.addItem("Castilla ");

lst.addItem("Baleares");

lst.addItem("Extremadura");

lst.addItem("Centro");

Las listas scroll tienen métodos predefinidos para trabajar con ellas:

|  |  |
| --- | --- |
| **Método** | **Acción** |
| getItem(int) | devuelve el string del ítem de la posición que le pasamos. |
| countItems() | devuelve el numero de ítems del la lista. |
| getSelectedIndex() | devuelve la posición del ítem seleccionado (para listas exclusivas). |
| getSelectedIndexes() | devuelve las posiciones de los ítems que están seleccionados (para listas no-exclusivas). |
| getSelectedItem() | devuelve el ítem seleccionado como String (para listas exclusivas). |
| getSelectedItems() | devuelve los ítems seleccionados como array de Strings. (para listas no- exclusivas). |
| select(int) | selecciona el ítem dada la posición. |
| select(string) | selecciona el ítem con el string dado. |

### Scrollbars y deslizadores.

Podemos también crear scrollbars y sliders individuales para manipular rangos de valores o para seleccionar un valor dentro de un rango predeterminado. Para cambiar dichos valores podemos utilizar tres facilidades:

a. Flechas de incremento/decremento.

b. Un rango en el medio.

c. Una caja en el medio para incrementar/decrementar.

Para este componente no hemos de manejar ningún evento, solo fija los valores máximo y mínimo. Para crear una scrollbar:

**Scrollbar()** crea una scrollbar de tamaño 0 (máximo y mínimo valor).

**Scrollbar(int)** scrollbar de tamaño 0 (máximo y mínimo valor). El valor int es para la orientación, int= Scrollbar.HORIZONTAL o Scrollbar.VERTICAL.

**Scrollbar(int, int, int, int, int)** crea la scrollbar y cada int indica y en este orden:

Orientación: Scrollbar.HORIZONTAL o Scrollbar.VERTICAL   
Valor inicial (debe estar entre el máximo y el mínimo)   
Anchura del rango total mostrada   
Mínimo valor   
Máximo valor

EJEMPLO:

import java.awt.\*;

public class Scroll extends java.applet.Applet {

Label l;

public void init() {

setBackground (Color.orange);

l = new Label("0");

add(l);

add(new Scrollbar(Scrollbar.HORIZONTAL,1,0,1,100));

}

public boolean handleEvent(Event evt) {

if (evt.target instanceof Scrollbar) {

int v = ((Scrollbar)evt.target).getValue();

l.setText(String.valueOf(v));

}

return true;

}

}

Salida:



Imgm2\_20

Los métodos ya definidos para las scrollbar son:

|  |  |
| --- | --- |
| **Método** | **Acción** |
| getMaximum() | devuelve el valor máximo. |
| getMinimum() | devuelve el valor mínimo. |
| getOrientation() | devuelve la orientación: o si es horizontal y 1 si es vertical. |
| getValue() | devuelve el valor actual de la scrollbar. |
| setValue(int) | fija el valor actual de la scrollbar. |

**Módulo 3. POO. Conceptos básicos II**

* 1. **Instancias y métodos constructores**

Los objetos son modelos de programación que abarcaran una serie de propiedades por medio de variables y una funcionalidad por medio de métodos en Java, y funciones en C++ por ejemplo.

Muchas de las características de Java en estos aspectos son herencias de C++, lenguaje del cual deriva casi directamente.

Por otro lado, las clases son la generalización de los objetos, de forma que un conjunto de objetos con las mismas características y funcionalidad formarán una clase.

Surge ahora el termino de **instancia**. Una instancia de una clase no es más que un objeto particular de esa clase. Cuando creamos un objeto de una determinada clase diremos que ese objeto se ha instanciado de la clase a la cual pertenece:

**Clase Objeto**

Las partes de una clase, como generalización de los objetos serán:

1. **Atributos o propiedades**. Características; son las variables.

2. **Métodos**(Comportamiento). Funcionalidad, son las funciones, que ahora se llaman métodos y que sirven para operar con las variables de la clase.

Podemos recordar el termino de herencia para mencionar la herencia de clases, que se refiere a que se tiene una clase Padre y una clase Hija que deriva de esta y que adquiere (hereda) todas las características (variables) y funciones (métodos) de la clase padre .

Java tiene ya un conjunto de clases predefinidas. Son las **clases primitivas**. Para incluir en nuestros programas **paquetes** (este conjunto de clases ya predefinidas en java y que son necesarias muchas veces , pues aportan una determinada funcionalidad a nuestro programa) con otras clases definidas por nosotros, teclearemos el comando **import** seguido del nombre del paquete que nos interese cargar y después el nombre de la clase perteneciente a ese paquete.

Ejemplo:

import java.appletApplet.;

importamos la clase Applet del paquete java.applet

import java.awt.Color;

importamos la clase Color del paquete java.awt

Si nos interesara incluir todas las clases del paquete java.awt , teclearemos el comando import seguido del nombre del paquete que nos interese cargar y después el símbolo \* (asterisco) para hacer referencia a todas las clases incluidas en ese paquete..

import java.awt.\*;

importamos todas las clases del paquete java.awt

Vamos a pasar a ver ahora todos los parámetros que pueden aparecer a la hora de definir una clase..

Definición de una clase:

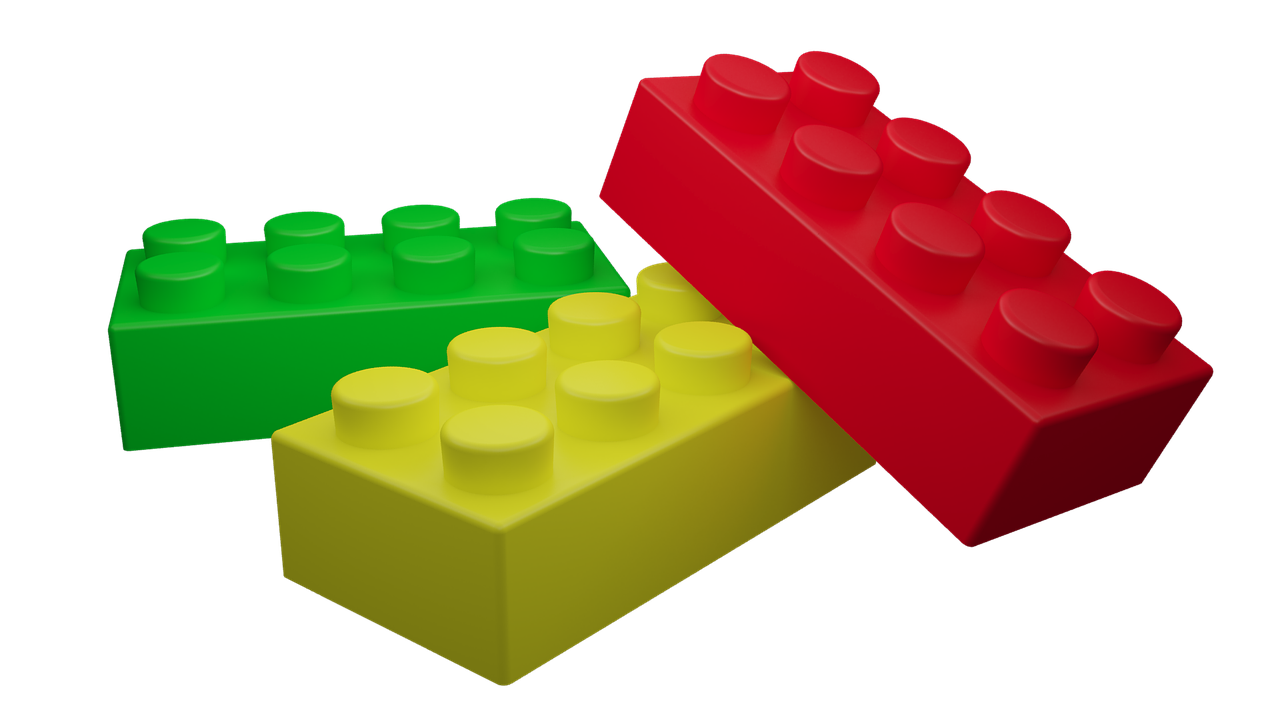
**[Modificador] class Nombre\_clase [extends Nombre\_clase] [implements Nombre\_Interface]   
{**

**}**

Todo lo que va entre corchetes es opcional.

* [Modificador] Aplicable a clases , se veran más adelante los posibles modificadores.
* class Palabra reservada .Sirve para declarar una clase .Todos los atributos y métodos de la clase se definirán dentro de las llaves de abajo.
* Nombre de Clase Es el nombre que le damos a la clase .
* [extends Nombre\_clase] Indica que la clase hereda de otra clase .También se indica el nombre de la clase de la que se hereda.
* [implements Nombre\_interface] Indica que se implementa un interface java . Posteriormente se definirá más a fondo el concepto de interface java.

# Constructores



Imgm3\_1

Es un método especial que se ejecuta justo después de haber creado el objeto y que sirve para inicializar el objeto. No reserva memoria para el objeto sino que lo inicializa. Quién reserva memoria para el objeto es el operador new.

El constructor se caracteriza porque tiene el mismo nombre que la clase y no tiene retorno, ni siquiera se le pone el modificador void:

Class punto {

int x, y;

punto (int x, int y) {

**CONSTRUCTOR**

this.x = x;

this.y = y;

}

}

punto p1 = new punto (3,1);

punto p2 = new punto (5,4);

El constructor puede estar sobrecargado:

class fecha {

int dia, mes, anno;

fecha () { // Constructor por defecto.

this.dia = 14;

this.mes = 10;

fecha f1 = new fecha()

this.anno = 2000;

}

fecha (int dia, int mes, int anno) {

this.dia = dia;

this.mes = mes;

fecha f2 = new fecha(1, 1, 2000)

this.anno = anno;

}

fecha (int dia, int mes) {

this.dia = dia;

fecha f1 = new fecha(1, 2)

this.mes = mes;

this.anno = 2000;

Llamamos **constructor por defecto**, al constructor que no recibe parámetros.

Llamamos **constructores generales** a los que reciben uno o más parámetros.

Si una clase no tiene constructores, el compilador le pone un **constructor por defecto anónimo**.Con lo que no es obligatorio definir un constructor dentro de una clase.

class punto {

int x, y;

punto (int x, int y) {

this.x = x;

this.y = y;

}

punto (int xy) {

this (xy, xy); // Llama al constructor anterior.

}

}

# Métodos

Es una función asociada a un objeto, de forma que siempre que se va a ejecutar un método se tiene que decir sobre qué datos del objeto. El método no tiene el modificador static delante del nombre.

class fecha {

int dia, mes, anno;

void imprimir () {

System.out.println (this.dia+”/”+this.mes+”/”+

this.anno);

}

}

Mientras las funciones no necesitan un objeto para ejecutarlas, los métodos necesitan que se haya creado el objeto para ejecutar el método:

fecha f1 = new fecha();

fecha f2 = new fecha();

f1.dia = 20; f1.mes = 9; f1.anno = 2000;

f2.dia = 6; f2.mes = 11; f2.anno = 1999;

f1.imprimir(); // Llamada al método 🡪 20/9/2000

f2.imprimir(); // Llamada al método 🡪 6/11/2000

Las operaciones que se pueden realizar sobre objetos se van realizar por medio de métodos.

# Sobrecarga de funciones

En Java es posible crear varias funciones con el mismo nombre de forma que se van a distinguir por sus parámetros. La diferencia de los parámetros se pueden establecer por:

* respecto al número de parámetros.
* respecto al tipo de los parámetros.

Cuando se llame a la función el compilador decidirá a cual de las funciones sobrecargadas llamar en función a los parámetros.

1. static int suma (int a, int b) { 🡪 suma\_int\_int

return (a + b);

}

1. static int suma (int a,int b,int c) { 🡪 suma\_int\_int\_int

return (a + b + c);

}

3 static double suma (double a, double b) { 🡪 suma\_double\_double

return (a + b);

}

int a = 3, b = 5, c = 8;

float f = 3.7f;

double d = 8.2, e = 1.7;

suma (a,b); // Llama 1.

suma (a, b, c); // Llama 2.

suma (d, e); // Llama 3.

suma (a, d); // Llama 3.

suma (a, b, f); // Error no se puede estrechar.

Suma (a, b, (int) f); // Llama 2.

La sobrecarga de funciones solamente se puede hacer respecto a los parámetros pero no sobre el retorno:

double x = suma (a, b); // Llama 1.

int y = suma (d, e); // Llama 2 y se produce error.

int y = (int) suma (d, e); // Llama 2.

* 1. **Interfaces**

El concepto de interface en java es importante. Vamos a introducir previamente una serie de conceptos previos y que no están de más.

#### Clases Abstractas

Una clase abstracta (modificador abstract) es una clase que puede tener clases que deriven de ella, pero no puede ser instanciada , en definitiva no se puede hacer un new .

Además pueden tener al menos un método abstract también, es decir un método que obligatoriamente hay que redefinirlo ( sobrescribirlo).

Las clases abstractas sirven para modelar conceptos. Por ejemplo, la clase Number es una clase abstracta que representa cualquier tipo de números y sus métodos no

están implementados , son abstractos , lo que quiere decir además que están vacíos.

Las clases descendientes de ésta, como Integer o Float, si implementan los métodos de la clase base Number, y se pueden instanciar

Es una clase diseñada para actuar como bases de otras clases derivadas en la cual se pone la funcionalidad (los métodos) común de los objetos derivados pero a diferencia de las clases normales no se puede crear objetos de la base abstracta.

Ejemplo:

abstractclass Figura {

// Cuerpo de la clase.

}

No se pueden crear objetos pero sí referencias:

Figura fig; // Referencia.

Figura fig = new Figura(); //Error, no se puede crear objeto.

Figura fig = new Alfil(); //Correcto, Alfil es derivada.

Como ya dijimos anteriormente las clases abstractas opcionalmente pueden tener métodos abstractos. Un método abstracto que no se implementa ( va sin código) para que se una clase derivada quien lo implemente.

Ejemplo:

abstract class Figura extends Casilla {

int color;

Figura (int x, int y, int color) {

super (x, y);

this.color = color;

}

public String toString() {

return “Color: “+color+super.toString();

}

abstract bolean movimientoACasilla (Casilla c);

**// Método no implementado ya que es abstracto, se**

**// redefine y se ejecuta el método en la clase derivada.**

boolean esAtacadaPor (Figura f) {

if (f.puedeMoverA(this)) return true;

return false;

}

}

Sólo las clases abstractas pueden tener métodos abstractos.

Una clase abstracta tiene que derivar de una clase abstracta.

Si una clase tiene un método abstracto sus derivadas deben implementar el método abstracto o seguirán siendo clases abstractas.

Una clase abstracta puede tener constructores serán obligatorios para las derivadas.

Los métodos estáticos no pueden ser abstractos.

# Clases Anidadas

Llamamos clase interna (inner class) a una clase que está declarada dentro de otra a la que se le denomina clase externa (outer class).

La clase interna solamente puede ser instanciada dentro de la clase externa. En este caso, solo la clase Robot puede acceder a la clase Descripción.

La clase interna va tener la ventaja de que puede acceder a los atributos de la clase externa aunque sean privados y a sus propios atributos.

La clase interna puede ser estática en cuyo caso sólo puede acceder a los atributos estáticos de la externa pero a cambio va a poder ser instanciada desde fuera de la clase:

class Contenedora {

static class contenida {

int x,y;

}

}

public static void main (String[] args) {

Contenedora.Contenida c1 = new Contenedora.Contenida();

Las clases anidadas pueden ser también abstractas y finales (abstract / final).

# Clases Locales

Se denomina clase local a una clase que está declarada dentro del método de otra clase con lo que la clase local va a poder acceder a las variables locales del método y a los atributos de la clase externa.

La clase local va a poder derivar de la clase externa pero no al revés.

Ejemplo:

class Prueba {

public static void main(String [] args) {

**class Punto {** // Clase Local puede acceder a las var.

**int x,y;** // locales de main y a los atributos

**}** // de la clase Prueba.

}

Punto p = new Punto();

}

# Clases Anónimas

Se denomina clase anónima a una clase que se crea sin nombre como derivada de otra y que a la vez que se crea la clase anónima se instancia un objeto de dicha clase.

Ejemplo:

public static void main (String [] args) {

Punto p = new Punto(2,3)

{

int color = 5;

public String toString() {

return super.toString()+” color: “+color;

}

}**;**

}

En este caso no se puede acceder a los atributos de la clase anónima:

p.color = 10; //Error.

Pixel px = (Pixel) p; // Error.

Las clases anónimas sirven para redefinir métodos.

imgm3\_2

**Una interface** es una clase que no implementa sus métodos sino que deja a cargo la implementación a otras clases. Las interfaces pueden derivar de otras interfaces pero no de otras clases.

Sus métodos suelen ser abstractos y sus atributos son finales (una interfaz contiene solamente métodos vacios y/o constantes).

Una utilidad es generar relaciones entre clases que de otro modo no están relacionadas (haciendo que implementen los mismos métodos), o para distribuir paquetes de clases indicando la estructura de la interface pero no las clases individuales.

Distintas clases pueden implementar las mismas interfaces, y a la vez derivar de otras clases. Una clase solo puede derivar a la vez de una sola clase , sin embargo puede implementar más de una interface .Es la forma de dotar a java d herencia múltiple ya que a diferencia de C++ carece de esta característica.

class <nombre\_clase> [extends <clase base>]

[implements <lista\_interfaces>]

{. . .}

Un interface es una especificación de las operaciones que tienen los objetos de un determinado tipo de forma que un fabricante define un interface y otros muchos cumplen con ese interface.

Ejemplo:

**interface** Television {

[abstract] void encender();

void apagar();

void subirVolumen (int cuanto);

void bajarVolumen ( int cuanto);

void cambiarCanal (int canal);

int getCanalActual();

}

class Sony **implements** Television {

int volumen, canal;

static final int MIN\_VOLUMEN = 0;

static final int MAX\_VOLUMEN = 20;

static final int MIN\_CANAL = 1;

static final int MAX\_CANAL = 99;

void apagar () {

System.out.println (“Televisor apagado. Adiós.”);

}

void encender () {

System.out.println (“Televisión Sony BT13, hola.”);

}

void subirVolumen (int cuanto) {

if (volumen + cuanto > MAX\_VOLUMEN)

volumen = MAX\_VOLUMEN;

else volumen += cuanto;

}

void bajarVolumen (int cuanto) {

if (volumen – cuanto > MIN\_VOLUMEN)

volumen = MIN\_VOLUMEN;

else volumen -= cuanto;

}

void cambiarCanal (int canal) {

if (canal < MIN\_CANAL || canal > MAX\_CANAl)

return;

this.canal = canal;

}

void getCanalActual() {

return canal;

}

} // Fin de clase Sony.

Una clase que implementa un interface, tiene que implementar todos sus métodos o bien declararse como abstracta.

* 1. **Polimorfismo**

Se refiere a la posibilidad de poder tener varios métodos dentro de una clase que se llamen igual pero que reciban distinto tipo o distinto número de parámetros:

Ejemplo:

suma (int x , int y)

suma (int x , int y , int z)

suma (int x , float y)

Este concepto será explicado más adelante una vez que se expliquen todos los tipos de datos java.

* 1. **Herencia**

La herencia se aplica sobre las clases. Indica que las clases puedan tener descendencia a la que cederá propiedades y métodos. Se conseguirá así una estructura jerarquizada.

Color

Motor

\_\_\_\_\_\_\_\_\_\_\_

Acelerar()

Frenar()

**Clase Coche**

Color =”Rojo”

Motor =”Diesel”

Puertas =5

ABS =”si”

Acelerar()

Frenar()

Girar()

Puertas

ABS

\_\_\_\_\_\_\_\_\_\_\_\_\_

Girar()

**Clase Seat Objeto Seat Cordoba**

Obsérvese que el objeto Seat Cordoba por pertenecer a la clase Seat toma todas sus propiedades y métodos , pero a su vez como la clase Seat *hereda* de la clase Coche , el objeto también tiene todas las propiedades y métodos de esta clase. Así mismo como se ve , sigue siendo en el objeto y no en las clases donde las propiedades toman valor.

# Agregación y Composición

**Agregación** , crear un objeto desde una clase que se denomina objeto contenedor que va a tener referencias a objetos de otra clase a los que llamamos objetos contenidos. La vida de los objetos contenidos no está limitada a la vida del contenedor.

**Composición** , consiste en que hay un objeto que llamamos contenedor que está formado por objetos de otra clase a los que llamamos contenidos. De forma que la vida de los contenidos está limitada a la del contenedor.

# Conversión Implícita y Explícita con Referencias a Objetos

Punto p = new Punto();

Fecha f = new Fecha();

p = f; // Error, son clases diferentes.

p = (Punto) f; // Error, con casting tampoco se puede hacer.

Si las clases están relacionadas mediante relaciones de herencia, si se pueden hacer conversiones.

Java permite convertir una referencia a clase derivada en una referencia a clase base implícitamente, y permite convertir una referencia clase base en una referencia clase derivada utilizando casting :

Animal a;

Caballo

Perro

Animal

a = new Perro(); //Correcto

Perro p;

p = new Animal(); // Error

Punto

Punto p = new Punto();

Pixel

Punto px = new Pixel();

p = px; // Correcto, conversión implícita.

px = p; // Error, se necesita casting.

px = (Pixel) p; // Correcto.

px.x = 3; px.y = 5; px.color = 1;

p.x = 3; p.y = 5; // No puede acceder a color.

Punto p = new Punto();

Pixel px = new Pixel();

p = px; // Se convierte la referencia y no el objeto.

. . . .

// Se pierde px.

. . . .

// Se quiere acceder a color.

p.color = 3; // Error, p solo puede acceder a x e y.

Pixel px2 = (Pixel) p;

px2.color = 3; // Correcto utilizando casting.

Punto p = new Punto (1,3);

Pixel px = new Pixel(2, 4, 1);

px = (Pixel) p; // Compila con casting.

px.color = 5; // Se le engaña en la compilación pero en tiempo

// de ejecución produce una excepción.

px

1

2

1

4

x x Punto

3

y y

color 5

Entre clases hermanas no se pueden realizar conversiones ni con el operador casting.

Deriv1 d1 = new Deriv1();

Deriv2

Deriv1

Base

Deriv2 d2 = new Deriv2();

d1 = d2; // Error.

d1 = (Deriv1) d2; // Error.

Es lógico que no se pueda hacer la conversión ya que existirán atributos y métodos en una clase y en otra no, y viceversa.

En Java no se permiten las conversiones entre las referencias y los tipos fundamentales de datos ni con casting:

int a = 3;

Punto p = (Punto) a; // Error.

En Java no exicte la herencia múltiple , esto significa que una clase solo puede heredar de una sola clase.En cambio en lenguajes como C++ sí que existe , significando esto que una clase podría heredar de más de una clase(puede tener más de un padre).

# Enlace Estático y Enlace Dinámico

class Base {

**final** void Metodo1() {

System.out.println (“Método de la Clase Base”);

}

}class Derivada extends Base {

void Metodo1() {

System.out.println (“Método de la Clase Derivada”);

}

}

Base b = new Derivada(); // Correcto;

b.Metodo1(); // ¿Cuál se ejecuta?

**Enlace estático** , consiste en decidir que método ejecutar en función del tipo de la referencia con la que se está apuntando al objeto. ( Se ejecuta Metodo1 de la clase Base).

**Enlace dinámico** , consiste en decidir que método se debe ejecutar en función del tipo de objeto que se está apuntando desde la referencia. (Se ejecuta Metodo1 de la clase Derivada)

La ventaja del enlace Estático es que es más rápido que el enlace Dinámico porque en tiempo de compilación ya se sabe qué método ejecutar.

Tiene el inconveniente de que normalmente se va a querer ejecutar el método del objeto apuntado y no el de la referencia que se usa para apuntar.

En el enlace Dinámico es más lento porque hasta que no llega el momento de ejecutar el método no sabe a que objeto estará apuntando la referencia.

Java utiliza por defecto el enlace dinámico y si se quiere que utilice el enlace estático se tiene que poner el modificador **final** al método.

# El operador instanceof

Sirve para preguntar si un objeto al que se está apuntando con una referencia es de una determinada clase. El formato es el siguiente:

**<Referencia> instanceof <clase>**

Ejemplo:

Pixel px = new Pixel();

if (px instanceof Pixel) {

// px es Pixel. Se cumple.

}

if (px instanceof Fecha) {

// px no es Fecha. No entra.

}

if (px instanceof Punto) {

// px no es Punto, pero Pixel deriva de Punto. Se cumple.

}

if (px instanceof Object) {

// px al derivar de Punto, Punto deriva de Object.

// Se cumple.

}

# El método finalize()

Es un método que se ejecuta justo antes de que el sistema de recogida de basura de Java libere la memoria del objeto. El método **finalize()** se puede redefinir para poner las operaciones que se quieran hacer antes de destruir el objeto:

class Punto p = new Punto (2,3);

p.finalize() // Ejecutaría las operaciones de desinicialización

// pero no libera memoria.

class Punto {

int x, y;

protected void finalize() throws Throwable {

//Operaciones de desinicialización.

El método **finalize()** sólo es ejecutado por JVM.

El método **finalize()** se encuentra en la clase Object, vacío. En las clases derivadas es donde se redefine.

super.finalize();

EJEMPLO:

class Pixel {

int x, y, color;

protected void finalize() throws Throwable {

super.finalize(); // Punto 🡪 Object.

}

}

* 1. **Encapsulación. Niveles de acceso**

Se refiere a los diferentes tipos de acceso o por llamarlo de otra forma el nivel de accesibilidad a las propiedades y métodos de un objeto.



Imgm3\_3

Habrá 4 niveles de accesibilidad donde la restricción irá aumentando paulatinamente. Aunque veremos este concepto más desarrollado diremos que los niveles son 4:

De menos a más restricción:

* public
* friendly
* protected
* private

El polimorfismo permite que objetos distintos puedan tener métodos con el mismo nombre o incluso un mismo objeto puede tener nombres de métodos idénticos pero con distintos parámetros) es prácticamente el concepto de sobrecarga de métodos.

En java existen multitud de métodos polimórficos , es decir varias versiones del mismo método

Que las usaremos según nos convenga en un programa , la única condición repetimos es que no reciban igual número o igual tipo de parámetros.

EJEMPLO:

class Prueba{

public void pintar( int x , int y){

System.outr.println( x+y);

}

public void pintar( String a , String b){

System.outr.println( a+b);

}

public void pintar( int x , int y,String z){

System.outr.println( (x+y) + “<>” + z);

}

}

Tiene 3 métodos con el mismo nombre pero parámetros distintos.

Lo que no podremos tener nunca serán 2 métodos pintar que reciban el mismo número y tipo de parámetros , la implementación de este método dentro de la clase Prueba daría un error:

public void pintar( int c , int m){

System.outr.println( x+y);

}

# El modificador final.Constantes

Es lo que en programación entendemos por constante, un valor que no varía durante la ejecución del programa.

En los atributos indica que el atributo es una constante, su valor no se puede modificar:

class fecha {

**final** int diasSemana = 7;

**final** int [] diasMes = {31,28,31,30,31,30,31,31,30,31,30,31};

int dia, mes, anno;

}

fecha f1 = new fecha()

f1.diasSemana = 5**; // Error**.

f1.diasMes[1] = 29**; // Correcto, los datos del array se**

**// pueden modificar.**

f1.diasMes = new int[12]; **// Error, no se puede crear un nuevo**

**// array, lo constante es la referencia**.

**-o**, opción de compilación para no incluir las constantes en los objetos. De esta forma las constantes se sustituyen por su valor y no reserva espacio en memoria.

Normalmente los atributos declarados como final van a ser estáticos:

static final float pi = 3.1416f;

static final int altas = 2;

La razón es que no se necesita crear un objeto para acceder a ellas:

En los métodos, el modificador final indica que no se pueden redefinir, es decir, que ninguna clase derivada puede cambiar ese método.

**El modificador protected**

Los atributos de una clase o métodos también de una clase que sean declarados como protected

solamente podrán ser accedidos desde la propia clase y desde las clases derivadas de ella , pero nunca desde otras clases.

EJEMPLO:

class A

**protected int num;**

class C

class B

String cadena

El atributo num al ser declarado como protected ,solamente podrá ser accedido desde objetos de las clases A ( su propia clase) y desde la clase B ya que es derivada de A , pero nunca desde la clase C

**El modificador private**

Los atributos de una clase o métodos también de una clase que sean declarados como private

solamente podrán ser accedidos desde la propia clase

EJEMPLO:

class A

**private int num;**

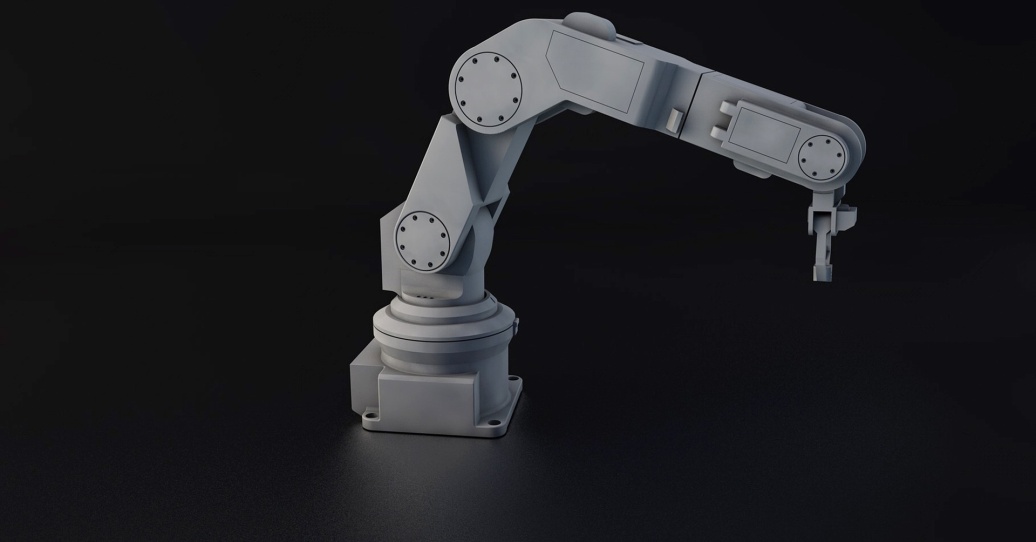
class C

class B

String cadena

El atributo num al ser declarado como private ,solamente podrá ser accedido desde objetos de la clase A , pero nunca desde la clases B y C , a pesar de que B sea derivada de A.

* 1. **Modificador de acceso static**

****

Imgm3\_4

En java este modificador que se aplica a variables y a métodos fundamentalmente lo que significa es que no necesitamos crear un objeto para poder referenciar un método o una propiedad de una clase.

Recordemos que siempre que queremos acceder a una propiedad o método ( no estáticos) de una clase necesitamos obligatoriamente crear un objeto de la misma .

Por ejemplo si tenemos la clase Pais de la siguiente manera:

class Pais {

String nombre;

Long población;

}

Para poder acceder a estas variables necesitaríamos un objeto de la clase Pais:

Pais p=new Pais();

p.nombre=”España”;

Sinembargo en caso de que la variable nombre fuese estática bastaría con anteponer el nombre de la clase para acceder a la variable *nombre:*

class Pais {

**static** String nombre;

Long población;

}

Y para acceder a ella:

Pais.nombre=”España”;

Como vemos no es necesario el objeto p.

Lo mismo sucede con los métodos, aquellos que se declaren con el modificador *static* delante podremos acceder a ellos sin necesidad de crear un objeto.

**3.7 Ejecución de constructores en la herencia. Uso de *super* y *this.***

# Orden de Ejecución de los constructores en la herencia

La clase derivada hereda todos los métodos y atributos excepto el constructor de la clase base (ni constructor por defecto ni los constructores generales).

Cuando se crea un objeto de la clase derivada, primero se ejecuta el constructor por defecto de la clase base y posteriormente el constructor de la clase derivada.

class Punto { class Pixel {

int x, y; int color;

Punto() { Pixel (int x,int y,int col) {

this.x = -1; this.x = x;

this.y = -1; this.y = y;

} this.color = col;

}

Punto (int x, int y)

{ }

this.x = x;

this.y = y;

}

}

*Pixel px = new Pixel (2, 5, 1);*

*Primero se ejecuta el constructor por defecto de*

*clase base (Punto).*

Seguidamente se ejecuta el constructor general de la clase derivada (Pixel).

# El operador super

Se puede usar para dos funcionalidades:

* Indicar que constructor de la clase base se quiere que se ejecute cuando se crea un objeto de la clase derivada.
* Indicar que se quiere ejecutar un método de la clase base que existe también en la clase derivada.

EJEMPLO:

class Punto {

int x, y;

Punto (int x, int y) {

this.x = x; this.y = y;

}

Public toString() {

return “[“+x+” , “+y+”]”;

}

} // Cierre de la clase Punto

class Pixel {

int color;

Pixel (int color) {

this.color = color;

super (x, y); // Llamada a constructor de la

//clase Punto.

}

Public String toString() {

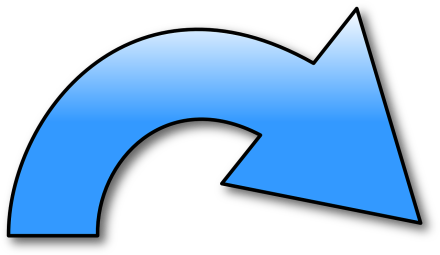
return super.toString() + “color: “ + color;

// Llamada a la función toString de la clase Punto.

}

}

**El operador this**

****

Imgm3\_5

**this**, es una referencia sobre el objeto que se está ejecutando en el momento actual .

Es un objeto que apunta a si mismo podría decirse ( a la propia clase).

Casi siempre se hace referencia al objeto actual y en algunos casos hace referencia a la clase actual. Como el significado de este parámetro depende en gran medida del contexto en el que lo estemos utilizando , haremos más hincapié en el this , cada vez que vaya surgiendo a lo largo del curso , explicando en cada momento a quién hace referencia.

Por ejemplo dentro de un constructor se puede referenciar a las variables de la clase por medio de este operador , como puede verse en la clase siguiente:

class Fecha{

int dia ;

int mes;

Fecha(int d){

this.dia=d; // **this.dia hace referencia a la variable dia de la clase Fecha**

}

Fecha(int d,int m){

this.dia=d; // **this.dia hace referencia a la variable dia de la clase Fecha**

this.mes=m; // **this.mes hace referencia a la variable mes de la clase Fecha**

}

}

**4. Módulo 4. Las actuales Tecnologías y sus Lenguajes de Programación**

* 1. **Plataforma JAVA-J2EE**

# Características de las aplicaciones de tres capas



# La arquitectura multicapa

Es una técnica muy utilizada en aplicaciones empresariales. Consiste en partir nuestro software en capas y básicamente hay dos arquitecturas:

**Arquitectura en 2 capas**

**Arquitectura en 3 capas**

Arquitectura en 2 capas

Consiste en dividir el programa en dos programas el programa cliente y el programa servidor.

Programa

Cliente

Programa Servidor

Programa

Cliente

Se comunican a través de una red. La capa cliente va a tener una interfaz gráfica para el usuario y la capa servidor tiene la Base de Datos de la empresa.

El problema surge en la siguiente pregunta:

¿Dónde poner la lógica de negocio (Bussiness Logic )?

**Logica de negocio:** Algoritmos que resuelven los problemas propios de la empresa.

Se puede poner en el cliente o en el servidor.

Ponerla en el cliente: Vamos a tener un cliente pesado y un servidor ligero.Tiene una ventaja y es que al **quitar carga al servidor** éste va a poder admitir más clientes. El inconveniente es que el **coste de distribución aumenta**. El coste de distribución el el coste de instalar los programas en el cliente , cada vez que cambie la lógica de negocio hay que cambiarla en los clientes. Otro inconveniente es que **aumenta el tráfico en la red** ya que el ResultSet hay que llevarlo hasta el cliente para que pueda trabajar con el.

Llevar parte de la lógica de negocio al servidor.Tendremos procedimientos almacenados pero se necesitan **Bases de Datos grandes**.La ventaja es que **disminuye el tráfico** por la red ( enviamos datos ya calculados).

# Arquitectura en 3 capas

La arquitectura de 3 capas o arquitectura multicapa es una arquitectura donde hay una separación física entre el cliente que solicita la información , los programas que la pocesan y los datos sobre los que operan.

La arquitectua de 3 capas separa la lógica de presentación de la lógica de negocio.

El código de negocio es almacenado y procesado sobre un servidor de aplicaciones en vez de en los clientes.

Esta separación supone que el código de negocio es independiente de cómo y donde se presenta.La capa de lógica de negocio no necesita saber que tipo de cliente muestra los datos.Las 3 capas son más portables , trabajan en diferentes tipos de plataformas y permiten un balance de las solicitudes del cliente entre varios servidores.

La seguridad es más fácil de implementar ya que el software de la aplicación está ahora fuera del cliente con lo que además el coste de distribución se reduce notablemente.

Vamos a tener 3 capas diferenciadas:

Capa de presentación o capa cliente

Capa de lógica de negocio

Capa de Datos

La **capa de presentación** está destinada a la recogida de datos , se instala en el cliente. Es la interfaz gráfica. Por ejemplo el browser , un cliente rico (aplicación java solitaria) , el network computer , etc.....

La **capa de negocio** contiene los algoritmos propios de la empresa así como los algoritmos que renderizan ( enviar HTML al cliente ) los datos ( los servlets o jsp’s)

Servidor

Cliente

v

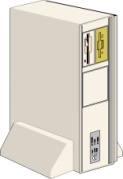
#### Capa de Datos

Capa de lógica de negocio

Capa de presentación

Está representada por un servidor web y uno o varios servidores de aplicaciones que ejecutan el código de negocio.

La **capa de datos** es la que utiliza la capa de lógica de negocio para almacenar y recuperar datos de forma persistente .Es la Base de Datos. Está representada por bases de datos relacionales u otras fuentes de datos finales como SAP R/3 por ejemplo.

Servidor web

Navegador web



Base de Datos empresarial



Servidor de Aplicaciones

Cliente Rico

**Capa de Datos**

#### Capa Servidor

#### Capa Cliente

Cuando un navegador web origina una solicitud , se envía al servidor web. Si la solicitud requiere una aplicación para procesa o acceder a los datos , el servidor web reenvía la petición al servidor de aplicaciones. El servidor de aplicaciones devuelve el resultado al servidor web , que a su vez reenvía la respuesta al cliente.

Cuando la solicitud es originada desde un cliente rico ( aplicación java solitaria ) se envía al servidor de aplicaciones mediante un enlace RMI-IIOP.

## Los monitores de transacciones

Fueron los primeros sistemas de 3 capas que se inventaron.

CICS (IBM)

Tuxedo ( Bea Logic )

Como capa de presentación usaban un terminal de pantalla verde.

En la capa de negocio tenían lenguajes de programación como COBOL , PL/1 , RPG y en la capa de datos una base de datos como DB2 de IBM por ejemplo.

Eran muy estables y aún se usan en **sistemas críticos** , sistemas que no se pueden permitir que fallen ( Bancos) .

Se les consideraba un sistema operativo porque controlaban todo el entorno en el que se ejecutaba la aplicación por ejemplo asignar el nivel de recursos de la aplicación , transacciones , etc...

Después con la aparición de los sistemas de objetos distribuidos se empezó a utilizar CORBA para implementar la lógica de negocio. Es lo mismo pero orientado a objetos.

CORBA es solo un sistema de comunicación entre objetos ( a través de ORB , peticiones a objetos remotos ) pero CORBA no tiene todos los servicios de un monitor de transacciones por lo que el programador tiene que implementar todos sus servicios a mano .De ahí que surjan los servidores de aplicaciones , estos proporcionan los servicio típicos de los monitores de transacciones:

**Swapping de objetos:** Almacenar objetos en disco cuando la memoria se gasta. Si se vuelven a necesitar se recuperan ( lo gestiona CORBA ).

**Pool de Objetos:** Permite escalabilidad .Permite tener objetos ya creados de forma que cuando lo solicite el cliente lo puede utilizar y después se vuelve a guardar en el pool. De un mismo objeto tenemos varios por si lo solicita más de un cliente a la vez , esto no siempre es así , a veces no utilizamos el pool de objetos.

**Activación de objetos:** No se crea el objeto hasta que no se solicite por primera vez por algún cliente ( similar a lo que sucede con los servlets).

# Los servidores de aplicaciones .Capa intermedia

Arquitectura sobre la que se ejecuta la lógica de negocios. Siempre sobre el modelo de 3 capas. En los servidores de aplicaciones se instalas los **“componentes de negocio” .**

Son objetos que implementan la funcionalidad de la lógica de negocios. Los llaman componentes porque se busca la **reutilización** y porque hay compañías que los fabrican y otras que los usan.

Permite el desarrollo rápido de aplicaciones.

En el caso de Java se les denomina EJB (Enteprise Java Beans).

Para trabajar con un componente de negocio vamos a necesitar dos herramientas:

1. **Herramientas para el desarrollo de componente y para depurarlos** como puede ser JBuilder , Visual Age , Symantec Café versión enterprise. Pero para aprender a hacerlos los haremos primeramente a mano JDK 1.3 puede compilar EJB´s.

1. **Un contenedor de componentes de negocio:**

Proporciona un entorno ( runtime environment) en el que ejecutar esos componente. Proporciona acceso a los clientes para utilizar los componentes.

Otra tarea que realiza es dar servicios a los componentes de negocio ( por ejemplo si el bean necesita una conexión a la base de datos el contenedor se encarga de dársela ).

Administración de componentes ( instalar componentes , etc...).

Servicios a prueba de fallos (la “alta disponibilidad” ) , que no pueda fallar el sistema , que no se caiga el servidor.

Java dispone de varios servidores de aplicaciones:

**J2EE** lo da JavaSoft como prototipo de servidor de aplicaciones. No se utiliza con fines comerciales ( no está permitido por su licencia) sólo sirve de ejemplo , es lo que se denomina **“ la implementación de referencia”.**

**Bea WebLogic EJB Application Server** <http://www.bea.com> es el más antiguo de todos pero el más usado.

**Imprise Application Server** (Borland)

**NAS** (NetScape Application Server )

**WebSphere** (IBM )

**I-Planet** (versión comercial de SUN en colaboración con NetScape )

**OAS**  (Oracle Application Server )

**OpenEJB** <http://www.openejb.org>

Todos estos servidores de aplicaciones incluyen servidor web.

Independientemente del fabricante la forma de comunicar el contenedor con los componentes ha de ser estándar ( tener interfaces Java comunes ).

Es importante diferenciar entre Java Beans y EJB.

Los dos son componentes Java. Los Java Bean tienen unas reglas para la fase de desarrollo y los EJB siguen una arquitectura tanto para el desarrollo como para la distribución.

# Tecnologías de servidores de aplicaciones actuales

**Microsoft DNA** (Distributed Internet Applications 1996) : Es el servidor de aplicaciones de Microsoft. Es una solución propietaria.Como sistema operativo usa **Windows NT** como tecnología de componentes usa **DCOM** . DCOM proporciona una separación entre las interfaces ( operaciones sobre el objeto ) y su implementación (el lenguaje ). Como lenguajes utiliza **Visual Basic y Visual C**. Tiene un sistema de comunicación entre componentes mediante mensajes asíncronos , que lo que proporciona es escalabilidad al sistema , meter los mensajes en colas. Para este sistema de mensajería asíncrona se utiliza el **MSMQ** (Microsoft Message Queue ). El servidor sobre el que se instalan los componentes se llama **MTS** ( Microsoft Transaction Server). También se permite hacer un *cluster de servidores*  , un conjunto de máquinas haciendo la tarea de servidores de aplicaciones ( más escalabilidad) .Para esto utilizamos **Microsoft WolfPack.** Como base de datos utiliza **SQL Server** y como servidor web **IIS** ( Internet Information Server ).Por último disponemos de la herramienta **Microsoft Management Console** que es una utilidad de instalación y administración , con menús tipo visual basic y podemos ir viendo las propiedades de los componentes.

**Object Management Group´s CORBA Components** (OMG) **.** Especificación que aún está sin acabar , para ofrecer los servicios que no da CORBA. Existe un debate en si usar la tecnología de J2EE para servidores de aplicaciones o usar su propia tecnología.

**SUN J2EE** (Java 2 Enterprise Edition ) . Es una especificación. Es más nueva que la de Microsoft (1999).Se utilizan más por escalabilidad. Se usa mucho para Bancos. Es más fácil de utilizar porque encapsula todo lo que se llama el *middleware* ( el software sobre el que se monta nuestro software ).Sería CORBA .J2EE va sobre CORBA pero para nosotros es trasparente . Es trasparente para el programador de aplicaciones mientras que en DNA debemos de hacer objetos DCOM .

# J2EE

J2EE resuelve el problema de complejidad y coste del desarrollo de servicios multi-capa. Con J2EE podemos desarrollar sistemas seguros , eficientes y escalables.

J2EE es una arquitectura de estándard abierto que ofrece lo que es la **plataforma J2EE** y el **modelo de aplicación J2EE**

**Modelo de Aplicación J2EE** (Lógica de negocio)

**Plataforma**

**J2EE** ( Maneja detalles de bajo nivel)

J2EE comprende un conjunto de servicios , protocolos e interfaces de programación.

J2EE es una arquitectura basada en componentes para programación multicapa.

En los servicios multicapa se requieren aplicaciones cliente , lógicas de negocio y presentación que obtienen , actualizan y presentan los datos y código de infraestructura que son componentes de bajo nivel del sistema ( acceden a recursos del sistema , seguridad , etc...).

El desarrollador solo ha de preocuparse por la lógica de negocio y presentación.

#### JSP´s

#### HTML

#### EJB´s

#### Java Applet

#### Java Servlet´s

Java Aplicación

**Server-Side Presentation**

**Server-Side Data Tier**

**Server-Side Bussiness Logic**

##### Client-Side Presentation

J2EE proporciona soluciones Java entre todas las capas

Separación de tareas en la plataforma multi-capa

Conjunto de estándares abiertos ( EJB , Servlets , JDBC , JNDI , RMI ,etc...)

Portabilidad ( WORA , escribe una vez , ejecuta en cualquier parte )

Escalabilidad (Arquitectura basada en componentes integrados )

La lógica de presentación , la lógica de negocio y la lógica de acceso a datos están separadas en diferentes componentes y desplegada en varios servidores

La recolección de basura y el manejo automático de excepciones reduce el problema de que un componente bloqueé la operación del servidor u otro componente.

# Componentes J2EE

El control de transacciones , el control del ciclo de vida y los almacenes de recursos están construidos internamente en la plataforma J2EE y son proporcionados automáticamente a los componentes que soporta.

El modelo de aplicación de J2EE encapsula la funcionalidad de las capas en componentes de tipos específicos , por ejemplo la lógica de negocio está encapsulada en componentes EJB (Enterprise Java Beans ).

La iteración con los clientes puede representarse a través de :

-Páginas HTML

-Applets de Java

-Java Servlets

-Páginas JSP

-Aplicaciones Java Solitarias

Los componentes se comunican de forma trasparente usando varios estándares como HTML , XML , http, RMI, IIOP , y otros.

J2EE está compuesto por diferentes componentes:

Petición del cliente

Llamada a la lógica de negocio

#### EJB´s

#### SERVLET

Despachador a

JSP

HTML

* 1. **Plataforma .NET**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Visual Basic Net | | | | |
| Visual Basic es uno de los lenguajes de programación integrantes del entorno de desarrollo Visual Studio Net. Es un lenguaje multipropósito, totalmente orientado a objetos (forma de programar que intenta simular lo mas posible el mundo real) con control de excepciones (errores), multihilo (múltiples tareas al mismo tiempo) y que puede utilizarse para desarrollos en los que se utilice todas las tecnologías mas avanzadas de programación actuales. | | | | |
| Visual Studio Net | | | | |
| Conjunto de lenguajes de programación que están integrados en un entorno de desarrollo común.  Un entorno de desarrollo habitualmente está compuesto de un editor de código, un depurador (programa encargado de rastrear y localizar errores en los desarrollos), opciones de gestión de proyectos, herramientas visuales de desarrollo y opciones de configuración.  Es lo que se conoce como una herramienta **RAD** (desarrollo rápido de aplicaciones).  Los lenguajes que componen Visual Studio Net son: Visual Basic Net, C# (C Sharp), Visual C++ Net, Jscript Net | | | | |
| Plataforma Net | | | | |
| Es una solución que ha creado Microsoft para conseguir que los ordenadores trabajen conjuntamente a través de un medio de comunicación como puede ser Internet. No es suficiente que los ordenadores se conecten a la red, se persigue que se integren todos ellos utilizando la misma.  Para conseguir estos objetivos Microsoft se ha planteado que los entornos de desarrollo futuros tengan los siguientes componentes:  Net Building Block Services – Acceso por programa a servicios, mas que programas a medida, los usuarios van a poder acceder a servicios que resuelvan sus problemas a través de Internet.  Software de dispositivos – el futuro software no ha de estar limitado a poderse ejecutar en una sola máquina sino que ha de poderse ejecutar en múltiples máquinas como puedan ser PDA, móviles, etc.  Interfaces naturales e inteligentes – El manejo de las futuras aplicaciones ha de ser los mas natural e intuitivo posible para lo cual Microsoft dota a sus futuros desarrollos de tecnologías especiales como **etiquetas inteligentes** (son hipervínculos que automáticamente se crean atendiendo al contenido de los documentos) y otras.  Infraestructura Net (todas las tecnologías encaminadas a la creación de aplicaciones robustas, escalables y redistribuidas):   * Net Framework * Visual Studio Net * Net Enterprise Servers * Microsoft Windows Net | | | | |
| Net Framework | | | | |
| Microsoft ha construido un marco de trabajo especial para conseguir todos los objetivos anteriormente comentados.  Para saber las características mas importantes de este marco de trabajo, primeramente vamos a centrarnos en el proceso de construcción de los desarrollos actuales: | | | | |
| Codificación de los programas | | | | |
| imgm4_1.jpgimgm4\_1 | | | | |
| Cuando nosotros queremos crearnos un programa, necesitamos codificarle mediante un editor en un lenguaje de programación determinado, es lo que se denomina **código fuente**. Una vez codificado debemos compilarlo mediante un programa llamado **compilador**, proceso que se encarga de verificar y controlar si nuestro código es adecuado, y posteriormente pasar el código fuente a **código máquina** (el que realmente entiende los ordenadores). En general se produce un **fichero ejecutable**, varios ficheros de **librerías DLL** (librería de linkado dinámico, pequeños trozos de código maquina que tienen la característica de que se cargan en la memoria dinámicamente cuando se necesitan, y se descargan cuando no se necesitan) y otros archivos. | | | | |
| Instalación de los programas compilados | | | | |
| El resultado de esta compilación se ha de instalar en el ordenador cliente: | | | | |
| imgm4_2.jpg  Imgm4\_2 | | | | |
| En el proceso de instalación de nuestros programas en los ordenadores clientes, además de copiarse los diferentes ficheros del mismo, se registraran los componentes más importantes, como las DLL, en el denominado **REGISTRO DEL SISTEMA** (Base de datos en donde se almacenan todos los datos más importantes de nuestro ordenador, tanto a nivel de hardware como de software). | | | | |
| Ejecución de los programas instalados | | | | |
| imgm4_3.jpg  Imgm4\_3 | | | | |
| Cuando un programa se ejecuta, el sistema operativo crea lo que se denomina un **PROCESO** para él (un conjunto de recursos como memoria, tiempo de procesador, etc).  En un momento puede haber varios procesos en ejecución, siendo el sistema operativo el encargado de su gestión y de la seguridad para que un proceso no interfiera con otros.  Las DLL cuando se invocan, siempre han de hacerse desde un programa cliente, el sistema operativo no les asigna ningún proceso sino que se ubican dentro del proceso de la aplicación que las llama. Por eso a las DLL se las denomina programas en proceso. | | | | |
| Codificación y compilación de programas en NET | | | | |
| El trabajo dentro de la plataforma NET para el desarrollo de programas es totalmente diferente y los vamos a ver en los siguientes esquemas: | | | | |
| imgm4_5.jpg  Imgm4\_5 | | | | |
| En Visual Studio Net se encuentra un solo editor de código para todos los lenguajes de programación.  El código fuente escrito en el editor puede ser compilado por varios compiladores disponibles en el entorno de desarrollo.  A diferencia de los anteriores compiladores no se crea un fichero ejecutable Standard sino un **ENSAMBLADO**, este ensamblado puede ocupar uno o varios ficheros y en su interior hay entre otras cosas, el llamado **MSIL** (Código Intermedio de Microsoft).  Físicamente en nuestro ordenador realmente vemos ficheros con la extensión EXE o DLL, pero el contenido de ellos no es lo que nosotros conocemos, es mas, no es posible ejecutar directamente estos ficheros desde el sistema operativo.  Este código intermedio no es realmente código máquina entendible por un ordenador determinado, sino un código similar al ensamblador que es directamente entendible por el NET Framework.  En el siguiente esquema vamos a ver el contenido de un ensamblado: | | | | |
| imgm4_6.jpgimgm4\_6 | | | | |
| Un ensamblado realmente es un concepto lógico, supone la unidad que compone toda la aplicación, aunque hay aplicaciones que pueden estar compuestas por varios ensamblados. Físicamente puede estar compuesto por uno o más ficheros de tipo DLL o de tipo EXE.  En el interior de estos se encuentra en su mayor parte el código intermedio de Microsoft, pero además existe un **MANIFIESTO**.  Un manifiesto es una descripción del propio ensamblado y sus relaciones y características, podemos distinguir dentro de él la siguiente información:   * Nombre del ensamblado. * Otros archivos que lo componen. * Dependencias: referencias de ensamblados externos que son llamados. * Permisos requeridos: opciones de seguridad posibles por el ensamblado. * Tipo de ensamblado: pueden ser privados o públicos. * Lista de tipos públicos: relación de todos los tipos que se encuentran definidos en el ensamblado, relación de las clases, propiedades, métodos, enumeraciones, etc. * Atributos personalizados: características personalizadas que pueden tener varios elementos de nuestros programas. * Información de producto. Compañía, marca registrada, etc. * Información de gestión de versiones: Numero principal y secundario de versión, número de revisión y de construcción.   Un manifiesto es posible almacenarlo en un fichero independiente, aunque lo más frecuente es que se encuentre integrado dentro del propio ensamblado.  Los ensamblados pueden ser privados o públicos, esto es debido a que hay programas que utilizan el mismo código que otros, por lo que no debería ser necesario volver a codificar el mismo ensamblado varias veces.  Todo ensamblado puede especificarse que sea público o privado en cualquier aplicación, esto es labor del desarrollador cuando programa.  Los ensamblados públicos o compartidos se caracterizan por que se encuentran alojados en un directorio especifico del ordenador que Net llama **GAC** (Global Asembly Cache) caché de ensamblados globales.  Para que un ensamblado sea compartido es necesario que su nombre sea único, aunque es posible incluso que ensamblados con el mismo nombre se alojen en la caché porque estos se distinguen por el número de versión- | | | | |
| Instalación de programas en NET | | | | |
| En los ensamblados que crean los compiladores de Visual Studio Net existe información sobre los propios ensamblados y sus características de manera que son auto descriptivos, no necesitando que dicha información tenga que estar en el registro del sistema, como sucedía anteriormente.  Para instalar un programa solo es necesario copiar los ficheros que se generen directamente en el ordenador cliente. | | | | |
| Ejecución de una aplicación NET | | | | |
| imgm4_7.jpg  Imgm4\_7 | | | | |
| En el esquema se observa el proceso de ejecución de un programa.  Inicialmente los ensamblados con código intermedio son verificados para ver si cumplen las normas especificadas por Microsoft para la plataforma, después hay un conjunto de compiladores **JIT** (just in time) que compilan los ensamblados justo cuando se necesitan a un código máquina especifico del ordenador donde se ejecuten. (FICHERO EJECUTABLE PORTABLE, **PE**, es como se denomina a estos ficheros). Finalmente este código es ejecutado y controlado por el **CLR** (entorno común de ejecución).  Es posible utilizar tres tipos diferentes de Jiters, estos son:  Generadores de código en tiempo de instalación, estos compilan un ensamblaje entero, disponible para la UCP en donde se ejecute la aplicación. Normalmente se utiliza cuando sabemos que vamos a construir un programa de instalación para nuestro desarrollo.  JIT – es la opción por defecto, compila solamente cuando se ejecuta por primera vez una determinada parte del código, si se vuelve a llamar a este código ya esta compilado.  ECONOJIT – se puede producir un descarte de código, es decir que ni se ejecuta ni se compila, esto se observa cuando el equipo donde se pretende ejecutar la aplicación carece o se queda sin recursos, como puede ser un PDA. | | | | |
| imgm4_8.jpg  Imgm4\_8 | | | | |
| En el esquema anterior vemos internamente como se ejecutan las aplicaciones en el CLR. El entorno común de ejecución utiliza un solo proceso para cualquier aplicación que se este ejecutando, dentro de estos procesos establece uno o varios espacios virtuales denominados:  **DOMINIOS DE APLICACIÓN**, en estos va a ubicar todos los ensamblados que pertenecen a una sola aplicación, independientemente que sean de tipo EXE o DLL.  El CLR vela por la seguridad de cada uno de estos dominios, de manera que no interactúe uno con otro.  A todo el código que es controlado por el CLR se le denomina **CODIGO ADMINISTRADO**.  Este código se caracteriza por que puede proceder de varios lenguajes, de manera que la integridad entre lenguajes es completa. El control de errores o excepciones entre lenguajes también es pleno. Existe una seguridad que incluso se puede establecer en orden a diferentes niveles deseados por el desarrollador, de forma que el código no acceda a determinados componentes del ordenador. | | | | |
| Otros conceptos de NET Framework | | | | |
| **CTS** – (Common Type System) Sistema común de tipos, para que varios lenguajes de programación puedan trabajar de forma integrada, es necesario que todos ellos dispongan de un conjunto de tipos de datos homogéneo, resulta que Net Framework ha establecido este sistema común de tipos de datos que todo lenguaje de programación que quiera trabajar en esta plataforma ha de utilizar. Estos son los diferentes tipos establecidos: | | | | |
| **TIPOS** | | | | |
| **VALOR** | | **REFERENCIA** | | |
| **INTEGRADOS** | | **AUTODESCRIPTIVOS** | | |
| **DE USUARIO** | |  | **CLASE** | |
| **ENUMERACIONES** | |  |  | **CLASES USUARIO** |
|  |  |  |  | **VALORES ENCUADRADOS** |
|  |  |  |  | **DELEGADOS** |
|  |  |  | **MATRICES** | |
|  |  | **PUNTERO** | | |
|  |  | **IINTERFACES** | | |
|  | | | | |
|  | | | | |
| **CLS -**  (Common Language Sspecification) Especificación común de lenguaje. Son una serie de normas en cuanto a la sintaxis y semántica que han de cumplir todos los lenguajes de programación que deseen ejecutarse en la plataforma Net.  **GC –** (Garbage Collection) Recolector de basura, es una característica del CLR encargada de eliminar todos aquellos objetos que estén en nuestras aplicaciones y que no se utilicen.  **BCL - BIBLIOTECA DE CLASES BASE** – o de clases Net Framework, es un conjunto de aproximadamente 3.000 clases compiladas como ensamblados que están disponibles en cualquiera de nuestros programas, sea cual sea el lenguaje de programación utilizado.  **ESPACIOS DE NOMBRES** - es la posibilidad de crearnos estructuras jerárquicas de clases, de manera que en cada uno de estos espacios pueda haber referencias de clases particulares. Las clases base por ejemplo están organizadas en varios espacios de nombres.  **SERVICIOS NET** – son como utilidades que se les brinda a los desarrolladores para utilizarlas en sus futuros desarrollos, todos se encuentran alojados en ensamblados, podemos distinguir los siguientes.   * Servicios Web * Web Services * Web Form * ASP.NET * Servicios Windows * Win Form * GDI+ (interface de desarrollo gráfico) * Win32 * Servicios comunes * ADO.NET (objetos para la conexión a fuentes de datos) * XML (lenguaje de marcado extendido) * SOAP (protocolo simple de acceso a objetos) * Gestión de entradas y salidas * Gestión de red * Gestión de seguridad | | | | |
| Resumen de características de NET Framework | | | | |
| De lo anteriormente comentado podemos deducir:  -Las aplicaciones que se desarrollen en esta plataforma disponen de un mayor rendimiento.  -Existe una gestión automática de la memoria, y de la eliminación de objetos no utilizados.  -Tanto en la Maquina de desarrollo como en la Maquina donde se ejecuten los programas, es necesario instalar el denominado Net Framework.  -Existe una compatibilidad a nivel de lenguajes de programación, de manera que los programas futuros se pueden crear con varios lenguajes.  -Existe gran facilidad de utilización de componentes (trozos de código compilados) creados con otros lenguajes.  -Todos los lenguajes disponen de características completas de programación orientada a objetos.  El código intermedio de Microsoft va a poder ser ejecutado en cualquier máquina que disponga del Net Framework, puesto que en estas máquinas se supone que estarán compiladores JIT y un CLR adecuado tanto para el sistema operativo como para el microprocesador. Incluso en maquinas de tipo PDA en el que se encuentre un tipo de Framework especial.  La unidad mínima de código que se crean con los compiladores de Visual Studio Net es el ensamblado.  Se aprovecha la simplicidad del Visual Basic con la potencia del C++.  Una particularidad de los lenguajes de programación llamada punteros (variables que contienen la dirección de otra variable) es sustituida por los llamados delegados, más flexibles y menos problemáticos. | | | | |
| El esquema siguiente representa las posibilidades de comunicación entre las aplicaciones NET.  imgm4_9.jpg  Imgm4\_9 | | | | |
| La comunicación entre maquinas de todo tipo se realiza mediante el intercambio de mensajes, estos mensajes están en formato **XML** que se ha convertido en un Standard de formato de datos no solamente en Internet sino entre aplicaciones de todo tipo... Estos mensajes utilizan los protocolos de Internet HTTP, TCP/IP, SOAP, SMTP que son estándares en los sistemas de comunicación.  Esquema general de todos los componentes de Visual Studio y Net Framwork integrados con el sistema operativo. | | | | |
| imgm4_10.jpg  Imgm4\_10 | | | | |
| Instalación del programa | | | | |
| El programa se puede instalar de varias maneras, viene acompañado de siete CD y un DVD.  Es necesario tener el permiso de Administrador en el ordenador donde se instala el producto, debido a que algunas herramientas pueden crear cuentas de usuario, sin embargo después de la instalación no es necesario permiso alguno, salvo a algunas herramientas. | | | | |

* 1. **Tecnologías Móviles**

****

Imgm4\_11

Hoy en día los móviles se han convertido en una herramienta de trabajo, diversión y entretenimiento pasando casi casi la propia función de lo que es un mero teléfono a segundo plano.

Todo ello a través d elos smart phones y a los sistemas operativos Android e IOS.

Previamente a Android la tecnología basada en Java para programar los dispositivos móviles era J2ME y aunque aún se sigue usando cada vez se está imponiendo más Android como tecnología basada en Java.

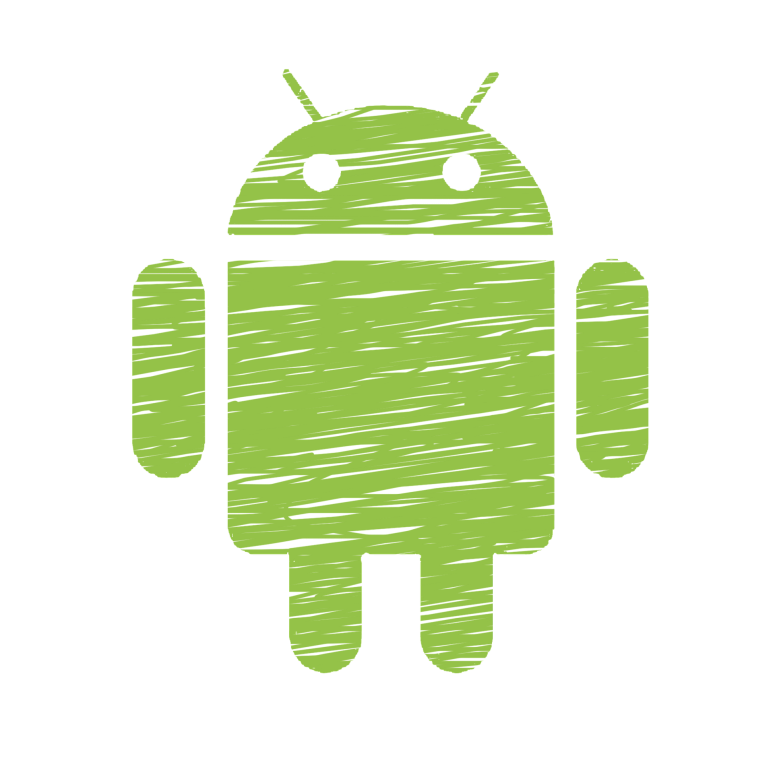
**J2ME**

* J2ME provee de un entorno de desarrollo para la creación de aplicaciones para dispositivos móviles.
* Tal como J2EE o J2SE, J2ME incluye JVMs y un conjunto de APIs para el desarrollo.
* Las aplicaciones basadas en las especificaciones de J2ME se escriben para un conjunto amplio de dispositivos, pero pueden aprovechar las capacidades especificas de cada uno de ellos.

La plataforma J2ME se encuentra implementada en millones de dispositivos, y es en definitiva la plataforma elegida hoy en día para la programación de dispositivos móviles.

Java ME fue desarrollado mediante el Java Community Process bajo la especificación JSR 68.

**Android**

****

Imgm4\_12

**Android** es un [sistema operativo móvil](https://es.wikipedia.org/wiki/Sistema_operativo_m%C3%B3vil) dcreado por [Google](https://es.wikipedia.org/wiki/Google), basado en el [Kernel de Linux](https://es.wikipedia.org/wiki/N%C3%BAcleo_Linux) y otros [software de código abierto](https://es.wikipedia.org/wiki/Software_de_c%C3%B3digo_abierto)(open source).

.Fue desarrollado para [dispositivos móviles](https://es.wikipedia.org/wiki/Dispositivo_m%C3%B3vil) con [pantalla táctil](https://es.wikipedia.org/wiki/Pantalla_t%C3%A1ctil), como [teléfonos inteligentes](https://es.wikipedia.org/wiki/Tel%C3%A9fono_inteligente), tablets, [relojes inteligentes](https://es.wikipedia.org/wiki/Reloj_inteligente), [coches](https://es.wikipedia.org/wiki/Autom%C3%B3vil) y [televisores](https://es.wikipedia.org/wiki/Televisor), así como otros dispositivos electrónicos de mano.

Inicialmente fue desarrollado por [Android Inc.](https://es.wikipedia.org/wiki/Android_Inc.), empresa que [Google](https://es.wikipedia.org/wiki/Google) compró en 2005.

Android fue presentado en 2007 junto con la fundación del [Open Handset Alliance](https://es.wikipedia.org/wiki/Open_Handset_Alliance)(un consorcio de compañías de [hardware](https://es.wikipedia.org/wiki/Hardware), [software](https://es.wikipedia.org/wiki/Software) y [telecomunicaciones](https://es.wikipedia.org/wiki/Telecomunicaciones)) para avanzar en los estándares abiertos de los dispositivos móviles.

En Java o .NET estamos acostumbrados a manejar conceptos como ventana, control, eventos o servicios como los elementos básicos en la construcción de una aplicación.

Pues bien, en Android vamos a disponer de esos mismos elementos básicos aunque con un pequeño cambio en la terminología y el enfoque. Repasemos los componentes principales que pueden formar parte de una aplicación Android

**Activity**

Las actividades (*activities*) representan el componente principal de la interfaz gráfica de una aplicación Android. Se puede pensar en una actividad como el elemento análogo a una ventana o pantalla en cualquier otro lenguaje visual.

**View**

Las vistas (*view*) son los componentes básicos con los que se construye la interfaz gráfica de la aplicación, análogo por ejemplo a los *controles* de Java o .NET. De inicio, Android pone a nuestra disposición una gran cantidad de controles básicos, como cuadros de texto, botones, listas desplegables o imágenes, aunque también existe la posibilidad de extender la funcionalidad de estos controles básicos o crear nuestros propios controles personalizados.

**Service**

Los servicios (*service*) son componentes sin interfaz gráfica que se ejecutan en segundo plano. En concepto, son similares a los servicios presentes en cualquier otro sistema operativo. Los servicios pueden realizar cualquier tipo de acciones, por ejemplo actualizar datos, lanzar notificaciones, o incluso mostrar elementos visuales (p.ej. actividades) si se necesita en algún momento la interacción con del usuario.

**Content Provider**

Un proveedor de contenidos (*content provider*) es el mecanismo que se ha definido en Android para compartir datos entre aplicaciones. Mediante estos componentes es posible compartir determinados datos de nuestra aplicación sin mostrar detalles sobre su almacenamiento interno, su estructura, o su implementación. De la misma forma, nuestra aplicación podrá acceder a los datos de otra a través de los *content provider* que se hayan definido.

**Broadcast Receiver**

Un *broadcast receiver* es un componente destinado a detectar y reaccionar ante determinados mensajes o eventos globales generados por el sistema  (por ejemplo: “Batería baja”, “SMS recibido”, “Tarjeta SD insertada”) o por otras aplicaciones (cualquier aplicación puede generar mensajes (*intents*, en terminología Android) broadcast, es decir, no dirigidos a una aplicación concreta sino a cualquiera que quiera escucharlo).

**Widget**

Los *widgets* son elementos visuales, normalmente interactivos, que pueden mostrarse en la pantalla principal (*home screen*) del dispositivo Android y recibir actualizaciones periódicas. Permiten mostrar información de la aplicación al usuario directamente sobre la pantalla principal.

**Intent**

Un *intent* es el elemento básico de comunicación entre los distintos componentes Android que hemos descrito anteriormente. Se pueden entender como los mensajes o peticiones que son enviados entre los distintos componentes de una aplicación o entre distintas aplicaciones. Mediante un *intent* se puede mostrar una actividad desde cualquier otra, iniciar un servicio, enviar un mensaje *broadcast*, iniciar otra aplicación, etc.

Para desarrollar los programas en Android vamos a utilizar el entorno de desarrollo Android Studio.

Android Studio es un IDE bastante grande que nos proporciona los medios para construir aplicaciones para móviles y tablets de forma sencilla y consta d emuchas plantillas ya creadas para poder realizar deforma rápida APPS adaptadas a demás a todo tipo de teléfonos y tablets.

Es una especie de Net Beans pero para el desarrollo de aplicaciones móviles.



Imgm4\_13