



Universidad de Navarra
Nafarroako Unibertsitatea

Escuela Superior de Ingenieros
Ingeniarien Goi Mailako Eskola

INFORMÁTICA II

Nicolás Serrano Bárcena
Dr. Ingeniero Industrial

Fernando Alonso Blázquez
Dr. Ingeniero Industrial

Sonia Calzada Mínguez
Dra. Ingeniero Industrial

San Sebastián, Enero de 2010

tecnun

CAMPUS TECNOLÓGICO DE LA UNIVERSIDAD DE NAVARRA. NAFARROAKO UNIBERTSITATEKO CAMPUS TEKNOLOGIKOA
Paseo de Manuel Lardizábal 13. 20018 Donostia-San Sebastián. Tel.: 943 219 877 Fax: 943 311 442 www.tecnun.es informacion@tecnun.es

Prefacio

El presente libro es el material teórico y ejemplos de la asignatura **Informática II**. Está basado en el libro de la asignatura **Informática III**, realizado por Fernando Alonso, con las modificaciones requeridas por el cambio de curso y contenido de la asignatura. Por ello este prefacio contiene parte del mismo escrito en Febrero de 2005:

Debido a la variedad de temas que se tocan en la asignatura, cada uno de ellos es tratado de forma muy somera. Este documento es pues, básicamente, una recopilación de varios manuales de diferente naturaleza y temática. En concreto, se han incorporado partes, realizando ciertas actualizaciones, de los siguientes manuales de la colección "**Aprenda Informática como si estuviera en primero**", disponibles en la Escuela Superior de Ingenieros, TECNUN:

- Aprenda Internet como si estuviera en primero.
- Aprenda Java como si estuviera en primero.
- Aprenda Servlets de Java como si estuviera en primero.

Esta versión incluye también los capítulos 2, 3 y 4 correspondientes al libro "Aprenda Java como si estuviera en primero" cuyos autores son: Javier García de Jalón, José Ignacio Rodríguez, Íñigo Mingo, Aitor Imaz, Alfonso Brazález, Alberto Larzabal, Jesús Calleja y Jon García

El libro tiene la estructura que se sigue en la asignatura, comenzando por el lenguaje Java, dos capítulos dedicados a Internet y el lenguaje HTML, la parte central corresponde al desarrollo de aplicaciones Web con Servlets de Java y finalmente hay dos capítulos de interface de usuario tanto para aplicaciones Web como de escritorio.

El hilo conductor de todos estos temas es conseguir diseñar, programar y poner en funcionamiento **Sistemas de Información en entorno Web**.

Este libro no tiene la intención de ser un manual de referencia para ninguno de los temas que trata, ya que se queda a un nivel introductorio en todos ellos, sino que debe tomarse como si de notas de clase se tratara y que por sí solas no serán de mucha utilidad para el alumno. Es fundamental la asistencia a clase y sobre todo la realización de las prácticas semanales que es donde se aplica lo visto en las exposiciones teóricas.

El complemento de estos apuntes es la **página web de la asignatura de Informática II** (<http://www.tecnun.es/asignaturas/Informat2>)

Nicolás Serrano

Enero de 2010

Índice de temas

Parte 1. Java

1. Introducción a Java	3
2. Programación en Java	13
3. Clases en Java	27
4. Librerías, herencia e interfaces	37
5. Clases de utilidad	47

Parte 2. Internet

6. Internet y HTML	71
7. Formularios y CGIs	87

Parte 3. Aplicaciones Web

8. Servlets	97
9. Servlet API	109
10. Sesión en Servlets	121

Parte 4. Graphical User Interface

11. JavaScript	129
12. GUI y otros elementos de Java	151

Bibliografía	163
--------------	-----

Índice

1. Introducción a Java	17
1.1. Programación Orientada a Objetos. Fundamentos	17
1.2. El lenguaje de programación Java.....	18
1.3. Características generales de Java	20
1.4. Entornos de desarrollo de Java	20
1.4.1. Java SE Development Kit (JDK).....	20
1.4.1.1. Instalación.....	21
1.4.1.2. Documentación	22
1.4.1.3. Manos a la obra.....	22
1.4.2. Entornos IDE (Integrated Development Environment).....	23
1.5. Estructura general de un programa en Java	24
1.5.1. Concepto de Clase	24
1.5.2. Herencia	24
1.5.3. Concepto de Interface	24
1.5.4. Concepto de Package	24
1.5.5. La jerarquía de clases de Java (API)	24
2. Programación en Java	27
2.1. Variables	27
2.1.1. Nombres de Variables	27
2.1.2. Tipos Primitivos de Variables	28
2.1.3. Cómo se definen e inicializan las variables	28
2.1.4. Visibilidad y vida de las variables.....	29
2.1.5. Casos especiales: Clases BigInteger y BigDecimal.....	30
2.2. Operadores de Java	30
2.2.1. Operadores aritméticos.....	30
2.2.2. Operadores de asignación	30
2.2.3. Operadores unarios	31
2.2.4. Operador instanceof.....	31
2.2.5. Operador condicional ?:	31
2.2.6. Operadores incrementales	31
2.2.7. Operadores relacionales	31
2.2.8. Operadores lógicos	32
2.2.9. Operador de concatenación de cadenas de caracteres (+).....	32
2.2.10. Operadores que actúan a nivel de bits	32
2.2.11. Precedencia de operadores.....	33
2.3. Estructuras de programación	34

2.3.1. Sentencias o expresiones	34
2.3.2. Comentarios	34
2.3.3. Bifurcaciones	34
2.3.3.1. Bifurcación if	35
2.3.3.2. Bifurcación if else.....	35
2.3.3.3. Bifurcación if elseif else	35
2.3.3.4. Sentencia switch	35
2.3.4. Bucles	36
2.3.4.1. Bucle while.....	36
2.3.4.2. Bucle for	36
2.3.4.3. Bucle do while	37
2.3.4.4. Sentencias break y continue	37
2.3.4.5. Sentencias break y continue con etiquetas	37
2.3.4.6. Sentencia return	38
2.3.4.7. Bloque try {...} catch {...} finally {...}	38
3. Clases en Java.....	41
3.1. Conceptos básicos	41
3.1.1. Concepto de Clase.....	41
3.1.2. Concepto de Interface	42
3.2. Ejemplo de definición de una clase	42
3.3. Variables miembro	43
3.3.1. Variables miembro de objeto.....	43
3.3.2. Variables miembro de clase (static).....	44
3.4. variables finales	44
3.5. Métodos (funciones miembro).....	45
3.5.1. Métodos de objeto.....	45
3.5.2. Métodos sobrecargados (overloaded)	46
3.5.3. Paso de argumentos a métodos	46
3.5.4. Métodos de clase (static).....	47
3.5.5. Constructores.....	47
3.5.6. Inicializadores	48
3.5.6.1. Inicializadores static	48
3.5.6.2. Inicializadores de objeto	48
3.5.7. Resumen del proceso de creación de un objeto.....	48
3.5.8. Destrucción de objetos (liberación de memoria)	49
3.5.9. Finalizadores.....	49
4. Librerías, herencia e interfaces	51
4.1. Packages	51
4.1.1. Qué es un package.....	51
4.1.2. Cómo funcionan los packages.....	51
4.2. Herencia	52
4.2.1. Concepto de herencia.....	52

Índice	iii
4.2.2. La clase Object.....	53
4.2.3. Redefinición de métodos heredados	53
4.2.4. Clases y métodos abstractos.....	53
4.2.5. Constructores en clases derivadas.....	54
4.3. Clases y métodos finales	54
4.4. Interfaces	54
4.4.1. Concepto de interface	54
4.4.2. Definición de interfaces	55
4.4.3. Herencia en interfaces.....	55
4.4.4. Utilización de interfaces	56
4.5. Permisos de acceso en Java	56
4.5.1. Accesibilidad de los packages	56
4.5.2. Accesibilidad de clases o interfaces.....	56
4.5.3. Accesibilidad de las variables y métodos miembros de una clase:	56
4.6. Transformaciones de tipo: casting	57
4.6.1. Conversión de tipos primitivos	57
4.7. Polimorfismo	57
4.7.1. Conversión de objetos	58
5. Clases de utilidad.....	61
5.1. Arrays	61
5.1.1. Arrays bidimensionales	62
5.2. Clases String y StringBuffer	63
5.2.1. Métodos de la clase String	63
5.2.2. Métodos de la clase StringBuffer.....	64
5.3. Wrappers	65
5.3.1. Clase Double	65
5.3.2. Clase Integer.....	66
5.4. Clase Math	67
5.5. Colecciones	67
5.5.1. Clase Vector	67
5.5.2. Interface Enumeration	68
5.5.3. Clase Hashtable	69
5.5.4. El Collections Framework de Java 1.2	70
5.5.4.1. Elementos del Java Collections Framework	71
5.5.4.2. Interface Collection	72
5.5.4.3. Interfaces Iterator y ListIterator	73
5.5.4.4. Interfaces Comparable y Comparator	74
5.5.4.5. Sets y SortedSets	74
5.5.4.6. Listas	75
5.5.4.7. Maps y SortedMaps.....	76
5.5.4.8. Algoritmos y otras características especiales: Clases Collections y Arrays	77
5.5.4.9. Desarrollo de clases por el usuario: clases abstract	78

5.5.4.10. Interfaces Cloneable y Serializable	78
5.6. Otras clases del package java.util.....	78
5.6.1. Clase Date.....	78
5.6.2. Clases Calendar y GregorianCalendar	79
5.6.3. Clases DateFormat y SimpleDateFormat.....	80
5.6.4. Clases TimeZone y SimpleTimeZone	81
6. Internet y HTML	85
6.1. Protocolo TCP/IP	85
6.2. Protocolo HTTP y lenguaje HTML.....	85
6.3. URL (Uniform Resource Locator).....	85
6.3.1. URLs del protocolo HTTP	87
6.3.2. URLs del protocolo FTP	87
6.3.3. URLs del protocolo correo electrónico (mailto)	87
6.3.4. URLs del protocolo Telnet.....	87
6.3.5. Nombres específicos de ficheros	87
6.4. HTML	87
6.5. Tags generales	88
6.6. Formato de texto	89
6.6.1. TAGs generales de un documento.	89
6.6.2. Comentarios en HTML	89
6.6.3. Caracteres de separación en HTML	89
6.6.4. TAGs de organización o partición de un documento	90
6.6.5. Formateo de textos sin particiones.....	90
6.6.6. Centrado de párrafos y cabeceras con <CENTER>	91
6.6.7. Efectos de formato en texto.....	91
6.6.8. Caracteres especiales	91
6.7. Listas.....	92
6.7.1. Listas desordenadas	92
6.7.2. Listas ordenadas	92
6.7.3. Listas de definiciones	92
6.8. Imágenes	93
6.9. Links	93
6.10. Tablas	94
6.11. Frames	95
6.11.1. Introducción a los frames	95
6.11.2. Salida a ventanas o frames concretas de un browser.....	97
6.11.3. Nombres de TARGET	98
6.12. Mapas de imágenes o imágenes sensibles.....	98
6.12.1. Cómo funciona un mapa de imágenes	98
6.12.2. Mapas de imágenes procesados por el browser	98
6.13. Editores y conversores HTML.....	99
7. Formularios y CGIs	101

7.1. Formularios (Forms).....	101
7.2. Programas CGI.....	106
8. Servlets	111
8.1. Clientes y Servidores.....	111
8.1.1. Clientes (clients).....	111
8.1.2. Servidores (servers)	111
8.2. Tendencias Actuales para las aplicaciones en Internet	112
8.3. Diferencias entre las tecnologías CGI y Servlet	114
8.4. Características de los <i>servlets</i>	114
8.5. Servidor de aplicaciones.....	115
8.5.1. Visión general del Servlet API	115
8.5.2. Apache Tomcat.....	116
8.6. Ejemplo Introductorio	117
8.6.1. Formulario	117
8.6.2. Código del Servlet	118
9. Servlet API	123
9.1. El ciclo de vida de un servlet: clase GenericServlet	123
9.1.1. El método init() en la clase GenericServlet	124
9.1.2. El método service() en la clase GenericServlet	124
9.1.3. El método destroy() en la clase GenericServlet:.....	125
9.2. El contexto del servlet (servlet context).....	128
9.2.1. Información durante la inicialización del servlet	128
9.2.2. Información contextual acerca del servidor	128
9.3. Clases de utilidades (Utility Classes).....	129
9.4. Clase HttpServlet: soporte específico para el protocolo HTTP.....	130
9.4.1. Método GET: codificación de URLs	130
9.4.2. Método HEAD: información de ficheros	131
9.4.3. Método POST: el más utilizado	132
9.4.4. Clases de soporte HTTP	132
9.4.5. Modo de empleo de la clase HttpServlet	133
10. Sesión en Servlets	135
10.1. Formas de seguir la trayectoria de los usuarios	135
10.2. Cookies.....	135
10.2.1. Crear un objeto Cookie.....	136
10.2.2. Establecer los atributos de la cookie	136
10.2.3. Enviar la cookie.....	137
10.2.4. Recoger las cookies	137
10.2.5. Obtener el valor de la cookie	138
10.3. Sesiones (Session Tracking).....	138
10.4. Reescritura de URLs	140
11. JavaScript	143
11.1. Introducción.....	143
11.1.1. Propiedades del Lenguaje JavaScript.....	143

11.1.2. El lenguaje JavaScript	143
11.1.3. Variables y valores	144
11.1.4. Sentencias, Expresiones y Operadores	144
11.1.5. Estructuras de Control.....	144
11.1.6. Funciones y Objetos	144
11.1.6.1. Funciones	145
11.1.6.2. Objetos	145
11.1.7. La TAG «Script».....	145
11.2. Activación de JavaScript: Eventos (Events)	146
11.2.1. Eventos y acciones	147
11.2.1.1. Acciones de Navegación y Eventos	147
11.2.2. Gestores de Eventos (Event Handlers).....	147
11.2.2.1. Declaración	147
11.2.2.2. Uso	148
11.2.2.2.1. Gestores a nivel de documento	148
11.2.2.2.2. Gestores a nivel de formulario	149
11.2.2.2.3. Gestores a nivel de elementos de formulario	149
11.3. Clases en JavaScript	150
11.3.1. Clases Predefinidas (Built-In Objects)	150
11.3.1.1. Clase String	150
11.3.1.2. Clase Math	151
11.3.1.3. Clase Date	152
11.3.2. Funciones Predefinidas (Built-in Functions): eval, parseFloat, parseInt	153
11.3.2.1. eval(string)	153
11.3.2.2. parseFloat(string).....	154
11.3.2.3. parseInt(string, base)	154
11.3.3. Clases del browser	154
11.3.3.1. Clase Window	154
11.3.3.2. Clase Document	155
11.3.3.3. Clase Location.....	155
11.3.3.4. Clase History	156
11.3.4. Clases del documento HTML (anchors, forms, links)	156
11.4. Clases y Funciones definidas por el usuario	157
11.4.1. Funciones (métodos).....	158
11.4.2. Objetos como Arrays (Vectores)	159
11.4.3. Extender Objetos.....	159
11.4.4. Funciones con un número variable de argumentos.....	160
11.5. Expresiones y operadores de JavaScript	160
11.5.1. Expresiones	160
11.5.1.1. Expresiones Condicionales	160
11.5.2. Operadores de asignación (=, +=, -=, *=, /=)	161

11.5.3. Operadores aritméticos	161
11.5.4. Operadores lógicos	161
11.5.5. Operadores de Comparación ($=$, $>$, \geq , $<$, \leq , $!=$)	162
11.5.6. Operadores de String	162
11.5.7. Prioridad de los operadores	162
11.6. Sentencias de control de JavaScript	163
11.6.1. La sentencia if	163
11.6.2. Bucles.....	163
11.6.2.1. Bucle for	163
11.6.2.2. Bucle while	164
11.7. Comentarios.....	164
12. Graphical User Interface	165
12.1. Graphical User Interface (GUI)	165
12.1.1. Componentes gráficos: Abstract Window Toolkit (AWT)	165
12.1.1.1. Widgets o componentes elementales	165
12.1.1.2. Métodos de organización: Contenedores	165
12.1.1.3. Diseño Visual: Layouts	165
12.1.2. Eventos	167
12.1.3. Applets	172
12.2. Otros elementos de Java.....	173
12.2.1. Manejo de Excepciones y Errores.....	173
12.2.2. Entrada/Salida de Datos	174
12.2.3. Subprocesos	175
Bibliografía	177

Parte 1

Java

1. Introducción a Java

1.1. Programación Orientada a Objetos. Fundamentos.

La producción de aplicaciones de altas prestaciones suele significar la presencia de una complejidad cada vez mayor. Los sistemas orientados a objetos tienen características adecuadas para expresar la complejidad de un sistema, algunas de las cuales son:

- **Adaptabilidad**, es decir, facilidad de transporte de unos sistemas a otros.
- **Reusabilidad**, total o parcial, para reducir costes y reutilizar componentes software cuya fiabilidad está comprobada.
- **Mantenibilidad**: Los programas son desarrollados por muchas personas agrupadas en equipos de trabajo. Las personas cambian pero la aplicación permanece e incluso necesita modificaciones. Por ello, es importante que los programas sean fáciles de comprender y mantener. En caso contrario, podría ser necesario descartar la aplicación y hacer una nueva.

Para conseguir estos objetivos, es necesario aplicar de forma rigurosa criterios de diseño claros y bien definidos que permitan hacer frente a la complejidad de las aplicaciones. El **diseño orientado a objetos** es la metodología que consiste en definir cuáles son los objetos de un sistema, las clases en las que pueden agruparse y las relaciones entre objetos.

Las características fundamentales de la **Programación Orientada a Objetos (POO)** son:

- **Abstracción**: Es la representación de las características esenciales de algo sin incluir los antecedentes o detalles irrelevantes. La clase es una abstracción porque en ella se definen las propiedades y los atributos genéricos de un conjunto de objetos
- **Encapsulación** u ocultamiento de información: Las variables y las funciones miembro de una clase pueden ser declaradas como *public*, *private* o *protected*. De esta forma se puede controlar el acceso a los miembros de la clase y evitar un uso inadecuado.
- **Herencia**: Es el mecanismo para compartir automáticamente métodos y atributos entre clases y subclases. Una clase puede derivar de otra, y en este caso hereda todas las variables y funciones miembro. Así, puede añadir nuevas funciones y datos miembros.
- **Polimorfismo**: Esta característica permite implementar múltiples formas de un mismo método, dependiendo cada una de ellas de la clase sobre la que se realice la implementación.

Una **Clase** es un tipo de datos definido por el usuario consistente, básicamente, en una agrupación de las definiciones de los datos (**variables**) y de las funciones (**métodos**) que operan sobre esos datos. Un **Objeto** es un ejemplar concreto de una clase: un conjunto concreto de datos y de los métodos para manipular éstos. La creación de un objeto a partir de una clase se denomina **instanciación**, es decir, crear una instancia concreta de la clase.

La definición de una clase consta de dos partes:

- La primera, formada por el nombre de la clase precedido por la palabra reservada *class*.
- La segunda parte es el cuerpo de la clase, encerrado entre llaves, y que consta de:
 - **Especificadores de acceso**: *public*, *protected* y *private*.
 - **Atributos**: datos miembro de la clase (variables).
 - **Métodos**: definiciones de funciones miembro de la clase.

Una forma de asegurar que los objetos siempre contengan valores válidos y que puedan ser inicializados en el momento de la declaración es escribiendo un **constructor**. Un constructor es una función miembro especial de una clase que es llamada de forma automática siempre que se declara un objeto de esa clase. Su función es crear e inicializar un objeto de su clase. Dado que un constructor es una función miembro, admite argumentos al igual que éstas. El constructor se puede distinguir claramente, con respecto a las demás funciones miembro de la clase, porque tiene el mismo nombre que el de la clase. Un constructor no retorna ningún valor ni se hereda. Si el usuario no ha creado uno, el compilador crea uno por omisión, sin argumentos. Una clase puede tener varios constructores, siempre que tengan distintos argumentos de entrada.

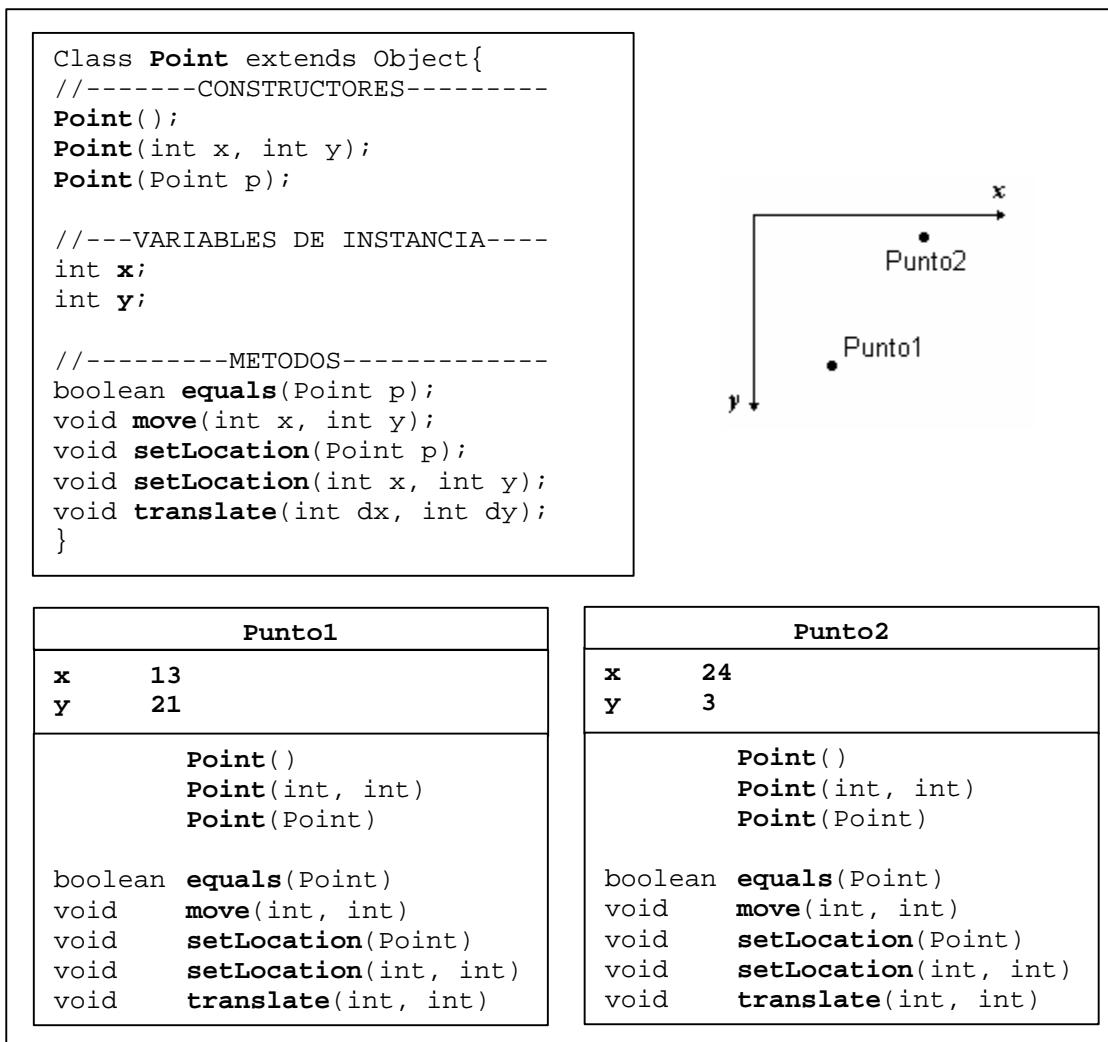


Figura 1.1. Clase Point y dos objetos de la clase Point.

1.2. El lenguaje de programación Java

Java surgió en 1991 cuando un grupo de ingenieros de *Sun Microsystems Inc.* trataron de diseñar un nuevo lenguaje de programación destinado a electrodomésticos. La reducida potencia de cálculo y memoria de los electrodomésticos llevó a desarrollar un lenguaje sencillo capaz de generar código de tamaño muy reducido.

Debido a la existencia de distintos tipos de CPUs y a los continuos cambios, era importante conseguir una herramienta independiente del tipo de CPU utilizada. Esto hizo de **Java** un lenguaje ideal para distribuir programas ejecutables vía la WWW, además de un lenguaje de programación de propósito general para desarrollar programas que sean fáciles de usar y portables en una gran variedad de plataformas. Desarrollaron un código "neutro" que no dependía del tipo de electrodoméstico, el cual se

ejecutaba sobre una "máquina hipotética o virtual" denominada **Java Virtual Machine (JVM)**. Era la **JVM** quien interpretaba el código neutro convirtiéndolo a código particular de la CPU utilizada. Esto permitía lo que luego se ha convertido en el principal lema del lenguaje: "*Write Once, Run Everywhere*". A pesar de los esfuerzos realizados por sus creadores, ninguna empresa de electrodomésticos se interesó por el nuevo lenguaje.

Como lenguaje de programación para computadoras, **Java** se introdujo a finales de 1995. Se usó en varios proyectos de Sun (en aquel entonces se llamaba Oak) sin mucho éxito comercial. Se difundió más cuando se unió con HotJava, un navegador Web experimental, para bajar y ejecutar subprogramas (los futuros applets). La clave del éxito fue la incorporación de un intérprete **Java** en la versión 2.0 del programa Netscape Navigator en 1994, produciendo una verdadera revolución en Internet. Obtuvo tanta atención que en Sun la división de **Java** se separó en la subsidiaria JavaSoft.

Java 1.1 apareció a principios de 1997, mejorando sustancialmente la primera versión del lenguaje. **Java 1.2**, más tarde rebautizado como **Java 2**, nació a finales de 1998.

A nivel de código fuente, los tipos de datos primitivos de Java tienen tamaños consistentes en todas las plataformas de desarrollo. Las bibliotecas de clases fundamentales en Java facilitan la escritura de código, el cual puede ser transportado de una plataforma a otra sin necesidad de rescribirlo para que trabaje en la nueva plataforma. Lo anterior también se aplica al código binario. Se puede ejecutar sin necesidad de recompilarlo.

Con los lenguajes convencionales al compilar se genera código binario para una plataforma en particular. Si se cambia la plataforma se tendrá que recompilar el programa. Por ejemplo un programa en C para una PC, no servirá en un servidor UNIX y viceversa.

Al programar en **Java** no se parte de cero. Cualquier aplicación que se desarrolle "cuelga" (o se apoya, según como se quiera ver) en un gran número de **clases** preexistentes. Algunas de ellas las ha podido hacer el propio usuario, otras pueden ser comerciales, pero siempre hay un número muy importante de clases que forman parte del propio lenguaje (el **API o Application Programming Interface de Java**). **Java** incorpora en el propio lenguaje muchos aspectos que en cualquier otro lenguaje son extensiones propiedad de empresas de software o fabricantes de ordenadores (threads, ejecución remota, componentes, seguridad, acceso a bases de datos, etc.). Por eso muchos expertos opinan que **Java** es el lenguaje ideal para aprender la informática moderna, porque incorpora todos estos conceptos de un modo estándar, mucho más sencillo y claro que con las citadas extensiones de otros lenguajes. Esto es consecuencia de haber sido diseñado más recientemente y por un único equipo.

El principal objetivo del lenguaje **Java** es llegar a ser el "nexo universal" que conecte a los usuarios con la información, esté ésta situada en la computadora local, en un servidor de **Web**, en una base de datos o en cualquier otro lugar.

Java es un lenguaje muy completo (de hecho se está convirtiendo en un macrolenguaje: **Java 1.0** tenía 8 paquetes; **Java 1.1** tenía 23 y **Java 1.2** tenía 59 y **Java 1.5** tenía 166 packages). En cierta forma casi todo depende de casi todo. Por ello, conviene aprenderlo de modo *iterativo*: primero una visión muy general, que se va refinando en sucesivas iteraciones. Una forma de hacerlo es empezar con un ejemplo completo en el que ya aparecen algunas de las características más importantes.

Java 2 (antes llamado Java 1.2 o JDK 1.2) es la tercera versión importante del lenguaje de programación Java. No hay cambios conceptuales importantes respecto a **Java 1.1** (en **Java 1.1** sí los hubo respecto a **Java 1.0**), sino extensiones y ampliaciones, lo cual hace que a muchos efectos, sea casi lo mismo trabajar con **Java 1.1** o con **Java 1.2**. A partir de esta versión, hay dos formas de denominar a los productos, por ejemplo **Java 1.5** o **Java 5**; **Java 1.6** o **Java 6**.

La compañía **Sun** describe el lenguaje **Java** como "*simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de altas prestaciones, multitarea y dinámico*". Además de una serie de halagos por parte de **Sun** hacia su propia criatura, el hecho es que todo ello describe bastante bien el lenguaje **Java**, aunque en algunas de esas características el lenguaje sea todavía bastante mejorable.

1.3. Características generales de Java

Java es un lenguaje de propósito general, de alto nivel y orientado a objetos. **Java** es un lenguaje compilado e interpretado. Una de las herramientas de desarrollo es el **compilador de Java**, que realiza un análisis de sintaxis del código escrito en los ficheros fuente de **Java** (con extensión ***.java**). Si no encuentra errores en el código genera los denominados “**bytecodes**” o ficheros compilados (con extensión ***.class**). En otro caso muestra la línea o líneas donde se han encontrado errores.

Por otro lado, la denominada **Java Virtual Machine (JVM)** anteriormente mencionada es el **intérprete de Java**, de forma que convierte el código neutro a código particular de la CPU que se está utilizando. Se evita tener que realizar un programa diferente para cada CPU o plataforma. La **JVM** ejecuta los “**bytecodes**” (ficheros compilados con extensión ***.class**) creados por el compilador de Java.

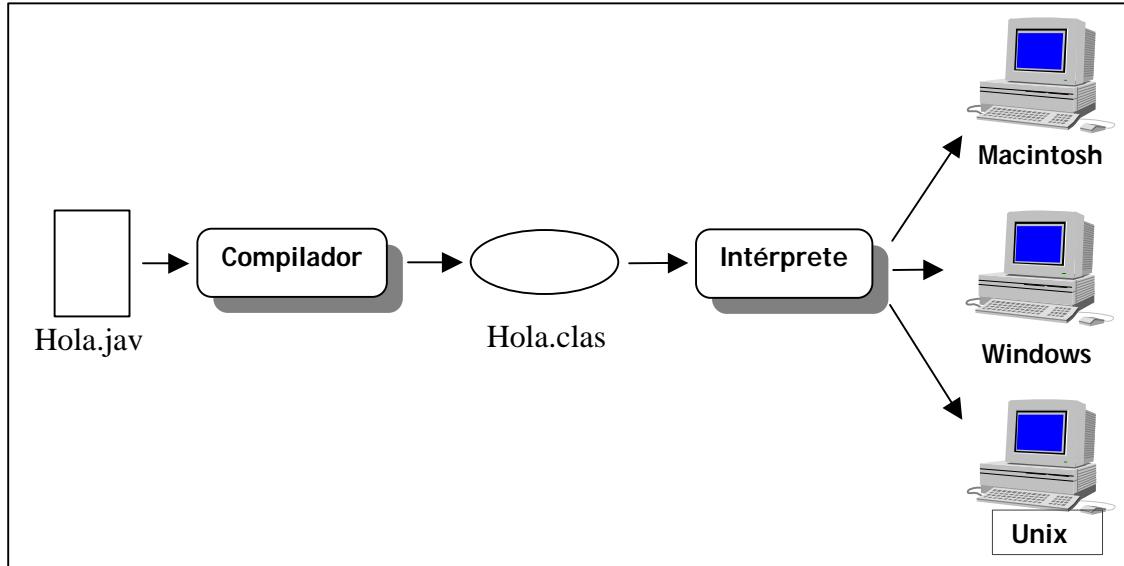


Figura 1.2. Java, un lenguaje compilado e interpretado.

La **API** de **Java** es muy rica, está formada por un conjunto de paquetes de clases que le proporcionan una gran funcionalidad. El núcleo de la **API** viene con cada una de las implementaciones de la **Java Virtual Machine**.

Java ofrece la posibilidad de crear programas que difieren en su forma de ejecución:

- **Aplicación independiente (Stand-alone Application)**: Es análoga a la de otros lenguajes.
- **Applet**: Es una aplicación especial que se descarga desde el servidor Web, viaja a través de la Internet y se ejecuta dentro de un navegador o browser al cargar una página HTML. El applet no requiere instalación en el ordenador donde se encuentra el browser.
- **Servlet**: Es una aplicación sin interfaz gráfica que se ejecuta en un servidor de Internet.

Java permite fácilmente el desarrollo tanto de arquitecturas *cliente-servidor* como de *aplicaciones distribuidas*, consistentes en crear aplicaciones capaces de conectarse a otros ordenadores y ejecutar tareas en varios ordenadores simultáneamente, repartiendo por lo tanto el trabajo. Aunque también otros lenguajes de programación permiten crear aplicaciones de este tipo, **Java** incorpora en su propio **API** estas funcionalidades.

1.4. Entornos de desarrollo de Java

1.4.1. Java SE Development Kit (JDK)

El **JDK** es un conjunto de herramientas (programas y librerías) que permiten desarrollar (compilar, ejecutar, generar documentación, etc.) programas en lenguaje **Java**. Incorpora además el denominado **Debugger**, que permite la posibilidad de ejecutar parcialmente el programa, deteniendo la ejecución en el punto deseado y estudiando en cada momento el valor de cada una de las variables, para poder depurar

el programa. Existe también una versión reducida del **JDK**, denominada **JRE** (*Java SE Runtime Environment*) destinada únicamente a ejecutar código **Java** (no permite compilar). Aunque viene incluida en el **JDK**, también se puede bajar de forma separada. Es de libre distribución, luego se puede incluir junto con los programas a la hora de distribuir aplicaciones si el usuario final no dispone de la **Java Virtual Machine (JVM)**.



La compañía *Sun Microsystems Inc.*, creadora de **Java**, distribuye gratuitamente estas herramientas, bajo el nombre de **Java SE Development Kit (JDK)**. Existen versiones para los distintos sistemas operativos (Windows 95/98/NT, Solaris y Linux). Se pueden obtener las últimas versiones del **JDK** directamente desde la dirección <http://java.sun.com/javase/downloads/index.jsp>.

1.4.1.1. Instalación

Para comenzar a trabajar se necesita tener una instalación del **Java Development Kit (JDK)** en el **ordenador local** o en una **unidad de red** (partiendo de un programa de instalación o copiándolo desde otro ordenador). Se llamará **JAVAPATH** al path completo del directorio donde se encuentra instalado el **JDK**. Dicha instalación pone a disposición una serie de ejecutables para poder compilar y ejecutar programas en **Java**, que se encuentran en el directorio **\bin** dentro del **JAVAPATH**. Algunos de estos programas son:

appletviewer.exe	Permite la visualización de Applets.
java.exe	Ejecución de programas como aplicaciones "Stand-alone"
javac.exe	Compilación de código
javadoc.exe	Creación de documentación a partir de las clases de Java
jar.exe	Creación o extracción de ficheros *.jar (ficheros comprimidos)

La ejecución de programas en **Java** utilizando el **JDK** se realiza desde **consolas MS-DOS**. Desde una ventana de comandos de MS-DOS, sólo se pueden ejecutar los programas que se encuentren en los directorios indicados en la variable de entorno **PATH** o en el directorio activo. Para que se encuentren accesibles las herramientas de compilación (**javac.exe**) y ejecución (**java.exe**), la variable de entorno **PATH** del ordenador debe incluir el directorio **JAVAPATH\bin**.

Además, **Java** requiere de una nueva variable de entorno denominada **CLASSPATH** que determina dónde encontrar las clases o librerías de **Java** o del usuario. A partir de la versión 1.1.4 del **JDK** no es necesario indicar esta variable salvo que se desee añadir conjuntos de clases de usuario que no vengan con dicho **JDK**. La variable **CLASSPATH** puede incluir la ruta de directorios o ficheros ***.zip** o ***.jar** en los que se encuentren los ficheros ***.class**.

Una forma más general de indicar estas dos variables es crear un **fichero batch o de proceso por lotes** de MS-DOS (***.bat**) donde se indiquen los valores de estas dos variables. Cada vez que se abra una ventana de MS-DOS será necesario ejecutar este fichero ***.bat** para asignar adecuadamente estos valores. El fichero contendrá las líneas:

```
SET JAVA_HOME=C:\JAVA\jdk1.6.0_04
set PATH=.;% JAVA_HOME %\bin;%PATH%
set CLASSPATH=
```

Si no se desea tener que ejecutar este fichero cada vez que se abre una consola de MS-DOS es necesario indicar estos cambios de forma permanente. La forma de hacerlo difiere según la plataforma:

- **Windows 95/98:** Consiste en modificar el fichero Autoexec.bat situado en el directorio raíz C:\ añadiendo las líneas antes mencionadas. Una vez re-arrancado el ordenador estarán presentes en cualquier consola MS-DOS que se cree.
- **Windows NT:** Se añadirán las variables JAVA_HOME, PATH y CLASSPATH en Settings -> Control Panel -> System -> Environment -> User Variables for *NombreUsuario*.
- **Windows XP:** Se añadirán las variables JAVA_HOME, PATH y CLASSPATH en Settings -> Control Panel -> System , en la pestaña Advanced, en el botón "Environment Variables".

1.4.1.2. Documentación

Existe una documentación muy completa que acompaña a cada versión del **JDK**, aunque hay que bajarla en un fichero aparte. Informa sobre los packages, clases e interfaces, con descripciones bastante detalladas de las variables y métodos, así como de las relaciones jerárquicas. La documentación de **Java** está escrita en **HTML** y se explora con un browser como Netscape Navigator o Microsoft Internet Explorer. De ordinario, en **Java** hay que programar teniendo a la vista esta información.

Puedes conseguirla de forma gratuita descargándola desde <http://java.sun.com/docs>.

1.4.1.3. Manos a la obra

Una vez que disponemos de una instalación del **JDK**, ya podemos compilar y ejecutar programas en **Java**. Vamos a aprender en primer lugar a desarrollar **Aplicaciones Independientes** (Stand-alone Applications). Comenzaremos con una de las aplicaciones más sencillas que pueden escribirse en **Java**, una aplicación que muestra un texto en la consola, y que nos servirá para ver cómo es la estructura general de un programa en **Java**.

El primer paso es escribir el programa. Para ello utilizaremos un editor de texto cualquiera (por ejemplo el Notepad). Abrimos, pues, el editor de texto y escribimos el siguiente programa, respetando las mayúsculas y minúsculas:

```
/* Estructura general de un programa en Java */
// Otra forma de comentar sólo una línea

public class MiPrograma {
    public static void main (String args[]){
        System.out.println("Mi primer programa en Java");
    } // Fin de main()
} // Fin de la clase MiPrograma
```

Guardamos el programa en el fichero **MiPrograma.java**. Abrimos una ventana o consola de MS-DOS. Recordar que cada vez que abrimos una ventana o consola de MS-DOS debemos establecer el valor de la variable **PATH** del entorno (siempre que no la tengamos definida de manera permanente en el sistema). Para ello ya sabemos que podemos crear un fichero batch (*.bat) y ejecutarlo.

Ya podemos proceder a compilar el programa. En la línea de comandos de la ventana de MS-DOS introducimos las órdenes necesarias para situarnos en el directorio donde se encuentra nuestro fichero **MiPrograma.java** y después ejecutamos la sentencia:

```
javac MiPrograma.java
```

Si se ha compilado y no ha mostrado errores, se habrá creado un nuevo fichero **MiPrograma.class** en el directorio actual. En caso contrario, hay que corregir los errores y volver a compilar. Para ejecutar el programa, se hace mediante la siguiente sentencia:

```
java MiPrograma
```

En la consola debe aparecer: **Mi primer programa en Java**

En la figura 1.3. se muestra el proceso completo, desde la creación del programa mediante un editor de texto, hasta la ejecución del mismo.

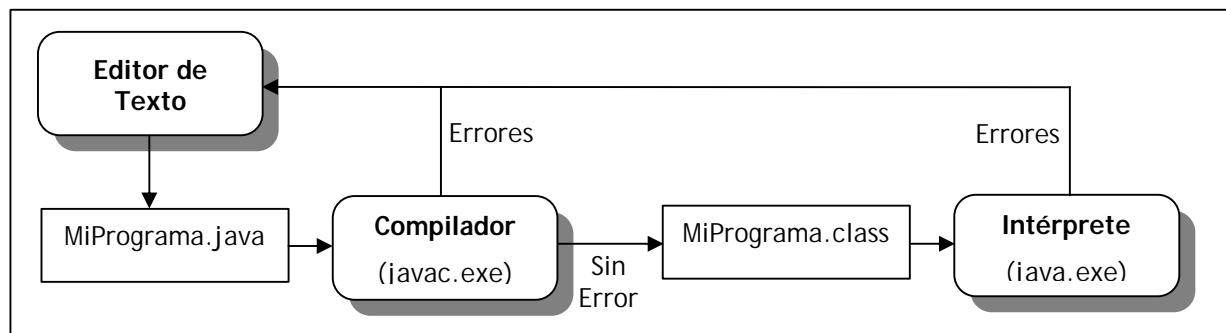


Figura 1.3. Uso del JDK.

1.4.2. Entornos IDE (*Integrated Development Environment*)

Se trata de entornos de desarrollo visual integrados. Las ventajas que tienen son las siguientes:

- Permiten desarrollar más rápidamente aplicaciones puesto que tienen incorporado el editor de texto, el compilador, el intérprete y permiten gestionar de manera eficiente proyectos o programas de cierta entidad.
- Incorporan librerías de **componentes**, los cuales se añaden al proyecto o programa.
- Facilitan enormemente el uso del **Debugger**.

Como inconvenientes se pueden señalar algunos fallos de compatibilidad entre plataformas y archivos resultantes de mayor tamaño que los basados en clases estándar.

Algunos **IDEs** conocidos son los siguientes:

- **Eclipse**, de Eclipse Foundation.
- **JBuilder**, de Borland.
- **JDeveloper**, de Oracle
- **IntelliJ IDEA**, de JetBrains.
- **NetBeans IDE** de Sun Microsystems.

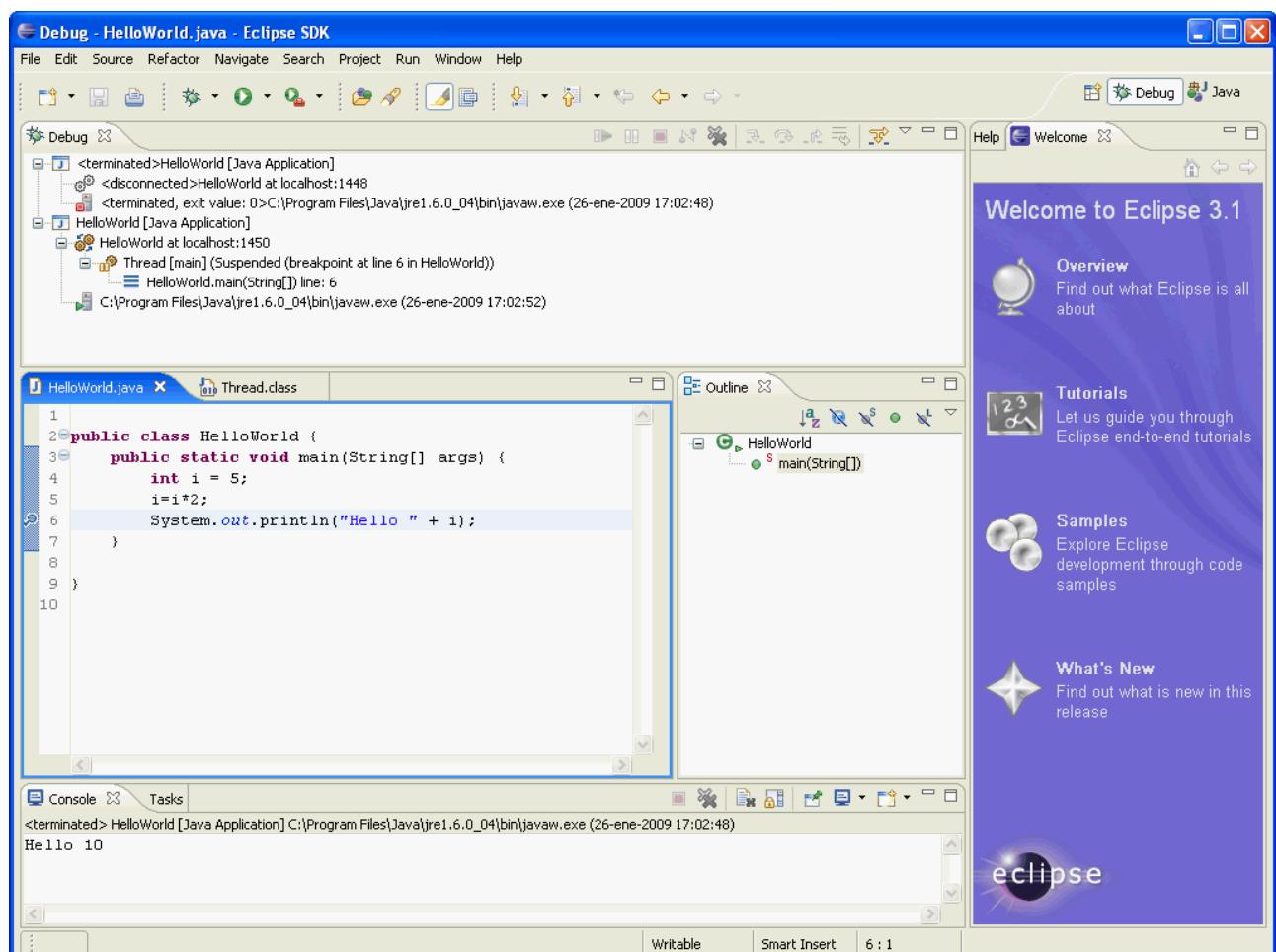


Figura 1.4. Aspecto del IDE "Eclipse"

1.5. Estructura general de un programa en Java

El ejemplo del apartado 1.4.1.3. presenta la estructura habitual de un programa realizado en **Java**. Aparece una clase que contiene el programa principal (aquel que contiene la función **main()**). Un fichero fuente (***.java**) puede contener más de una clase, pero sólo una puede ser **public**. El nombre del fichero fuente debe coincidir con el de la clase **public** (con la extensión ***.java**). Es importante que coincidan mayúsculas y minúsculas puesto que **Java** es sensible a ello, de forma que **MiClase** y **miclase** serían dos clases diferentes.

Por cada clase definida en los ficheros fuente (***.java**) el compilador genera un ***.class** para dicha clase, luego de ordinario, una aplicación de Java está constituida por varios ficheros ***.class**. Cada clase realiza unas funciones particulares, permitiendo construir las aplicaciones con gran modularidad e independencia entre clases. La aplicación se ejecuta por medio del nombre de la clase que contiene la función **main()** (sin la extensión ***.class**).

1.5.1. Concepto de Clase

Una **clase** es una agrupación de **datos** (variables o campos) y de **funciones** (métodos) que operan sobre esos datos. A estos datos y funciones pertenecientes a una clase se les denomina **variables** y **métodos** o **funciones miembro**. La programación orientada a objetos se basa en la programación de clases. Un programa se construye a partir de un conjunto de clases.

Una vez definida e implementada una clase, es posible declarar elementos de esta clase de modo similar a como se declaran las variables del lenguaje (de los tipos primitivos *int*, *double*, *String*, ...). Los elementos declarados de una clase se denominan **objetos** de la clase. De una única clase se pueden declarar o crear numerosos **objetos**. La **clase** es lo genérico: es el patrón o modelo para crear **objetos**. Cada objeto tiene sus propias copias de las variables miembro, con sus propios valores, en general distintos de los demás objetos de la clase. Las clases pueden tener variables **static**, que son propias de la clase y no de cada objeto.

Métodos de clase (**static**) son funciones que no actúan sobre objetos concretos de la clase. Son ejemplos de métodos **static** los métodos matemáticos de la clase `java.lang.Math`. **Métodos de objeto** son funciones definidas dentro de una clase que se aplican siempre a un objeto de la clase.

1.5.2. Herencia

La herencia permite que se puedan definir nuevas clases basadas en clases existentes, lo cual facilita re-utilizar código previamente desarrollado. Si una clase deriva de otra (**extends**) hereda todas sus variables y métodos. La clase derivada puede **añadir** nuevas variables y métodos y/o **redefinir** las variables y métodos heredados.

En **Java**, a diferencia de otros lenguajes orientados a objetos, una clase sólo puede derivar de una única clase, con lo cual no es posible realizar **herencia múltiple** en base a clases. Sin embargo es posible "simular" la herencia múltiple en base a las **interfaces**.

1.5.3. Concepto de Interface

Una **interface** es un conjunto de declaraciones de funciones. Si una clase implementa (**implements**) una **interface**, debe definir **todas** las funciones especificadas por la **interface**. Una **clase** puede implementar más de una **interface**, representando una forma alternativa de la herencia múltiple.

A su vez, una **interface** puede derivar de otra o incluso de varias **interfaces**, en cuyo caso incorpora todos los métodos de las **interfaces** de las que deriva.

1.5.4. Concepto de Package

Un **package** es una agrupación de clases. Existen una serie de **packages** incluidos en el lenguaje (ver jerarquía de clases que aparece en el **API de Java**).

Además el usuario puede crear sus propios **packages**. Lo habitual es juntar en **packages** las clases que estén relacionadas. Todas las clases que formen parte de un **package** deben estar en el mismo directorio.

1.5.5. La jerarquía de clases de Java (API)

Durante la generación de código en **Java**, es recomendable y casi necesario tener siempre a la vista la documentación **on-line** del **API de Java 1.1** ó **Java 1.2**. En dicha documentación es posible ver tanto la

jerarquía de clases, es decir, la relación de herencia entre clases, como la información de los distintos **packages** que componen las librerías base de **Java**.

Es importante distinguir entre lo que significa **herencia** y **package**. Un **package** es una agrupación arbitraria de clases, una forma de organizar las clases. La **herencia** sin embargo consiste en crear nuevas clases en base a otras ya existentes. Las clases incluidas en un **package** **no derivan** por lo general de una única clase.

En la documentación **on-line** se presentan ambas visiones: “**Package Index**” y “**Class Hierarchy**”, tanto en **Java 1.1** como en **Java 1.2**, con pequeñas variantes. La primera presenta la estructura del **API** de **Java** agrupada por **packages**, mientras que en la segunda aparece la jerarquía de clases. Hay que resaltar el hecho de que todas las clases en **Java** son derivadas de la clase **java.lang.Object**, por lo que heredan todos los métodos y variables de ésta.

Si se selecciona una clase en particular, la documentación muestra una descripción detallada de todos los métodos y variables de la clase. A su vez muestra su herencia completa (partiendo de la clase **java.lang.Object**).

2. Programación en Java

En este capítulo se presentan las características generales de **Java** como lenguaje de programación algorítmico. En este apartado **Java** es muy similar a **C/C++**, lenguajes en los que está inspirado. Se va a intentar ser breve, considerando que el lector ya conoce algunos otros lenguajes de programación y está familiarizado con lo que son variables, bifurcaciones, bucles, etc.

2.1. Variables

Una **variable** es un **nombre** que contiene un valor que puede cambiar a lo largo del programa. De acuerdo con el tipo de información que contienen, en **Java** hay dos tipos principales de variables:

1. Variables de **tipos primitivos**. Están definidas mediante un valor único que puede ser entero, de punto flotante, carácter o booleano. **Java** permite distinta precisión y distintos rangos de valores para estos tipos de variables (**char, byte, short, int, long, float, double, boolean**). Ejemplos de variables de tipos primitivos podrían ser: 123, 3456754, 3.1415, 12e-09, 'A', True, etc.
2. Variables **referencia**. Las variables referencia son referencias o nombres de una información más compleja: **arrays** u **objetos** de una determinada clase.

Desde el punto de vista del papel o misión en el programa, las variables pueden ser:

1. Variables **miembro** de una clase: Se definen en una clase, fuera de cualquier método; pueden ser **tipos primitivos** o **referencias**.
2. Variables **locales**: Se definen dentro de un método o más en general *dentro de cualquier bloque* entre **llaves** {}. Se crean en el interior del bloque y se destruyen al finalizar dicho bloque. Pueden ser también **tipos primitivos** o **referencias**.

2.1.1. Nombres de Variables

Los nombres de variables en **Java** se pueden crear con mucha libertad. Pueden ser cualquier conjunto de caracteres numéricos y alfanuméricos, sin algunos caracteres especiales utilizados por **Java** como operadores o separadores (.,+,*/, etc.).

Existe una serie de **palabras reservadas** las cuales tienen un significado especial para **Java** y por lo tanto no se pueden utilizar como nombres de variables. Dichas palabras son:

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

(*) son palabras reservadas, pero no se utilizan en la actual implementación del lenguaje **Java**.

2.1.2. Tipos Primitivos de Variables

Se llaman **tipos primitivos** de variables de **Java** a aquellas variables sencillas que contienen los tipos de información más habituales: valores *boolean*, *caracteres* y *valores numéricos* enteros o de punto flotante.

Java dispone de ocho tipos primitivos de variables: un tipo para almacenar valores *true* y *false* (**boolean**); un tipo para almacenar caracteres (**char**), y 6 tipos para guardar valores numéricos, cuatro tipos para enteros (**byte**, **short**, **int** y **long**) y dos para valores reales de punto flotante (**float** y **double**). Los rangos y la memoria que ocupa cada uno de estos tipos se muestran en la Tabla 2.1.

Tipo de variable	Descripción
Boolean	1 byte. Valores true y false
Char	2 bytes. Unicode. Comprende el código ASCII
Byte	1 byte. Valor entero entre -128 y 127
Short	2 bytes. Valor entero entre -32768 y 32767
Int	4 bytes. Valor entero entre -2.147.483.648 y 2.147.483.647
Long	8 bytes. Valor entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
Float	4 bytes (entre 6 y 7 cifras decimales equivalentes). De -3.402823E38 a -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Double	8 bytes (unas 15 cifras decimales equivalentes). De -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Tabla 2.1. Tipos primitivos de variables en Java.

Los tipos primitivos de **Java** tienen algunas características importantes que se resumen a continuación:

1. El tipo **boolean** no es un valor numérico: sólo admite los valores *true* o *false*. El tipo **boolean** no se identifica con el igual o distinto de cero, como en C/C++. El resultado de la expresión lógica que aparece como condición en un bucle o en una bifurcación debe ser **boolean**.
2. El tipo **char** contiene caracteres en código UNICODE (que incluye el código ASCII), y ocupan 16 bits por carácter. Comprende los caracteres de prácticamente todos los idiomas.
3. Los tipos **byte**, **short**, **int** y **long** son números enteros que pueden ser positivos o negativos, con distintos valores máximos y mínimos. A diferencia de C/C++, en **Java** no hay enteros **unsigned**.
4. Los tipos **float** y **double** son valores de punto flotante (números reales) con 6-7 y 15 cifras decimales equivalentes, respectivamente.
5. Se utiliza la palabra **void** para indicar la ausencia de un tipo de variable determinado.
6. A diferencia de C/C++, los tipos de variables en **Java** están perfectamente definidos en todas y cada una de las posibles plataformas. Por ejemplo, un **int** ocupa siempre la misma memoria y tiene el mismo rango de valores, en cualquier tipo de ordenador.
7. Existen extensiones de **Java 1.2** para aprovechar la arquitectura de los procesadores **Intel**, que permiten realizar operaciones de punto flotante con una precisión extendida de 80 bits.

2.1.3. Cómo se definen e inicializan las variables

Una variable se define especificando el **tipo** y el **nombre** de dicha variable. Estas variables pueden ser tanto de tipos **primitivos** como **referencias** a objetos de alguna clase perteneciente al **API** de **Java** o

generada por el usuario. Si no se especifica un valor en su declaración, las variables **primitivas** se inicializan a cero (salvo *boolean* y *char*, que se inicializan a *false* y '\0'). Análogamente las variables de tipo **referencia** son inicializadas por defecto a un valor especial: **null**.

Es importante distinguir entre la **referencia** a un objeto y el **objeto** mismo. Una **referencia** es una variable que indica dónde está guardado un objeto en la memoria del ordenador (a diferencia de C/C++, **Java** no permite acceder al valor de la dirección, pues en este lenguaje se han eliminado los **punteros**). Al declarar una referencia todavía no se encuentra "apuntando" a ningún objeto en particular (salvo que se cree explícitamente un nuevo objeto en la declaración), y por eso se le asigna el valor **null**. Si se desea que esta **referencia** apunte a un nuevo objeto es necesario crear el objeto utilizando el operador **new**. Este operador reserva en la memoria del ordenador espacio para ese objeto (variables y funciones). También es posible igualar la **referencia** declarada a otra referencia a un objeto existente previamente.

Un tipo particular de referencias son los **arrays** o vectores, sean éstos de variables primitivas (por ejemplo, un vector de enteros) o de objetos. En la declaración de una referencia de tipo **array** hay que incluir los **corthetes** **[]**. En los siguientes ejemplos aparece cómo crear un vector de 10 números enteros y cómo crear un vector de elementos **MyClass**. **Java** garantiza que los elementos del vector son inicializados a **null** o a cero (según el tipo de dato) en caso de no indicar otro valor.

Ejemplos de declaración e inicialización de variables:

```

int x;                                // Declaración de la variable primitiva x. Se inicializa a 0
int y = 5;                             // Declaración de la variable primitiva y. Se inicializa a 5

MyClass unaRef;                        // Declaración de una referencia a un objeto
MyClass.                                // Se inicializa a null
unaRef = new MyClass();                // La referencia "apunta" al nuevo objeto creado
                                         // Se ha utilizado el constructor por defecto
MyClass segundaRef = unaRef;          // Declaración de una referencia a un objeto
MyClass.                                // Se inicializa al mismo valor que unaRef
int [] vector;                         // Declaración de un array. Se inicializa a null
vector = new int[10];                  // Vector de 10 enteros, inicializados a 0
double [] v = {1.0, 2.65, 3.1};        // Declaración e inicialización de un vector de 3
                                         // elementos con los valores entre llaves
MyClass [] lista=new MyClass[5];       // Se crea un vector de 5 referencias a objetos
                                         // Las 5 referencias son inicializadas a null
lista[0] = unaRef;                     // Se asigna a lista[0] el mismo valor que unaRef
lista[1] = new MyClass();              // Se asigna a lista[1] la referencia al nuevo objeto
                                         // El resto (lista[2]...lista[4]) siguen con valor null

```

En el ejemplo mostrado las referencias **unaRef**, **segundaRef** y **lista[0]** actuarán sobre el mismo objeto. Es equivalente utilizar cualquiera de las referencias ya que el objeto al que se refieren es el mismo.

2.1.4. Visibilidad y vida de las variables

Se entiende por **visibilidad**, **ámbito** o **scope** de una variable, la parte de la aplicación donde dicha variable es accesible y por lo tanto puede ser utilizada en una expresión. En **Java** todas las variables deben estar incluidas en una clase. En general las variables declaradas dentro de unas llaves {}, es decir dentro de un **bloque**, son visibles y existen dentro de estas llaves. Por ejemplo las variables declaradas al principio de una función existen mientras se ejecute la función; las variables declaradas dentro de un bloque **if** no serán válidas al finalizar las sentencias correspondientes a dicho **if** y las variables miembro de una **clase** (es decir declaradas entre las llaves {} de la clase pero fuera de cualquier método) son válidas mientras existe el objeto de la clase.

Las variables miembro de una clase declaradas como **public** son accesibles a través de una **referencia** a un objeto de dicha clase utilizando el operador punto (.). Las variables miembro declaradas como **private** no son accesibles directamente desde otras clases. Las **funciones miembro** de una clase tienen acceso directo a **todas** las variables miembro de la clase sin necesidad de anteponer el nombre de

un objeto de la clase. Sin embargo las funciones miembro de una clase **B** derivada de otra **A**, tienen acceso a todas las variables miembro de **A** declaradas como **public** o **protected**, pero no a las declaradas como **private**. Una clase derivada sólo puede acceder directamente a las variables y funciones miembro de su clase base declaradas como **public** o **protected**. Otra característica del lenguaje es que es posible declarar una variable dentro de un bloque con el mismo nombre que una variable miembro, pero no con el nombre de otra variable local que ya existiera. La variable declarada dentro del bloque oculta a la variable miembro en ese bloque. Para acceder a la variable miembro oculta será preciso utilizar el operador **this**, en la forma **this.varname**.

Uno de los aspectos más importantes en la programación orientada a objetos (OOP) es la forma en la cual son creados y eliminados los objetos. En **Java** la forma de crear nuevos **objetos** es utilizando el operador **new**. Cuando se utiliza el operador **new**, la variable de tipo **referencia** guarda la posición de memoria donde está almacenado este nuevo objeto. Para cada objeto se lleva cuenta de por cuántas variables de tipo **referencia** es apuntado. La eliminación de los objetos la realiza el programa denominado **garbage collector**, quien automáticamente libera o borra la memoria ocupada por un **objeto** cuando no existe ninguna **referencia** apuntando a ese objeto. Lo anterior significa que aunque una variable de tipo referencia deje de existir, el objeto al cual apunta no es eliminado si hay otras referencias apuntando a ese mismo objeto.

2.1.5. Casos especiales: Clases **BigInteger** y **BigDecimal**

Java 1.1 incorporó dos nuevas clases destinadas a operaciones aritméticas que requieran gran precisión: **BigInteger** y **BigDecimal**. La forma de operar con objetos de estas clases difiere de las operaciones con variables primitivas. En este caso hay que realizar las operaciones utilizando métodos propios de estas clases (**add()** para la suma, **subtract()** para la resta, **divide()** para la división, etc.). Se puede consultar la ayuda sobre el package **java.math**, donde aparecen ambas clases con todos sus métodos.

Los objetos de tipo **BigInteger** son capaces de almacenar cualquier número entero sin perder información durante las operaciones. Análogamente los objetos de tipo **BigDecimal** permiten trabajar con el número de decimales deseado.

2.2. Operadores de Java

Java es un lenguaje rico en operadores, que son casi idénticos a los de C/C++. Estos operadores se describen brevemente en los apartados siguientes.

2.2.1. Operadores aritméticos

Son operadores binarios (requieren siempre dos operandos) que realizan las operaciones aritméticas habituales: **suma** (+), **resta** (-), **multiplicación** (*), **división** (/) y **resto de la división** (%).

2.2.2. Operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador de asignación por excelencia es el **operador igual** (=). La forma general de las sentencias de asignación con este operador es:

```
variable = expression;
```

Java dispone de otros operadores de asignación. Se trata de versiones abreviadas del operador (=) que realizan operaciones "acumulativas" sobre una variable. La Tabla 2.2

Operador	Utilización	Expresión equivalente
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>

Tabla 2.2. Otros operadores de asignación.

muestra estos operadores y su equivalencia con el uso del **operador igual** (=).

2.2.3. Operadores unarios

Los operadores **más** (+) y **menos** (-) unarios sirven para mantener o cambiar el signo de una variable, constante o expresión numérica. Su uso en **Java** es el estándar de estos operadores.

2.2.4. Operador instanceof

El operador **instanceof** permite saber si un objeto pertenece o no a una determinada clase. Es un operador binario cuya forma general es,

```
objectName instanceof ClassName
```

y que devuelve **true** o **false** según el objeto pertenezca o no a la clase.

2.2.5. Operador condicional ?:

Este operador, tomado de C/C++, permite realizar bifurcaciones condicionales sencillas. Su forma general es la siguiente:

```
booleanExpression ? res1 : res2
```

donde se evalúa **booleanExpression** y se devuelve **res1** si el resultado es **true** y **res2** si el resultado es **false**. Es el único operador ternario (tres argumentos) de **Java**. Como todo operador que devuelve un valor puede ser utilizado en una expresión. Por ejemplo las sentencias:

```
x=1 ; y=10; z = (x<y)?x+3:y+8;
```

asignarán a **z** el valor 4, es decir x+3.

2.2.6. Operadores incrementales

Java dispone del operador **incremento** (++) y **decremento** (--). El operador (++) incrementa en una unidad la variable a la que se aplica, mientras que (--) la reduce en una unidad. Estos operadores se pueden utilizar de dos formas:

1. *Precediendo a la variable* (por ejemplo: **++i**). En este caso primero se incrementa la variable y luego se utiliza (ya incrementada) en la expresión en la que aparece.
2. *Siguiendo a la variable* (por ejemplo: **i++**). En este caso primero se utiliza la variable en la expresión (con el valor anterior) y luego se incrementa.

En muchas ocasiones estos operadores se utilizan para incrementar una variable fuera de una expresión. En este caso ambos operadores son equivalente. Si se utilizan en una expresión más complicada, el resultado de utilizar estos operadores en una u otra de sus formas será diferente. La actualización de contadores en bucles **for** es una de las aplicaciones más frecuentes de estos operadores.

2.2.7. Operadores relacionales

Los **operadores relacionales** sirven para realizar comparaciones de igualdad, desigualdad y relación de menor o mayor. El resultado de estos operadores es siempre un valor **boolean** (**true** o **false**) según se cumpla o no la relación considerada. La Tabla 2.3 muestra los operadores relacionales de **Java**.

Estos operadores se utilizan con mucha frecuencia en las **bifurcaciones** y en los **bucles**, que se verán en próximos apartados de este capítulo.

Operador	Utilización	El resultado es true
>	op1 > op2	si op1 es mayor que op2
>=	op1 >= op2	si op1 es mayor o igual que op2
<	op1 < op2	si op1 es menor que op2
<=	op1 <= op2	si op1 es menor o igual que op2
==	op1 == op2	si op1 y op2 son iguales
!=	op1 != op2	si op1 y op2 son diferentes

Tabla 2.3. Operadores relacionales.

2.2.8. Operadores lógicos

Los operadores lógicos se utilizan para construir **expresiones lógicas**, combinando valores lógicos (**true** y/o **false**) o los resultados de los operadores **relacionales**. La Tabla 2.4 muestra los operadores lógicos de **Java**. Debe notarse que en ciertos casos el segundo operando no se evalúa porque ya no es necesario (si ambos tienen que ser **true** y el primero es **false**, ya se sabe que la condición de que ambos sean **true** no se va a cumplir). Esto puede traer resultados no deseados y por eso se han añadido los operadores **(&)** y **(|)** que garantizan que los dos operandos se evalúan siempre.

Operador	Nombre	Utilización	Resultado
&&	AND	<code>op1 && op2</code>	true si op1 y op2 son true. Si op1 es false ya no se evalúa op2
 	OR	<code>op1 op2</code>	true si op1 u op2 son true. Si op1 es true ya no se evalúa op2
!	negación	! op	true si op es false y false si op es true
&	AND	<code>op1 & op2</code>	true si op1 y op2 son true. Siempre se evalúa op2
 	OR	<code>op1 op2</code>	true si op1 u op2 son true. Siempre se evalúa op2

Tabla 2.4. Operadores lógicos.

2.2.9. Operador de concatenación de cadenas de caracteres (+)

El operador más (+) se utiliza también para concatenar cadenas de caracteres. Por ejemplo, para escribir una cantidad con un rótulo y unas unidades puede utilizarse la sentencia:

```
System.out.println("El total asciende a " + result + " unidades");
```

donde el operador de concatenación se utiliza dos veces para construir la cadena de caracteres que se desea imprimir por medio del método **println()**. La variable numérica **result** es convertida automáticamente por **Java** en cadena de caracteres para poderla concatenar. En otras ocasiones se deberá llamar explícitamente a un método para que realice esta conversión.

2.2.10. Operadores que actúan a nivel de bits

Java dispone también de un conjunto de operadores que actúan a nivel de bits. Las operaciones de bits se utilizan con frecuencia para definir señales o **flags**, esto es, variables de tipo entero en las que cada uno de sus bits indican si una opción está activada o no. La Tabla 2.5 muestra los operadores de **Java** que actúan a nivel de bits.

Operador	Utilización	Resultado
>>	<code>op1 >> op2</code>	Desplaza los bits de op1 a la derecha una distancia op2
<<	<code>op1 << op2</code>	Desplaza los bits de op1 a la izquierda una distancia op2
>>>	<code>op1 >>> op2</code>	Desplaza los bits de op1 a la derecha una distancia op2 (positiva)
&	<code>op1 & op2</code>	Operador AND a nivel de bits
 	<code>op1 op2</code>	Operador OR a nivel de bits
^	<code>op1 ^ op2</code>	Operador XOR a nivel de bits (1 si sólo uno de los operandos es 1)
~	<code>-op2</code>	Operador complemento (invierte el valor de cada bit)

Tabla 2.5. Operadores a nivel de bits.

En binario, las potencias de dos se representan con un único bit activado. Por ejemplo, los números (1, 2, 4, 8, 16, 32, 64, 128) se representan respectivamente de modo binario en la forma (00000001, 00000010, 00000100, 00001000, 00010000, 01000000, 10000000), utilizando sólo 8 bits. La suma

de estos números permite construir una variable **flags** con los bits activados que se deseen. Por ejemplo, para construir una variable **flags** que sea 00010010 bastaría hacer **flags=2+16**. Para saber si el segundo bit por la derecha está o no activado bastaría utilizar la sentencia,

```
if (flags & 2 == 2) { ... }
```

Operador	Utilización	Equivalente a
&=	op1 &= op2	op1 = op1 & op2
 =	op1 = op2	op1 = op1 op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Tabla 2.6. Operadores de asignación a nivel de bits.

La Tabla 2.6 muestra los operadores de asignación a nivel de bits.

2.2.11. Precedencia de operadores

El orden en que se realizan las operaciones es fundamental para determinar el resultado de una expresión. Por ejemplo, el resultado de $x/y*z$ depende de qué operación (la división o el producto) se realice primero. La siguiente lista muestra el orden en que se ejecutan los distintos operadores en un sentencia, de **mayor a menor** precedencia:

postfix operators	[] . (params) expr++ expr--
unary operators	++expr --expr +expr -expr ~ !
creation or cast	new (type)expr
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
conditional	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

En **Java**, todos los operadores binarios, excepto los operadores de asignación, se evalúan de **izquierda a derecha**. Los operadores de asignación se evalúan de derecha a izquierda, lo que significa que el valor de la derecha se copia sobre la variable de la izquierda.

2.3. Estructuras de programación

En este apartado se supone que el lector tiene algunos conocimientos de programación y por lo tanto no se explican en profundidad los conceptos que aparecen.

Las **estructuras de programación** o **estructuras de control** permiten tomar decisiones y realizar un proceso repetidas veces. Son los denominados **bifurcaciones** y **bucles**. En la mayoría de los lenguajes de programación, este tipo de estructuras son comunes en cuanto a *concepto*, aunque su *sintaxis* varía de un lenguaje a otro. La sintaxis de **Java** coincide prácticamente con la utilizada en C/C++, lo que hace que para un programador de C/C++ no suponga ninguna dificultad adicional.

2.3.1. Sentencias o expresiones

Una **expresión** es un conjunto variables unidos por **operadores**. Son órdenes que se le dan al computador para que realice una tarea determinada.

Una **sentencia** es una **expresión** que acaba en **punto y coma** (:). Se permite incluir varias sentencias en una línea, aunque lo habitual es utilizar una línea para cada sentencia. Por ejemplo:

```
i = 0; j = 5; x = i + j; // Línea compuesta de tres sentencias
```

2.3.2. Comentarios

Existen dos formas diferentes de introducir comentarios entre el código de **Java** (en realidad son tres, como pronto se verá). Son similares a la forma de realizar comentarios en el lenguaje C++. Los comentarios son tremadamente útiles para poder entender el código utilizado, facilitando de ese modo futuras revisiones y correcciones. Además permite que cualquier persona distinta al programador original pueda comprender el código escrito de una forma más rápida. Se recomienda acostumbrarse a comentar el código desarrollado. De esta forma se simplifica también la tarea de estudio y revisión posteriores.

Java interpreta que todo lo que aparece a la derecha de dos barras // en una línea cualquiera del código es un comentario del programador y no lo tiene en cuenta. El comentario puede empezar al comienzo de la línea o a continuación de una instrucción que debe ser ejecutada. La segunda forma de incluir comentarios consiste en escribir el texto entre los símbolos /*...*/. Este segundo método es válido para comentar más de una línea de código. Por ejemplo:

```
// Esta linea es un comentario
int a=1; // Comentario a la derecha de una sentencia
// Esta es la forma de comentar más de una linea utilizando
// las dos barras. Requiere incluir dos barras al comienzo de cada linea
/* Esta segunda forma es mucho más cómoda para comentar un número elevado
de líneas
ya que sólo requiere modificar
el comienzo y el final. */
```

En **Java** existe además una forma especial de introducir los comentarios (utilizando /*...*/ más algunos caracteres especiales) que permite generar automáticamente la documentación sobre las **clases** y **packages** desarrollados por el programador. Una vez introducidos los comentarios, el programa **javadoc.exe** (incluido en el **JDK**) genera de forma automática la información de forma similar a la presentada en la propia documentación del **JDK**. La sintaxis de estos comentarios y la forma de utilizar el programa **javadoc.exe** se puede encontrar en la información que viene con el **JDK**.

2.3.3. Bifurcaciones

Las bifurcaciones permiten ejecutar una de entre varias acciones en función del valor de una expresión lógica o relacional. Se tratan de estructuras muy importantes ya que son las encargadas de controlar el *flujo de ejecución* de un programa. Existen dos bifurcaciones diferentes: **if** y **switch**.

2.3.3.1. Bifurcación if

Esta estructura permite ejecutar un conjunto de sentencias en función del valor que tenga la expresión de comparación (se ejecuta si la expresión de comparación tiene valor **true**). Tiene la forma siguiente:

```
if (booleanExpression) {
    statements;
}
```

Las **llaves** {} sirven para agrupar en un **bloque** las sentencias que se han de ejecutar, y no son necesarias si sólo hay una sentencia dentro del **if**.

2.3.3.2. Bifurcación if else

Análoga a la anterior, de la cual es una ampliación. Las sentencias incluidas en el **else** se ejecutan en el caso de no cumplirse la expresión de comparación (**false**),

```
if (booleanExpression) {
    statements1;
} else {
    statements2;
}
```

2.3.3.3. Bifurcación if elseif else

Permite introducir más de una expresión de comparación. Si la primera condición no se cumple, se compara la segunda y así sucesivamente. En el caso de que no se cumpla ninguna de las comparaciones se ejecutan las sentencias correspondientes al **else**.

```
if (booleanExpression1) {
    statements1;
} else if (booleanExpression2) {
    statements2;
} else if (booleanExpression3) {
    statements3;
} else {
    statements4;
}
```

Véase a continuación el siguiente ejemplo:

```
int numero = 61;                                // La variable "numero" tiene dos
dígitos
if(Math.abs(numero) < 10)           // Math.abs() calcula el valor absoluto.
(false)
    System.out.println("Numero tiene 1 digito ");
else if (Math.abs(numero) < 100) // Si numero es 61, estamos en este caso
(true)
    System.out.println("Numero tiene 1 digito ");
else {                                         // Resto de los casos
    System.out.println("Numero tiene mas de 3 digitos ");
    System.out.println("Se ha ejecutado la opción por defecto ");
}
```

2.3.3.4. Sentencia switch

Se trata de una alternativa a la bifurcación **if elseif else** cuando se compara la **misma expresión** con distintos valores. Su forma general es la siguiente:

```
switch (expression) {
    case value1: statements1; break;
    case value2: statements2; break;
    case value3: statements3; break;
    case value4: statements4; break;
    case value5: statements5; break;
    case value6: statements6; break;
```

```
[default: statements7;]
}
```

Las características más relevantes de **switch** son las siguientes:

1. Cada sentencia **case** se corresponde con un único valor de **expression**. No se pueden establecer rangos o condiciones sino que se debe comparar con valores concretos. El ejemplo del Apartado 2.3.3.3 no se podría realizar utilizando **switch**.
2. Los valores no comprendidos en ninguna sentencia **case** se pueden gestionar en **default**, que es opcional.
3. En ausencia de **break**, cuando se ejecuta una sentencia **case** se ejecutan también todas las **case** que van a continuación, hasta que se llega a un **break** o hasta que se termina el **switch**.

Ejemplo:

```
char c = (char)(Math.random()*26+'a'); // Generación aleatoria de letras
minúsculas
System.out.println("La letra " + c );
switch (c) {
    case 'a': // Se compara con la letra a
    case 'e': // Se compara con la letra e
    case 'i': // Se compara con la letra i
    case 'o': // Se compara con la letra o
    case 'u': // Se compara con la letra u
        System.out.println(" Es una vocal "); break;
    default:
        System.out.println(" Es una consonante ");
}
```

2.3.4. Bucles

Un **bucle** se utiliza para realizar un proceso repetidas veces. Se denomina también **lazo** o **loop**. El código incluido entre las **llaves {}** (opcionales si el proceso repetitivo consta de una sola línea), se ejecutará mientras se cumpla unas determinadas condiciones. Hay que prestar especial atención a los bucles infinitos, hecho que ocurre cuando la condición de finalizar el bucle (**booleanExpression**) no se llega a cumplir nunca. Se trata de un fallo muy típico, habitual sobre todo entre programadores poco experimentados.

2.3.4.1. Bucle while

Las sentencias **statements** se ejecutan mientras **booleanExpression** sea **true**.

```
while (booleanExpression) {
    statements;
}
```

2.3.4.2. Bucle for

La forma general del bucle **for** es la siguiente:

```
for (initialization; booleanExpression; increment) {
    statements;
}
```

que es equivalente a utilizar **while** en la siguiente forma,

```
initialization;
while (booleanExpression) {
    statements;
    increment;
}
```

La sentencia o sentencias **initialization** se ejecuta al comienzo del **for**, e **increment** después de **statements**. La **booleanExpression** se evalúa al comienzo de cada iteración; el bucle termina cuando la

expresión de comparación toma el valor **false**. Cualquiera de las tres partes puede estar vacía. La **initialization** y el **increment** pueden tener varias expresiones separadas por comas.

Por ejemplo, el código situado a la izquierda produce la salida que aparece a la derecha:

Código:	Salida:
<pre>for(int i = 1, j = i + 10; i < 5; i++, j = 2*i) { System.out.println(" i = " + i + " j = " + j); }</pre>	<pre>i = 1 j = 11 i = 2 j = 4 i = 3 j = 6 i = 4 j = 8</pre>

2.3.4.3. Bucle do while

Es similar al bucle **while** pero con la particularidad de que el control está al final del bucle (lo que hace que el bucle se ejecute al menos una vez, independientemente de que la condición se cumpla o no). Una vez ejecutados los **statements**, se evalúa la condición: si resulta **true** se vuelven a ejecutar las sentencias incluidas en el bucle, mientras que si la condición se evalúa a **false** finaliza el bucle. Este tipo de bucles se utiliza con frecuencia para controlar la satisfacción de una determinada condición de error o de convergencia.

```
do {
    statements
} while (booleanExpression);
```

2.3.4.4. Sentencias break y continue

La sentencia **break** es válida tanto para las bifurcaciones como para los bucles. Hace que se salga inmediatamente del bucle o bloque que se está ejecutando, sin sin realizar la ejecución del resto de las sentencias.

La sentencia **continue** se utiliza en los bucles (no en bifurcaciones). Finaliza la iteración “i” que en ese momento se está ejecutando (no ejecuta el resto de sentencias que hubiera hasta el final del bucle). Vuelve al comienzo del bucle y comienza la siguiente iteración (i+1).

2.3.4.5. Sentencias break y continue con etiquetas

Las **etiquetas** permiten indicar un lugar donde continuar la ejecución de un programa después de un **break** o **continue**. El único lugar donde se pueden incluir etiquetas es **justo delante de un bloque** de código entre llaves {} (*if*, *switch*, *do...while*, *while*, *for*) y sólo se deben utilizar cuando se tiene uno o más bucles (o bloques) dentro de otro bucle y se desea salir (*break*) o continuar con la siguiente iteración (*continue*) de un bucle que no es el actual.

Por tanto, la sentencia **break labelName** finaliza el bloque que se encuentre a continuación de **labelName**. Por ejemplo, en las sentencias,

```
bucleI: // etiqueta o label
for ( int i = 0, j = 0; i < 100; i++){
    while ( true ) {
        if( (++j) > 5) { break bucleI; } // Finaliza ambos bucles
        else { break; } // Finaliza el bucle interior
    }
}
```

la expresión **break bucleI;** finaliza los dos bucles simultáneamente, mientras que la expresión **break;** sale del bucle **while** interior y seguiría con el bucle **for** en **i**. Con los valores presentados ambos bucles finalizarán con **i = 5** y **j = 6** (se invita al lector a comprobarlo).

La sentencia **continue** (siempre dentro de al menos un bucle) permite transferir el control a un bucle con nombre o etiqueta. Por ejemplo, la sentencia,

```
continue bucle1;
```

transfiere el control al bucle **for** que comienza después de la etiqueta **bucle1**: para que realice una nueva iteración, como por ejemplo:

```
bucle1:
for (int i=0; i<n; i++) {
    bucle2:
    for (int j=0; j<m; j++) {
        ...
        if (expression) continue bucle1; then continue bucle2;
        ...
    }
}
```

2.3.4.6. Sentencia return

Otra forma de salir de un bucle (y de un método) es utilizar la sentencia **return**. A diferencia de **continue** o **break**, la sentencia **return** sale también del método o función. En el caso de que la función devuelva alguna variable, este valor se deberá poner a continuación del **return** (**return value;**).

2.3.4.7. Bloque try {...} catch {...} finally {...}

Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados en esta operación. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje **Java**, una **Exception** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas **excepciones** son **fatales** y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras **excepciones**, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser **recuperables**. En este caso el programa debe dar al usuario la oportunidad de corregir el error (definiendo por ejemplo un nuevo **path** del fichero no encontrado).

Los errores se representan mediante clases derivadas de la clase **Throwable**, pero los que tiene que chequear un programador derivan de **Exception** (**java.lang.Exception** que a su vez deriva de **Throwable**). Existen algunos tipos de excepciones que **Java** obliga a tener en cuenta. Esto se hace mediante el uso de bloques **try**, **catch** y **finally**.

El código dentro del bloque **try** está "vigilado". Si se produce una situación anormal y se lanza como consecuencia una excepción, el control pasa al bloque **catch**, que se hace cargo de la situación y decide lo que hay que hacer. Se pueden incluir tantos bloques **catch** como se desee, cada uno de los cuales tratará un tipo de excepción. Finalmente, si está presente, se ejecuta el bloque **finally**, que es opcional, pero que en caso de existir se ejecuta siempre, sea cual sea el tipo de error.

En el caso en que el código de un método pueda generar una **Exception** y no se deseé incluir en dicho método la gestión del error (es decir los bucles **try/catch** correspondientes), es necesario que el método pase la **Exception** al método desde el que ha sido llamado. Esto se consigue mediante la adición de la palabra **throws** seguida del nombre de la **Exception** concreta, después de la lista de argumentos del método. A su vez el método superior deberá incluir los bloques **try/catch** o volver a pasar la **Exception**. De esta forma se puede ir pasando la **Exception** de un método a otro hasta llegar al último método del programa, el método **main()**.

En el siguiente ejemplo se presentan dos métodos que deben "controlar" una **IOException** relacionada con la lectura ficheros y una **MyException** propia. El primero de ellos (*metodo1*) realiza la gestión de las excepciones y el segundo (*metodo2*) las pasa al siguiente método.

```
void metodo1() {
    ...
    try {

```

```
... // Código que puede lanzar las excepciones IOException y
MyException
} catch (IOException e1) {// Se ocupa de IOException simplemente
dando aviso
    System.out.println(e1.getMessage());
} catch (MyException e2) {
    // Se ocupa de MyException dando un aviso y finalizando la
función
    System.out.println(e2.getMessage()); return;
} finally { // Sentencias que se ejecutarán en cualquier caso
...
}
...
} // Fin del metodo1

void metodo2() throws IOException, MyException {
...
// Código que puede lanzar las excepciones IOException y MyException
...
} // Fin del metodo2
```


3. Clases en Java

Las **clases** son el centro de la **Programación Orientada a Objetos** (OOP - *Object Oriented Programming*). Algunos de los conceptos más importantes de la POO son los siguientes:

1. **Encapsulación.** Las clases pueden ser declaradas como públicas (**public**) y como **package** (accesibles sólo para otras clases del **package**). Las variables miembro y los métodos pueden ser **public, private, protected** y **package**. De esta forma se puede controlar el acceso y evitar un uso inadecuado.
2. **Herencia.** Una clase puede derivar de otra (**extends**), y en ese caso hereda todas sus variables y métodos. Una clase derivada puede **añadir** nuevas variables y métodos y/o **redefinir** las variables y métodos heredados.
3. **Polimorfismo.** Los objetos de distintas clases pertenecientes a una misma jerarquía o que implementan una misma interface pueden tratarse de una forma general e individualizada, al mismo tiempo. Esto facilita la programación y el mantenimiento del código.

En este Capítulo se presentan las **clases** y las **interfaces** tal como están implementadas en el lenguaje **Java**.

3.1. Conceptos básicos

3.1.1. Concepto de Clase

Una clase es una agrupación de **datos** (variables o campos) y de **funciones** (métodos) que operan sobre esos datos. La definición de una clase se realiza en la siguiente forma:

```
[public] class Classname {  
    // definición de variables y métodos  
    ...  
}
```

donde la palabra **public** es opcional: si no se pone, la clase tiene la visibilidad por defecto, esto es, sólo es visible para las demás clases del **package**. Todos los métodos y variables deben ser definidos dentro del **bloque** {} de la clase.

Un **objeto** (en inglés, **instance**) es un ejemplar concreto de una clase. Las **clases** son como tipos de variables, mientras que los **objetos** son como variables concretas de un tipo determinado.

```
Classname unObjeto;  
Classname otroObjeto;
```

A continuación se enumeran algunas características importantes de las clases:

1. Todas las variables y funciones de **Java** deben pertenecer a una clase. No hay variables y funciones globales.
2. Si una clase deriva de otra (**extends**), hereda todas sus variables y métodos.
3. **Java** tiene una jerarquía de clases estándar de la que pueden derivar las clases que crean los usuarios.

4. Una clase sólo puede heredar de una única clase (en **Java** no hay herencia múltiple). Si al definir una clase no se especifica de qué clase deriva, por defecto la clase deriva de **Object**. La clase **Object** es la base de toda la jerarquía de clases de **Java**.
5. En un fichero se pueden definir varias clases, pero en un fichero no puede haber más que una clase **public**. Este fichero se debe llamar como la clase **public** que contiene con extensión ***.java**. Con algunas excepciones, lo habitual es escribir una sola clase por fichero.
6. Si una clase contenida en un fichero no es **public**, no es necesario que el fichero se llame como la clase.
7. Los métodos de una clase pueden referirse de modo global al **objeto** de esa clase al que se aplican por medio de la referencia **this**.
8. Las clases se pueden agrupar en **packages**, introduciendo una línea al comienzo del fichero (**package packageName;**). Esta agrupación en **packages** está relacionada con la jerarquía de directorios y ficheros en la que se guardan las clases.

3.1.2. Concepto de Interface

Una **interface** es un conjunto de declaraciones de funciones. Si una clase implementa (**implements**) una **interface**, debe definir **todas** las funciones especificadas por la **interface**. Las interfaces pueden definir también **variables finales** (constantes). Una **clase** puede implementar más de una **interface**, representando una alternativa a la herencia múltiple.

En algunos aspectos los nombres de las **interfaces** pueden utilizarse en lugar de las **clases**. Por ejemplo, las **interfaces** sirven para definir **referencias** a cualquier objeto de cualquiera de las clases que implementan esa **interface**. Con ese nombre o referencia, sin embargo, sólo se pueden utilizar los métodos de la interface. Éste es un aspecto importante del **polimorfismo**.

Una **interface** puede derivar de otra o incluso de varias **interfaces**, en cuyo caso incorpora las declaraciones de todos los métodos de las **interfaces** de las que deriva (a diferencia de las clases, las interfaces de **Java** sí tienen herencia múltiple).

3.2. Ejemplo de definición de una clase

A continuación se reproduce como ejemplo la clase **Circulo**.

```
// fichero Circulo.java

public class Circulo extends Geometria {
    static int numCirculos = 0;
    public static final double PI=3.14159265358979323846;
    public double x, y, r;

    public Circulo(double x, double y, double r) {
        this.x=x; this.y=y; this.r=r;
        numCirculos++;
    }

    public Circulo(double r) { this(0.0, 0.0, r); }
    public Circulo(Circulo c) { this(c.x, c.y, c.r); }
    public Circulo() { this(0.0, 0.0, 1.0); }

    public double perimetro() { return 2.0 * PI * r; }
    public double area() { return PI * r * r; }

    // método de objeto para comparar círculos
    public Circulo elMayor(Circulo c) {
```

```

        if (this.r>=c.r) return this; else return c;
    }

    // método de clase para comparar círculos
    public static Circulo elMayor(Circulo c, Circulo d) {
        if (c.r>=d.r) return c; else return d;
    }

} // fin de la clase Circulo

```

En este ejemplo se ve cómo se definen las variables miembro y los métodos (cuyos nombres se han resaltado en negrita) dentro de la clase. Dichas variables y métodos pueden ser **de objeto** o **de clase (static)**. Se puede ver también cómo el nombre del fichero coincide con el de la clase **public** con la extensión ***.java**.

3.3. Variables miembro

A diferencia de la programación algorítmica clásica, que estaba centrada en las funciones, la programación orientada a objetos está **centrada en los datos**. Una clase está constituida por unos **datos** y unos **métodos** que operan sobre esos datos.

3.3.1. Variables miembro de objeto

Cada objeto, es decir cada ejemplar concreto de la clase, tiene su propia copia de las variables miembro. Las variables miembro de una clase (también llamadas **campos**) pueden ser de **tipos primitivos** (*boolean*, *int*, *long*, *double*, ...) o referencias a **objetos** de otra clase (*composición*).

Un aspecto muy importante del correcto funcionamiento de los programas es que no haya datos sin inicializar. Por eso las variables miembro de **tipos primitivos** se inicializan siempre de modo automático, incluso antes de llamar al **constructor** (*false* para *boolean*, el carácter nulo para *char* y cero para los tipos numéricos). De todas formas, lo más adecuado es inicializarlas también en el constructor.

Las variables miembro pueden también inicializarse explícitamente en la **declaración**, como las variables locales, por medio de constantes o llamadas a métodos (esta inicialización no está permitida en C++). Por ejemplo,

```
long nDatos = 100;
```

Las variables miembro se inicializan en el mismo orden en que aparecen en el código de la clase. Esto es importante porque unas variables pueden apoyarse en otras previamente definidas.

Cada **objeto** que se crea de una clase tiene **su propia copia** de las variables miembro. Por ejemplo, cada objeto de la clase **Circulo** tiene sus propias coordenadas del centro **x** e **y**, y su propio valor del radio **r**.

Los **métodos de objeto** se aplican a un objeto concreto poniendo el nombre del objeto y luego el nombre del método, separados por un punto. A este objeto se le llama **argumento implícito**. Por ejemplo, para calcular el área de un objeto de la clase **Circulo** llamado **c1** se escribirá: **c1.area()**. Las variables miembro del argumento implícito se acceden directamente o precedidas por la palabra **this** y el operador punto.

Las variables miembro pueden ir precedidas en su declaración por uno de los modificadores de acceso: **public**, **private**, **protected** y **package** (que es el valor por defecto y puede omitirse). Junto con los modificadores de acceso de la clase (**public** y **package**), determinan qué clases y métodos van a tener permiso para utilizar la clase y sus métodos y variables miembro. En el Apartado 4.5, en la página 56, se especifican con detalle las consecuencias de estos modificadores de acceso.

Existen otros dos modificadores (no de acceso) para las variables miembro:

1. ***transient***: indica que esta variable miembro no forma parte de la ***persistencia*** (capacidad de los objetos de mantener su valor cuando termina la ejecución de un programa) de un objeto y por tanto no debe ser ***serializada*** (convertida en flujo de caracteres para poder ser almacenada en disco o en una base de datos) con el resto del objeto.
2. ***volatile***: indica que esta variable puede ser utilizada por distintas ***threads*** sincronizadas y que el compilador no debe realizar optimizaciones con esta variable.

Al nivel de estos apuntes, los modificadores ***transient*** y ***volatile*** no serán utilizados.

3.3.2. Variables miembro de clase (***static***)

Una clase puede tener variables propias de la clase y no de cada objeto. A estas variables se les llama ***variables de clase*** o variables ***static***. Las variables ***static*** se suelen utilizar para definir constantes comunes para todos los objetos de la clase (por ejemplo ***PI*** en la clase ***Circulo***) o variables que sólo tienen sentido para toda la clase (por ejemplo, un contador de objetos creados como ***numCirculos*** en la clase ***Circulo***).

Las variables de clase son lo más parecido que **Java** tiene a las ***variables globales*** de C/C++.

Las variables de clase se crean anteponiendo la palabra ***static*** a su declaración. Para llamarlas se suele utilizar el nombre de la clase (no es imprescindible, pues se puede utilizar también el nombre de cualquier objeto), porque de esta forma su sentido queda más claro. Por ejemplo, ***Circulo.numCirculos*** es una variable de clase que cuenta el número de círculos creados.

Si no se les da valor en la declaración, las variables miembro ***static*** se inicializan con los valores por defecto para los tipos primitivos (***false*** para ***boolean***, el carácter nulo para ***char*** y cero para los tipos numéricos), y con ***null*** si es una referencia.

Las variables miembro ***static*** se crean en el momento en que pueden ser necesarias: cuando se va a crear el primer objeto de la clase, en cuanto se llama a un método ***static*** o en cuanto se utiliza una variable ***static*** de dicha clase. Lo importante es que las variables miembro ***static*** se inicializan siempre antes que cualquier objeto de la clase.

3.4. variables finales

Una variable de un tipo primitivo declarada como ***final*** no puede cambiar su valor a lo largo de la ejecución del programa. Puede ser considerada como una ***constante***, y equivale a la palabra ***const*** de C/C++.

Java permite separar la ***definición*** de la ***inicialización*** de una variable ***final***. La inicialización puede hacerse más tarde, en tiempo de ejecución, llamando a métodos o en función de otros datos. La variable ***final*** así definida es ***constante*** (no puede cambiar), pero no tiene por qué tener el mismo valor en todas las ejecuciones del programa, pues depende de cómo haya sido inicializada.

Además de las variables miembro, también las variables locales y los propios argumentos de un método pueden ser declarados ***final***.

Declarar como ***final*** un objeto miembro de una clase hace ***constante*** la ***referencia***, pero no el propio objeto, que puede ser modificado a través de otra referencia. En **Java** no es posible hacer que un objeto sea constante.

3.5. Métodos (funciones miembro)

3.5.1. Métodos de objeto

Los **métodos** son funciones definidas dentro de una clase. Salvo los métodos **static** o de clase, se aplican siempre a un objeto de la clase por medio del **operador punto** (.). Dicho objeto es su **argumento implícito**. Los métodos pueden además tener otros **argumentos explícitos** que van entre paréntesis, a continuación del nombre del método.

La primera línea de la definición de un método se llama **declaración** o **header**; el código comprendido entre las **llaves** (...) es el **cuerpo** o **body** del método. Considérese el siguiente método tomado de la clase **Circulo**:

```
public Circulo elMayor(Circulo c) {           // header y comienzo del método
    if (this.r>=c.r)                         // body
        return this;                          // body
    else                                     // body
        return c;                           // body
}                                         // final del método
```

El **header** consta del cualificador de acceso (**public**, en este caso), del tipo del valor de retorno (**Circulo** en este ejemplo, **void** si no tiene), del **nombre de la función** y de una lista de **argumentos explícitos** entre paréntesis, separados por comas. Si no hay argumentos explícitos se dejan los paréntesis vacíos.

Los métodos tienen **visibilidad directa** de las variables miembro del objeto que es su **argumento implícito**, es decir, pueden acceder a ellas sin cualificarlas con un nombre de objeto y el operador punto (.). De todas formas, también se puede acceder a ellas mediante la referencia **this**, de modo discrecional (como en el ejemplo anterior con **this.r**) o si alguna variable local o argumento las oculta.

El **valor de retorno** puede ser un valor de un **tipo primitivo** o una **referencia**. En cualquier caso no puede haber más que un único valor de retorno (que puede ser un objeto o un array). Se puede devolver también una referencia a un objeto por medio de un nombre de **interface**. El objeto devuelto debe pertenecer a una clase que implemente esa interface.

Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una super-clase.

Los métodos pueden definir **variables locales**. Su visibilidad llega desde la definición al final del bloque en el que han sido definidas. No hace falta inicializar las variables locales en el punto en que se definen, pero el compilador no permite utilizarlas sin haberles dado un valor. A diferencia de las variables miembro, las variables locales no se inicializan por defecto.

Si en el **header** del método se incluye la palabra **native** (Ej: `public native void miMetodo();`) no hay que incluir el código o implementación del método. Este código deberá estar en una librería dinámica (*Dynamic Link Library* o DLL). Estas librerías son ficheros de funciones compiladas normalmente en lenguajes distintos de **Java** (C, C++, Fortran, etc.). Es la forma de poder utilizar conjuntamente funciones realizadas en otros lenguajes desde código escrito en **Java**. Este tema queda fuera del carácter fundamentalmente introductorio de este manual.

Un método también puede declararse como **synchronized** (Ej: `public synchronized double miMetodoSynch(){...}`). Estos métodos tienen la particularidad de que sobre un objeto no pueden ejecutarse simultáneamente dos métodos que estén sincronizados.

3.5.2. Métodos sobrecargados (overloaded)

Al igual que C++, **Java** permite métodos **sobrecargados (overloaded)**, es decir, métodos distintos que tienen **el mismo nombre**, pero que se diferencian por el número y/o tipo de los argumentos. El ejemplo de la clase **Círculo** del Apartado 3.2 presenta dos casos de métodos sobrecargados: los cuatro constructores y los dos métodos llamados **elMayor()**.

A la hora de llamar a un método sobrecargado, **Java** sigue unas reglas para determinar el método concreto que debe llamar:

1. Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
2. Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (por ejemplo *char* a *int*, *int* a *long*, *float* a *double*, etc.) y se llama el método correspondiente.
3. Si sólo existen métodos con argumentos de un tipo más restringido (por ejemplo, *int* en vez de *long*), el programador debe hacer un **cast** explícito en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
4. El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados, es decir con el mismo nombre, que sólo difieran en el valor de retorno.

Diferente de la **sobrecarga** de métodos es la **redefinición**. Una clase puede **redefinir (override)** un método heredado de una superclase. **Redefinir** un método es dar una nueva definición. En este caso el método debe tener exactamente los mismos argumentos en tipo y número que el método redefinido. Este tema se verá de nuevo al hablar de la **herencia**.

3.5.3. Paso de argumentos a métodos

En **Java** los argumentos de los **tipos primitivos** se pasan siempre **por valor**. El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado. La forma de modificar dentro de un método una variable de un tipo primitivo es incluirla como variable miembro en una clase y pasar como argumento una referencia a un objeto de dicha clase. Las **referencias** se pasan también **por valor**, pero a través de ellas se pueden modificar los objetos referenciados.

En **Java** no se pueden pasar métodos como argumentos a otros métodos (en C/C++ se pueden pasar punteros a función como argumentos). Lo que se puede hacer en **Java** es pasar una referencia a un objeto y dentro de la función utilizar los métodos de ese objeto.

Dentro de un método se pueden crear **variables locales** de los tipos primitivos o referencias. Estas variables locales dejan de existir al terminar la ejecución del método¹. Los argumentos formales de un método (las variables que aparecen en el **header** del método para recibir el valor de los argumentos actuales) tienen categoría de variables locales del método.

Si un método devuelve **this** (es decir, un objeto de la clase) o una referencia a otro objeto, ese objeto puede encadenarse con otra llamada a otro método de la misma o de diferente clase y así sucesivamente. En este caso aparecerán varios métodos en la misma sentencia unidos por el operador punto (.), por ejemplo,

```
String numeroComoString = "8.978";
```

¹ En **Java** no hay variables locales **static**, que en C/C++ y Visual Basic son variables locales que conservan su valor entre las distintas llamadas a un método.

```
float p = Float.valueOf(numeroComoString).floatValue();
```

donde el método ***valueOf(String)*** de la clase ***java.lang.Float*** devuelve un objeto de la clase ***Float*** sobre el que se aplica el método ***floatValue()***, que finalmente devuelve una variable primitiva de tipo ***float***. El ejemplo anterior se podía desdoblar en las siguientes sentencias:

```
String numeroComoString = "8.978";
Float f = Float.valueOf(numeroComoString);
float p = f.floatValue();
```

Obsérvese que se pueden encadenar varias llamadas a métodos por medio del operador punto (.) que, como todos los operadores de ***Java*** excepto los de asignación, se ejecuta de izquierda a derecha (ver Apartado 2.2.11, en la página 33).

3.5.4. Métodos de clase (***static***)

Análogamente, puede también haber métodos que no actúen sobre objetos concretos a través del operador punto. A estos métodos se les llama **métodos de clase** o ***static***. Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumento implícito ni pueden utilizar la referencia ***this***. Un ejemplo típico de métodos ***static*** son los métodos matemáticos de la clase ***java.lang.Math*** (***sin()***, ***cos()***, ***exp()***, ***pow()***, etc.). De ordinario el argumento de estos métodos será de un tipo primitivo y se le pasará como argumento explícito. Estos métodos no tienen sentido como métodos de objeto.

Los métodos y variables de clase se crean anteponiendo la palabra ***static***. Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase (por ejemplo, ***Math.sin(ang)***, para calcular el seno de un ángulo).

Los métodos y las variables de clase son lo más parecido que ***Java*** tiene a las funciones y variables globales de C/C++ o Visual Basic.

3.5.5. Constructores

Un punto clave de la *Programación Orientada Objetos* es el evitar información incorrecta por no haber sido correctamente inicializadas las variables. ***Java*** no permite que haya variables miembro que no estén inicializadas². Ya se ha dicho que ***Java*** inicializa siempre con **valores por defecto** las variables miembro de clases y objetos. El segundo paso en la inicialización correcta de objetos es el uso de **constructores**.

Un **constructor** es un método que se llama automáticamente cada vez que se crea un objeto de una clase. La principal misión del **constructor** es reservar memoria e inicializar las variables miembro de la clase.

Los **constructores** no tienen valor de retorno (ni siquiera ***void***) y su **nombre** es el mismo que el de la clase. Su **argumento implícito** es el objeto que se está creando.

De ordinario una clase tiene **varios constructores**, que se diferencian por el tipo y número de sus argumentos (son un ejemplo típico de métodos **sobrecargados**). Se llama **constructor por defecto** al constructor que no tiene argumentos. El programador debe proporcionar en el código valores iniciales adecuados para todas las variables miembro.

Un **constructor** de una clase puede llamar a **otro constructor previamente definido** en la misma clase por medio de la palabra ***this***. En este contexto, la palabra ***this*** sólo puede aparecer en la **primera sentencia** de un **constructor**.

² Sí puede haber variables locales de métodos sin inicializar, pero el compilador da un error si se intentan utilizar sin asignarles previamente un valor.

El **constructor** de una **sub-clase** puede llamar al constructor de su **super-clase** por medio de la palabra **super**, seguida de los argumentos apropiados entre paréntesis. De esta forma, un constructor sólo tiene que inicializar por sí mismo las variables no heredadas.

El **constructor** es tan importante que, si el programador no prepara **ningún constructor** para una clase, el **compilador** crea un **constructor por defecto**, inicializando las variables de los tipos primitivos a su valor por defecto, y los **Strings** y las demás **referencias** a objetos a **null**. Si hace falta, se llama al **constructor** de la **super-clase** para que inicialice las variables heredadas.

Al igual que los demás métodos de una clase, los **constructores** pueden tener también los modificadores de acceso **public**, **private**, **protected** y **package**. Si un **constructor** es **private**, ninguna otra clase puede crear un objeto de esa clase. En este caso, puede haber métodos **public** y **static** (**factory methods**) que llamen al **constructor** y devuelvan un objeto de esa clase.

Dentro de una clase, los **constructores** sólo pueden ser llamados por otros **constructores** o por métodos **static**. No pueden ser llamados por los **métodos de objeto** de la clase.

3.5.6. Inicializadores

Por motivos que se verán más adelante, **Java** todavía dispone de una tercera línea de actuación para evitar que haya variables sin inicializar correctamente. Son los **inicializadores**, que pueden ser **static** (para la clase) o **de objeto**.

3.5.6.1. Inicializadores static

Un **inicializador static** es un algo parecido a un método (un bloque {...} de código, sin nombre y sin argumentos, precedido por la palabra **static**) que se llama automáticamente al crear la clase (al utilizarla por primera vez). También se diferencia del **constructor** en que no es llamado para cada objeto, sino una sola vez para toda la clase.

Los tipos primitivos pueden inicializarse directamente con asignaciones en la clase o en el constructor, pero para inicializar objetos o elementos más complicados es bueno utilizar un **inicializador** (un bloque de código {...}), ya que permite gestionar **excepciones**³ con **try...catch**.

Los **inicializadores static** se crean dentro de la clase, como métodos sin nombre, sin argumentos y sin valor de retorno, con tan sólo la palabra **static** y el código entre llaves {...}. En una clase pueden definirse **varios inicializadores static**, que se llamarán en el orden en que han sido definidos.

Los **inicializadores static** se pueden utilizar para dar valor a las variables **static**. Además se suelen utilizar para llamar a **métodos nativos**, esto es, a métodos escritos por ejemplo en C/C++ (llamando a los métodos **System.load()** o **System.loadLibrary()**, que leen las librerías nativas). Por ejemplo:

```
static{
    System.loadLibrary( "MyNativeLibrary" );
}
```

3.5.6.2. Inicializadores de objeto

A partir de **Java 1.1** existen también **inicializadores de objeto**, que no llevan la palabra **static**. Se utilizan para las **clases anónimas**, que por no tener nombre no pueden tener constructor. En este caso, los inicializadores de objeto se llaman cada vez que se crea un objeto de la clase anónima.

3.5.7. Resumen del proceso de creación de un objeto

El proceso de creación de objetos de una clase es el siguiente:

³ Las **excepciones** son situaciones de error o, en general, situaciones anómalas que puede exigir ciertas actuaciones del propio programa o del usuario.

1. Al crear el primer objeto de la clase o al utilizar el primer método o variable **static** se localiza la clase y se carga en memoria.
2. Se ejecutan los **inicializadores static** (sólo una vez).
3. Cada vez que se quiere crear un **nuevo objeto**:
 - se comienza reservando la memoria necesaria
 - se da valor por defecto a las variables miembro de los tipos primitivos
 - se ejecutan los inicializadores de objeto
 - se ejecutan los constructores

3.5.8. Destrucción de objetos (liberación de memoria)

En **Java** no hay **destructores** como en C++. El sistema se ocupa automáticamente de liberar la memoria de los objetos que ya han **perdido la referencia**, esto es, objetos que ya no tienen ningún nombre que permita acceder a ellos, por ejemplo por haber llegado al final del bloque en el que habían sido definidos, porque a la **referencia** se le ha asignado el valor **null** o porque a la **referencia** se le ha asignado la dirección de otro objeto. A esta característica de **Java** se le llama **garbage collection** (recogida de basura).

En **Java** es normal que varias variables de tipo referencia apunten al mismo objeto. **Java** lleva internamente un contador de cuántas referencias hay sobre cada objeto. El objeto podrá ser borrado cuando el número de referencias sea cero. Como ya se ha dicho, una forma de hacer que un objeto quede sin referencia es cambiar ésta a **null**, haciendo por ejemplo:

```
ObjetoRef = null;
```

En **Java** no se sabe exactamente cuándo se va a activar el **garbage collector**. Si no falta memoria es posible que no se llegue a activar en ningún momento. No es pues conveniente confiar en él para la realización de otras tareas más críticas.

Se puede llamar explícitamente al **garbage collector** con el método **System.gc()**, aunque esto es considerado por el sistema sólo como una "sugerencia" a la JVM.

3.5.9. Finalizadores

Los **finalizadores** son métodos que vienen a completar la labor del **garbage collector**. Un **finalizador** es un método que se llama automáticamente cuando se va a destruir un objeto (antes de que la memoria sea liberada de modo automático por el sistema). Se utilizan para ciertas **operaciones de terminación** distintas de liberar memoria (por ejemplo: cerrar ficheros, cerrar conexiones de red, liberar memoria reservada por funciones nativas, etc.). Hay que tener en cuenta que el **garbage collector** sólo libera la memoria reservada con **new**. Si por ejemplo se ha reservado memoria con funciones nativas en C (por ejemplo, utilizando la función **malloc()**), esta memoria hay que liberarla explícitamente utilizando el método **finalize()**.

Un **finalizador** es un método de objeto (no **static**), sin valor de retorno (**void**), sin argumentos y que siempre se llama **finalize()**. Los **finalizadores** se llaman de modo automático siempre que hayan sido definidos por el programador de la clase. Para realizar su tarea correctamente, un **finalizador** debería terminar siempre llamando al **finalizador** de su **super-clase**.

Tampoco se puede saber el momento preciso en que los **finalizadores** van a ser llamados. En muchas ocasiones será conveniente que el programador realice esas operaciones de finalización de modo explícito mediante otros métodos que él mismo llame.

El método **System.runFinalization()** "sugiere" a la JVM que ejecute los **finalizadores** de los objetos pendientes (que han perdido la referencia). Parece ser que para que este método se ejecute, en Java 1.1 hay que llamar primero a **gc()** y luego a **runFinalization()**.

4. Librerías, herencia e interfaces

4.1. Packages

4.1.1. Qué es un package

Un **package** es una agrupación de clases. En la API de **Java 1.1** había 22 **packages**; en **Java 1.2** hay 59 **packages**, lo que da una idea del “crecimiento” experimentado por el lenguaje.

Además, el usuario puede crear sus propios **packages**. Para que una clase pase a formar parte de un **package** llamado **pkgName**, hay que introducir en ella la sentencia:

```
package pkgName;
```

que debe ser la primera sentencia del fichero sin contar comentarios y líneas en blanco.

Los nombres de los **packages** se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula. El nombre de un **package** puede constar de varios nombres unidos por puntos (los propios **packages** de **Java** siguen esta norma, como por ejemplo **java.awt.event**).

Todas las clases que forman parte de un **package** deben estar en el mismo directorio. Los nombres compuestos de los **packages** están relacionados con la jerarquía de directorios en que se guardan las clases. Es recomendable que los **nombres de las clases** de **Java** sean únicos en **Internet**. Es el nombre del **package** lo que permite obtener esta característica. Una forma de conseguirlo es incluir el **nombre del dominio** (quitando quizás el país), como por ejemplo en el **package** siguiente:

```
es.ceit.jgjalon.infor2.ordenar
```

Las clases de un **package** se almacenan en un directorio con el mismo nombre largo (**path**) que el **package**. Por ejemplo, la clase,

```
es.ceit.jgjalon.infor2.ordenar.QuickSort.class
```

debería estar en el directorio,

```
CLASSPATH\es\ceit\jgjalon\infor2\ordenar\QuickSort.class
```

donde CLASSPATH es una variable de entorno del PC que establece la posición absoluta de los directorios en los que hay clases de **Java** (clases del sistema o de usuario), en este caso la posición del directorio **es** en los discos locales del ordenador.

Los **packages** se utilizan con las finalidades siguientes:

1. Para agrupar clases relacionadas.
2. Para evitar conflictos de nombres (se recuerda que el dominio de nombres de **Java** es la **Internet**). En caso de conflicto de nombres entre clases importadas, el compilador obliga a cualificar en el código los nombres de dichas clases con el nombre del **package**.
3. Para ayudar en el control de la accesibilidad de clases y miembros.

4.1.2. Cómo funcionan los packages

Con la sentencia **import packname**; se puede evitar tener que utilizar nombres muy largos, al mismo tiempo que se evitan los conflictos entre nombres. Si a pesar de todo hay conflicto entre nombres de

clases, **Java** da un error y obliga a utilizar los nombres de las clases cualificados con el nombre del **package**.

El importar un **package** no hace que se carguen todas las clases del **package**: sólo se cargarán las clases **public** que se vayan a utilizar. Al importar un **package** no se importan los **sub-packages**. Éstos deben ser importados explícitamente, pues en realidad son **packages** distintos. Por ejemplo, al importar **java.awt** no se importa **java.awt.event**.

Es posible guardar en jerarquías de directorios diferentes los ficheros ***.class** y ***.java**, con objeto por ejemplo de no mostrar la situación del código fuente. Los **packages** hacen referencia a los ficheros compilados ***.class**.

En un programa de **Java**, una clase puede ser referida con su nombre completo (el nombre del **package** más el de la clase, separados por un punto). También se pueden referir con el nombre completo las variables y los métodos de las clases. Esto se puede hacer siempre de modo opcional, pero es incómodo y hace más difícil el reutilizar el código y portarlo a otras máquinas.

La sentencia **import** permite abreviar los nombres de las clases, variables y métodos, evitando el tener que escribir continuamente el nombre del **package** importado. Se importan por defecto el **package** **java.lang** y el **package** actual o por defecto (las clases del directorio actual).

Existen dos formas de utilizar **import**: para **una clase** y para **todo un package**:

```
import es.ceit.jgjalon.infor2.ordenar.QuickSort.class;
import es.ceit.jgjalon.infor2.ordenar.*;
```

que deberían estar en el directorio:

```
classpath\es\ceit\jgjalon\infor2\ordenar
```

El cómo afectan los **packages** a los permisos de acceso de una clase se estudia en el Apartado 4.5, en la página 56.

4.2. Herencia

4.2.1. Concepto de herencia

Se puede construir una clase a partir de otra mediante el mecanismo de la **herencia**. Para indicar que una clase deriva de otra se utiliza la palabra **extends**, como por ejemplo:

```
class CirculoGrafico extends Circulo {...}
```

Cuando una clase deriva de otra, hereda todas sus variables y métodos. Estas funciones y variables miembro pueden ser **redefinidas (overridden)** en la clase derivada, que puede también definir o añadir nuevas variables y métodos. En cierta forma es como si la **sub-clase** (la clase derivada) "contuviera" un objeto de la **super-clase**; en realidad lo "amplía" con nuevas variables y métodos.

Java permite múltiples niveles de herencia, pero no permite que una clase derive de varias (no es posible la herencia múltiple). Se pueden crear tantas clases derivadas de una misma clase como se quiera.

Todas las clases de **Java** creadas por el programador tienen una **super-clase**. Cuando no se indica explícitamente una **super-clase** con la palabra **extends**, la clase deriva de **java.lang.Object**, que es la clase raíz de toda la jerarquía de clases de **Java**. Como consecuencia, todas las clases tienen algunos métodos que han heredado de **Object**.

La **composición** (el que una clase contenga un objeto de otra clase como variable miembro) se diferencia de la **herencia** en que incorpora los datos del objeto miembro, pero no sus métodos o interface (si dicha variable miembro se hace **private**).

4.2.2. La clase Object

Como ya se ha dicho, la clase **Object** es la raíz de toda la jerarquía de clases de **Java**. Todas las clases de **Java** derivan de **Object**.

La clase **Object** tiene métodos interesantes para cualquier objeto que son heredados por cualquier clase. Entre ellos se pueden citar los siguientes:

1. Métodos que pueden ser redefinidos por el programador:

clone() Crea un objeto a partir de otro objeto de la misma clase. El método original heredado de **Object** lanza una *CloneNotSupportedException*. Si se desea poder clonar una clase hay que implementar la interface **Cloneable** y redefinir el método **clone()**. Este método debe hacer una copia miembro a miembro del objeto original. No debería llamar al operador **new** ni a los constructores.

equals() Indica si dos objetos son o no iguales. Devuelve **true** si son iguales, tanto si son referencias al mismo objeto como si son objetos distintos con iguales valores de las variables miembro.

toString() Devuelve un **String** que contiene una representación del objeto como cadena de caracteres, por ejemplo para imprimirla o exportarlo.

finalize() Este método ya se ha visto al hablar de los **finalizadores**.

2. Métodos que no pueden ser redefinidos (son métodos **final**):

getClass() Devuelve un objeto de la clase **Class**, al cual se le pueden aplicar métodos para determinar el nombre de la clase, su super-clase, las interfaces implementadas, etc. Se puede crear un objeto de la misma clase que otro sin saber de qué clase es.

notify(), notifyAll() y wait() Son métodos relacionados con las **threads**.

4.2.3. Redefinición de métodos heredados

Una clase puede **redefinir** (volver a definir) cualquiera de los métodos heredados de su **super-clase** que no sean **final**. El nuevo método sustituye al heredado para todos los efectos en la clase que lo ha redefinido.

Las métodos de la **super-clase** que han sido redefinidos pueden ser todavía accedidos por medio de la palabra **super** desde los métodos de la clase derivada, aunque con este sistema sólo se puede subir un nivel en la jerarquía de clases.

Los métodos redefinidos pueden **ampliar los derechos de acceso** de la **super-clase** (por ejemplo ser **public**, en vez de **protected** o **package**), pero nunca restringirlos.

Los **métodos de clase** o **static** no pueden ser redefinidos en las clases derivadas.

4.2.4. Clases y métodos abstractos

Una **clase abstracta** (**abstract**) es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general. Las clases abstractas se declaran anteponiéndoles la palabra **abstract**, como por ejemplo,

```
public abstract class Geometria { ... }
```

Una clase **abstract** puede tener métodos declarados como **abstract**, en cuyo caso no se da definición del método. Si una clase tiene algún método **abstract** es obligatorio que la clase sea **abstract**. En cualquier **sub-clase** este método deberá bien ser redefinido, bien volver a declararse como **abstract** (el método y la **sub-clase**).

Una clase **abstract** puede tener métodos que no son **abstract**. Aunque no se puedan crear objetos de esta clase, sus **sub-clases** heredarán el método completamente a punto para ser utilizado.

Como los métodos **static** no pueden ser redefinidos, un método **abstract** no puede ser **static**.

4.2.5. Constructores en clases derivadas

Ya se comentó que un **constructor** de una clase puede llamar por medio de la palabra **this** a otro **constructor** previamente definido en la misma clase. En este contexto, la palabra **this** sólo puede aparecer en la primera sentencia de un **constructor**.

De forma análoga el **constructor** de una clase derivada puede llamar al **constructor** de su **super-clase** por medio de la palabra **super()**, seguida entre paréntesis de los argumentos apropiados para uno de los constructores de la **super-clase**. De esta forma, un **constructor** sólo tiene que inicializar directamente las variables no heredadas.

La llamada al **constructor** de la **super-clase** debe ser la **primera sentencia del constructor**⁴, excepto si se llama a otro constructor de la misma clase con **this()**. Si el programador no la incluye, Java incluye automáticamente una llamada al **constructor por defecto** de la **super-clase**, **super()**. Esta llamada en cadena a los **constructores de las super-clases** llega hasta el origen de la jerarquía de clases, esto es al constructor de **Object**.

Como ya se ha dicho, si el programador no prepara un **constructor por defecto**, el compilador crea uno, inicializando las variables de los **tipos primitivos** a sus valores por defecto, y los **Strings** y demás **referencias** a objetos a **null**. Antes, incluirá una llamada al constructor de la **super-clase**.

En el proceso de finalización o de liberación de recursos (diferentes de la memoria reservada con **new**, de la que se encarga el **garbage collector**), es importante llamar a los **finalizadores** de las distintas clases, normalmente en orden inverso al de llamada de los constructores. Esto hace que el **finalizador de la sub-clase** deba realizar todas sus tareas primero y luego llamar al finalizador de la super-clase en la forma **super.finalize()**. Los métodos **finalize()** deben ser al menos **protected**, ya que el método **finalize()** de **Object** lo es, y no está permitido reducir los permisos de acceso en la herencia.

4.3. Clases y métodos finales

Recuérdese que las **variables** declaradas como **final** no pueden cambiar su valor una vez que han sido inicializadas. En este apartado se van a presentar otros dos usos de la palabra **final**.

Una **clase** declarada **final** no puede tener clases derivadas. Esto se puede hacer por motivos de **seguridad** y también por motivos de **eficiencia**, porque cuando el compilador sabe que los métodos no van a ser redefinidos puede hacer optimizaciones adicionales.

Análogamente, un **método** declarado como **final** no puede ser redefinido por una clase que derive de su propia clase.

4.4. Interfaces

4.4.1. Concepto de interface

Una **interface** es un conjunto de **declaraciones de métodos** (sin **definición**). También puede definir **constants**, que son implícitamente **public**, **static** y **final**, y deben siempre inicializarse en la declaración. Estos métodos definen un **tipo de conducta**. Todas las clases que implementan una determinada **interface** están obligadas a proporcionar una definición de los métodos de la **interface**, y en ese sentido adquieren una **conducta** o **modo de funcionamiento**.

Una **clase** puede **implementar** una o varias **interfaces**. Para indicar que una clase implementa una o más interfaces se ponen los nombres de las **interfaces**, separados por comas, detrás de la palabra

⁴ De todas formas, antes de ejecutar esa llamada ya se ha reservado la memoria necesaria para crear el objeto y se han inicializado las variables miembro (ver Apartado 3.5.7).

implements, que a su vez va siempre a la derecha del nombre de la clase o del nombre de la super-clase en el caso de herencia. Por ejemplo,

```
public class CirculoGrafico extends Circulo
    implements Dibujable, Cloneable {
    ...
}
```

¿Qué diferencia hay entre una **interface** y una **clase abstract**? Ambas tienen en común que pueden contener varias declaraciones de métodos (la clase **abstract** puede además definirlos). A pesar de esta semejanza, que hace que en algunas ocasiones se pueda sustituir una por otra, existen también algunas **diferencias importantes**:

1. Una clase no puede heredar de dos clases **abstract**, pero sí puede heredar de una clase **abstract** e implementar una **interface**, o bien implementar dos o más **interfaces**.
2. Una clase no puede heredar métodos -definidos- de una **interface**, aunque sí **constants**.
3. Las **interfaces** permiten mucha más flexibilidad para conseguir que dos clases tengan el mismo comportamiento, independientemente de su situación en la jerarquía de clases de **Java**.
4. Las **interfaces** permiten “publicar” el comportamiento de una clase desvelando un mínimo de información.
5. Las **interfaces** tienen una **jerarquía** propia, independiente y más flexible que la de las clases, ya que tienen permitida la **herencia múltiple**.
6. De cara al **polimorfismo**, las **referencias** de un tipo **interface** se pueden utilizar de modo similar a las clases **abstract**.

4.4.2. Definición de interfaces

Una **interface** se define de un modo muy similar a las clases. A modo de ejemplo se reproduce aquí la definición de la interface **Dibujable** del ejemplo del libro “Aprenda Java como si estuviera en primero”:

```
// fichero Dibujable.java

import java.awt.Graphics;

public interface Dibujable {
    public void setPosicion(double x, double y);
    public void dibujar(Graphics dw);
}
```

Cada **interface public** debe ser definida en un fichero ***.java** con el mismo nombre de la **interface**. Los **nombres de las interfaces** suelen comenzar también con **mayúscula**.

Las **interfaces** no admiten más que los modificadores de acceso **public** y **package**. Si la **interface** no es **public** no será accesible desde fuera del **package** (tendrá la accesibilidad por defecto, que es **package**). Los métodos declarados en una **interface** son siempre **public** y **abstract**, de modo implícito.

4.4.3. Herencia en interfaces

Entre las **interfaces** existe una **jerarquía** (independiente de la de las clases) que permite **herencia simple y múltiple**. Cuando una **interface** deriva de otra, incluye todas sus constantes y declaraciones de métodos.

Una **interface** puede derivar de varias **interfaces**. Para la herencia de **interfaces** se utiliza asimismo la palabra **extends**, seguida por el nombre de las **interfaces** de las que deriva, separadas por comas.

Una **interface** puede ocultar una constante definida en una **super-interface** definiendo otra constante con el mismo nombre. De la misma forma puede ocultar, re-declarándolo de nuevo, la declaración de un método heredado de una **super-interface**.

Las **interfaces** no deberían ser modificadas más que en caso de extrema necesidad. Si se modifican, por ejemplo añadiendo alguna nueva declaración de un método, las clases que hayan implementado dicha **interface** dejarán de funcionar, a menos que implementen el nuevo método.

4.4.4. Utilización de interfaces

Las **constantes** definidas en una **interface** se pueden utilizar en cualquier clase (aunque no implemente la **interface**) precediéndolas del nombre de la **interface**, como por ejemplo (suponiendo que PI hubiera sido definida en **Dibujable**):

```
area = 2.0*Dibujable.PI*r;
```

Sin embargo, en las clases que **implementan** la **interface** las constantes se pueden utilizar directamente, como si fueran constantes de la clase. A veces se crean interfaces para agrupar constantes simbólicas relacionadas (en este sentido pueden en parte suplir las variables **enum** de C/C++).

De cara al **polimorfismo**, el nombre de una **interface** se puede utilizar como un **nuevo tipo de referencia**. En este sentido, el nombre de una interface puede ser utilizado en lugar del nombre de cualquier clase que la implemente, aunque su uso estará restringido a los métodos de la **interface**. Un objeto de ese tipo puede también ser utilizado como valor de retorno o como argumento de un método.

4.5. Permisos de acceso en Java

Una de las características de la *Programación Orientada a Objetos* es la **encapsulación**, que consiste básicamente en ocultar la información que no es pertinente o necesaria para realizar una determinada tarea. Los permisos de acceso de **Java** son una de las herramientas para conseguir esta finalidad.

4.5.1. Accesibilidad de los packages

El primer tipo de accesibilidad hace referencia a la conexión física de los ordenadores y a los permisos de acceso entre ellos y en sus directorios y ficheros. En este sentido, un **package** es accesible si sus directorios y ficheros son accesibles (si están en un ordenador accesible y se tiene permiso de lectura). Además de la propia conexión física, serán accesibles aquellos **packages** que se encuentren en la variable **CLASSPATH** del sistema.

4.5.2. Accesibilidad de clases o interfaces

En principio, cualquier **clase** o **interface** de un **package** es accesible para todas las demás clases del **package**, tanto si es **public** como si no lo es. Una clase **public** es accesible para cualquier otra clase siempre que su **package** sea accesible. Recuérdese que las **clases** e **interfaces** sólo pueden ser **public** o **package** (la opción por defecto cuando no se pone ningún modificador).

4.5.3. Accesibilidad de las variables y métodos miembros de una clase:

Desde dentro de la propia clase:

1. Todos los miembros de una clase son directamente accesibles (sin cualificar con ningún nombre o cualificando con la referencia **this**) desde dentro de la propia clase. Los métodos no necesitan que las variables miembro sean pasadas como argumento.
2. Los miembros **private** de una clase sólo son accesibles para la propia clase.
3. Si el **constructor** de una clase es **private**, sólo un método **static** de la propia clase puede crear objetos.

Desde una **sub-clase**:

1. Las **sub-clases** heredan los miembros **private** de su **super-clase**, pero sólo pueden acceder a ellos a través de métodos **public**, **protected** o **package** de la super-clase.

Desde otras clases del **package**:

1. Desde una clase de un **package** se tiene acceso a todos los miembros que no sean **private** de las demás clases del **package**.

Desde otras clases fuera del **package**:

1. Los métodos y variables son accesibles si la clase es **public** y el miembro es **public**.
2. También son accesibles si la clase que accede es una **sub-clase** y el miembro es **protected**.

La Tabla 4.1 muestra un resumen de los permisos de acceso en **Java**.

Visibilidad	pu blic	protec ted	priv ate	defa ult
Desde la propia clase	Sí	Sí	Sí	Sí
Desde otra clase en el propio package	Sí	Sí	No	Sí
Desde otra clase fuera del package	Sí	No	No	No
Desde una sub-clase en el propio package	Sí	Sí	No	Sí
Desde una sub-clase fuera del propio package	Sí	Sí	No	No

Tabla 4.1. Resumen de los permisos de acceso de Java.

4.6. Transformaciones de tipo: casting

En muchas ocasiones hay que transformar una variable de un tipo a otro, por ejemplo de **int** a **double**, o de **float** a **long**. En otras ocasiones la conversión debe hacerse entre objetos de clases diferentes, aunque relacionadas mediante la herencia. En este apartado se explican brevemente estas transformaciones de tipo.

4.6.1. Conversión de tipos primitivos

La conversión entre tipos primitivos es más sencilla. En **Java** se realizan de modo automático conversiones implícitas **de un tipo a otro de más precisión**, por ejemplo de **int** a **long**, de **float** a **double**, etc. Estas conversiones se hacen al mezclar variables de distintos tipos en expresiones matemáticas o al ejecutar sentencias de asignación en las que el miembro izquierdo tiene un tipo distinto (más amplio) que el resultado de evaluar el miembro derecho.

Las conversiones de un tipo de mayor a otro de menor precisión requieren una orden explícita del programador, pues son **conversiones inseguras** que pueden dar lugar a errores (por ejemplo, para pasar a **short** un número almacenado como **int**, hay que estar seguro de que puede ser representado con el número de cifras binarias de **short**). A estas conversiones explícitas de tipo se les llama **cast**. El **cast** se hace poniendo el tipo al que se desea transformar entre paréntesis, como por ejemplo,

```
long result;
result = (long) (a/(b+c));
```

A diferencia de C/C++, en **Java** no se puede convertir un tipo numérico a **boolean**.

4.7. Polimorfismo

El **polimorfismo** tiene que ver con la relación que se establece entre la llamada a un método y el código que efectivamente se asocia con dicha llamada. A esta relación se llama **vinculación** (*binding*). La **vinculación** puede ser **temprana** (en tiempo de compilación) o **tardía** (en tiempo de ejecución). Con

funciones normales o sobrecargadas se utiliza vinculación temprana (es posible y es lo más eficiente). Con funciones redefinidas en **Java** se utiliza siempre **vinculación tardía**, excepto si el método es **final**. El **polimorfismo** es la **opción por defecto** en **Java**.

La **vinculación tardía** hace posible que, con un método declarado en una clase base (o en una interface) y redefinido en las clases derivadas (o en clases que implementan esa interface), sea **el tipo de objeto y no el tipo de la referencia** lo que determine qué definición del método se va a utilizar. El tipo del objeto al que apunta una referencia sólo puede conocerse en tiempo de ejecución, y por eso el **polimorfismo** necesita evaluación tardía.

El **polimorfismo** permite a los programadores separar las cosas que cambian de las que no cambian, y de esta manera hacer más fácil la ampliación, el mantenimiento y la reutilización de los programas.

El **polimorfismo** puede hacerse con referencias de **super-clases abstract**, **super-clases normales** e **interfaces**. Por su mayor flexibilidad y por su independencia de la jerarquía de clases estándar, las **interfaces** permiten ampliar muchísimo las posibilidades del polimorfismo.

4.7.1. Conversión de objetos

El **polimorfismo** visto previamente está basado en utilizar referencias de un tipo más "amplio" que los objetos a los que apuntan. Las ventajas del polimorfismo son evidentes, pero hay una importante limitación: el tipo de la referencia (clase abstracta, clase base o interface) limita los métodos que se pueden utilizar y las variables miembro a las que se pueden acceder. Por ejemplo, un objeto puede tener una referencia cuyo tipo sea una **interface**, aunque sólo en el caso en que su **clase o una de sus super-clases** implemente dicha **interface**. Un objeto cuya referencia es un tipo **interface** sólo puede utilizar los métodos definidos en dicha **interface**. Dicho de otro modo, ese objeto no puede utilizar las variables y los métodos propios de su clase. De esta forma las **referencias de tipo interface** definen, limitan y unifican la forma de utilizarse de objetos pertenecientes a clases muy distintas (que implementan dicha interface).

Si se desea utilizar todos los métodos y acceder a todas las variables que la clase de un objeto permite, hay que utilizar un **cast explícito**, que convierta su referencia más general en la del tipo específico del objeto. De aquí una parte importante del interés del **cast** entre objetos (más bien entre referencias, habría que decir).

Para la conversión entre objetos de distintas clases, **Java** exige que dichas clases estén relacionadas por **herencia** (una deberá ser **sub-clase** de la otra). Se realiza una **conversión implícita o automática** de una **sub-clase** a una **super-clase** siempre que se necesite, ya que el objeto de la **sub-clase** siempre tiene toda la información necesaria para ser utilizado en lugar de un objeto de la **super-clase**. No importa que la **super-clase** no sea capaz de contener toda la información de la **sub-clase**.

La conversión en sentido contrario -utilizar un objeto de una **super-clase** donde se espera encontrar uno de la **sub-clase**- debe hacerse de modo **explícito** y puede producir errores por falta de información o de métodos. Si falta información, se obtiene una **ClassCastException**.

No se puede acceder a las variables exclusivas de la sub-clase a través de una referencia de la super-clase. Sólo se pueden utilizar los métodos definidos en la super-clase, aunque la definición utilizada para dichos métodos sea la de la sub-clase.

Por ejemplo, supóngase que se crea un objeto de una **sub-clase B** y se referencia con un nombre de una **super-clase A**,

```
A a = new B();
```

en este caso el objeto creado dispone de más información de la que la referencia **a** le permite acceder (podría ser, por ejemplo, una nueva variable miembro **j** declarada en **B**). Para acceder a esta información adicional hay que hacer un **cast** explícito en la forma **(B)a**. Para imprimir esa variable **j** habría que escribir (los paréntesis son necesarios):

```
System.out.println( ((B)a).j );
```

Un **cast** de un objeto a la **super-clase** puede permitir utilizar variables -no métodos- de la **super-clase**, aunque estén redefinidos en la **sub-clase**. Considérese el siguiente ejemplo: La clase **C** deriva de **B** y **B** deriva de **A**. Las tres definen una variable **x**. En este caso, si desde el código de la sub-clase **C** se utiliza:

```
x                  // se accede a la x de C
this.x              // se accede a la x de C
super.x             // se accede a la x de B. Sólo se puede subir un nivel
((B)this).x         // se accede a la x de B
((A)this).x         // se accede a la x de A
```


5. Clases de utilidad

Programando en **Java** nunca se parte de cero: siempre se parte de la infraestructura definida por el API de **Java**, cuyos **packages** proporcionan una buena base para que el programador construya sus aplicaciones. En este Capítulo se describen algunas clases que serán de utilidad para muchos programadores.

5.1. Arrays

Los **arrays** de **Java** (vectores, matrices, hiper-matrices de más de dos dimensiones) se tratan como objetos de una clase predefinida. Los **arrays** son **objetos**, pero con algunas características propias. Los **arrays** pueden ser asignados a objetos de la clase **Object** y los métodos de **Object** pueden ser utilizados con **arrays**.

Algunas de sus características más importantes de los **arrays** son las siguientes:

1. Los **arrays** se crean con el operador **new** seguido del tipo y número de elementos.
2. Se puede acceder al número de elementos de un array con la variable miembro implícita **length** (por ejemplo, **vect.length**).
3. Se accede a los elementos de un **array** con los **corthetes []** y un **índice** que varía de 0 a **length-1**.
4. Se pueden crear **arrays** de objetos de cualquier tipo. En principio un **array** de objetos es un **array de referencias** que hay que completar llamando al operador **new**.
5. Los elementos de un **array** se inicializan al valor por defecto del tipo correspondiente (cero para valores numéricos, el carácter nulo para **char**, **false** para **boolean**, **null** para **Strings** y para referencias).
6. Como todos los objetos, los **arrays** se pasan como argumentos a los métodos **por referencia**.
7. Se pueden crear **arrays anónimos** (por ejemplo, crear un nuevo array como argumento actual en la llamada a un método).

Inicialización de **arrays**:

1. Los **arrays** se pueden inicializar con valores entre llaves {} separados por comas.
2. También los **arrays de objetos** se pueden inicializar con varias llamadas a **new** dentro de unas llaves {}.
3. Si se igualan dos referencias a un array no se copia el array, sino que se tiene un array con dos nombres, apuntando al mismo y único objeto.
4. Creación de una **referencia** a un array. Son posibles dos formas:

```
double[] x; // preferable  
double x[];
```

5. Creación del **array** con el operador **new**:

```
x = new double[100];
```

6. Las dos etapas 4 y 5 se pueden unir en una sola:

```
double[] x = new double[100];
```

A continuación se presentan algunos ejemplos de creación de arrays:

```
// crear un array de 10 enteros, que por defecto se inicializan a cero
int v[] = new int[10];
// crear arrays inicializando con determinados valores
int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
String dias[] = {"lunes", "martes", "miercoles", "jueves",
                 "viernes", "sabado", "domingo"};
// array de 5 objetos
MiClase listaObj[] = new MiClase[5]; // de momento hay 5 referencias a null
for( i = 0 ; i < 5;i++)
    listaObj[i] = new MiClase(...);
// array anónimo
obj.metodo(new String[]{"uno", "dos", "tres"});
```

5.1.1. Arrays bidimensionales

Los arrays bidimensionales de **Java** se crean de un modo muy similar al de C++ (con reserva dinámica de memoria). En **Java** una **matriz** es un **vector** de **vectores fila**, o más en concreto un vector de referencias a los vectores fila. Con este esquema, cada fila podría tener un número de elementos diferente.

Una matriz se puede crear directamente en la forma,

```
int [][] mat = new int[3][4];
```

o bien se puede crear de modo dinámico dando los siguientes pasos:

1. Crear la **referencia** indicando con un doble corchete que es una **referencia a matriz**,

```
int[][] mat;
```

2. Crear el vector de referencias a las filas,

```
mat = new int[nfilas][];
```

3. Reservar memoria para los vectores correspondientes a las filas,

```
for (int i=0; i<nfilas; i++)
    mat[i] = new int[ncols];
```

A continuación se presentan algunos ejemplos de creación de arrays bidimensionales:

```
// crear una matriz 3x3
//     se inicializan a cero
double mat[][] = new double[3][3];
int [][] b = {{1, 2, 3},
              {4, 5, 6}, // esta coma es permitida
              };
int c = new[3][]; // se crea el array de referencias a arrays
c[0] = new int[5];
c[1] = new int[4];
c[2] = new int[8];
```

En el caso de una matriz **b**, **b.length** es el número de filas y **b[0].length** es el número de columnas (de la fila 0). Por supuesto, los arrays bidimensionales pueden contener tipos primitivos de cualquier tipo u objetos de cualquier clase.

5.2. Clases String y StringBuffer

Las clases **String** y **StringBuffer** están orientadas a manejar cadenas de caracteres. La clase **String** está orientada a manejar cadenas de caracteres constantes, es decir, que no pueden cambiar. La clase **StringBuffer** permite que el programador cambie la cadena insertando, borrando, etc. La primera es más eficiente, mientras que la segunda permite más posibilidades.

Ambas clases pertenecen al package **java.lang**, y por lo tanto no hay que importarlas. Hay que indicar que el **operador de concatenación** (+) entre objetos de tipo **String** utiliza internamente objetos de la clase **StringBuffer** y el método **append()**.

Los métodos de **String** se pueden utilizar directamente sobre **literals** (cadenas entre comillas), como por ejemplo: "Hola".**length()**.

5.2.1. Métodos de la clase String

Los objetos de la clase **String** se pueden crear a partir de cadenas constantes o **literals**, definidas entre dobles comillas, como por ejemplo: "Hola". **Java** crea siempre un objeto **String** al encontrar una cadena entre comillas. A continuación se describen dos formas de crear objetos de la clase **String**,

```
String str1 = "Hola";           // el sistema más eficaz de crear Strings  
String str2 = new String("Hola"); // también se pueden crear con un constructor
```

El primero de los métodos expuestos es el más eficiente, porque como al encontrar un texto entre comillas se crea automáticamente un objeto **String**, en la práctica utilizando **new** se llama al constructor dos veces. También se pueden crear objetos de la clase **String** llamando a otros constructores de la clase, a partir de objetos **StringBuffer**, y de arrays de **bytes** o de **chars**.

La Tabla 5.1 muestra los métodos más importantes de la clase **String**.

Métodos de String	Función que realizan
String(...)	Constructores para crear Strings a partir de arrays de bytes o de caracteres (ver documentación on-line)
String(String str) y String(StringBuffer sb)	Costructores a partir de un objeto String o StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
indexOf(String, [int])	Devuelve la posición en la que aparece por primera vez un String en otro String, a partir de una posición dada (opcional)
lastIndexOf(String, [int])	Devuelve la última vez que un String aparece en otro empezando en una posición y hacia el principio
length()	Devuelve el número de caracteres de la cadena
replace(char, char)	Sustituye un carácter por otro en un String
startsWith(String)	Indica si un String comienza con otro String o no
substring(int, int)	Devuelve un String extraído de otro
toLowerCase()	Convierte en minúsculas (puede tener en cuenta el locale)
toUpperCase()	Convierte en mayúsculas (puede tener en cuenta el locale)
trim()	Elimina los espacios en blanco al comienzo y final de la cadena
valueOf()	Devuelve la representación como String de sus argumento. Admite Object, arrays de caracteres y los tipos primitivos

Tabla 5.1. Algunos métodos de la clase String.

Un punto importante a tener en cuenta es que hay métodos, tales como `System.out.println()`, que exigen que su argumento sea un objeto de la clase **String**. Si no lo es, habrá que utilizar algún método que lo convierta en **String**.

El **locale**, citado en la Tabla 5.1, es la forma que java tiene para adaptarse a las peculiaridades de los idiomas distintos del inglés: acentos, caracteres especiales, forma de escribir las fechas y las horas, unidades monetarias, etc.

5.2.2. Métodos de la clase StringBuffer

La clase **StringBuffer** se utiliza prácticamente siempre que se desee modificar una cadena de caracteres. Completa los métodos de la clase **String** ya que éstos realizan sólo operaciones sobre el texto que no conllevan un aumento o disminución del número de letras del **String**.

Recuérdese que hay muchos métodos cuyos argumentos deben ser objetos **String**, que antes de pasar esos argumentos habrá que realizar la conversión correspondiente. La Tabla 5.2 muestra los métodos más importantes de la clase **StringBuffer**.

Métodos de StringBuffer	Función que realizan
StringBuffer(), StringBuffer(int), StringBuffer(String)	Constructores
append(...)	Tiene muchas definiciones diferentes para añadir un String o una variable (int, long, double, etc.) a su objeto
capacity()	Devuelve el espacio libre del StringBuffer
charAt(int)	Devuelve el carácter en la posición especificada
getChars(int, int, char[], int)	Copia los caracteres indicados en la posición indicada de un array de caracteres
insert(int,)	Inserta un String o un valor (int, long, double, ...) en la posición especificada de un StringBuffer
length()	Devuelve el número de caracteres de la cadena
reverse()	Cambia el orden de los caracteres
setCharAt(int, char)	Cambia el carácter en la posición indicada
setLength(int)	Cambia el tamaño de un StringBuffer
toString()	Convierte en objeto de tipo String

Tabla 5.2. Algunos métodos de la clase StringBuffer.

5.3. Wrappers

Los **Wrappers** (*envoltorios*) son clases diseñadas para ser un **complemento** de los **tipos primitivos**. En efecto, los tipos primitivos son los únicos elementos de **Java** que no son objetos. Esto tiene algunas ventajas desde el punto de vista de la **eficiencia**, pero algunos inconvenientes desde el punto de vista de la **funcionalidad**. Por ejemplo, los **tipos primitivos** siempre se pasan como argumento a los métodos **por valor**, mientras que los **objetos** se pasan **por referencia**. No hay forma de modificar en un método un argumento de tipo primitivo y que esa modificación se trasmite al entorno que hizo la llamada. Una forma de conseguir esto es utilizar un **Wrapper**, esto es un objeto cuya variable miembro es el tipo primitivo que se quiere modificar. Las clases **Wrapper** también proporcionan métodos para realizar otras tareas con los tipos primitivos, tales como conversión con cadenas de caracteres en uno y otro sentido.

Existe una clase **Wrapper** para cada uno de los tipos primitivos numéricos, esto es, existen las clases **Byte**, **Short**, **Integer**, **Long**, **Float** y **Double** (obsérvese que los nombres empiezan por mayúscula, siguiendo la nomenclatura típica de **Java**). A continuación se van a ver dos de estas clases: **Double** e **Integer**. Las otras cuatro son similares y sus características pueden consultarse en la documentación online.

5.3.1. Clase Double

La clase **java.lang.Double** deriva de **Number**, que a su vez deriva de **Object**. Esta clase contiene un valor primitivo de tipo **double**. La Tabla 5.3 muestra algunos métodos y constantes predefinidas de la clase **Double**.

Métodos	Función que desempeñan
Double(double) y Double(String)	Los constructores de esta clase
doubleValue(), floatValue(), longValue(), intValue(), shortValue(), byteValue()	Métodos para obtener el valor del tipo primitivo
String toString(), Double valueOf(String)	Conversores con la clase String
isInfinite(), isNaN()	Métodos de chequear condiciones
equals(Object)	Compara con otro objeto
MAX_DOUBLE, MIN_DOUBLE, POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN, TYPE	Constantes predefinidas. TYPE es el objeto Class representando esta clase

Tabla 5.3. Métodos y constantes de la clase Double.

El Wrapper **Float** es similar al Wrapper **Double**.

5.3.2. Clase Integer

La clase **java.lang.Integer** tiene como variable miembro un valor de tipo **int**. La Tabla 5.4 muestra los métodos y constantes de la clase **Integer**.

Métodos	Función que desempeñan
Integer(int) y Integer(String)	Constructores de la clase
doubleValue(), floatValue(), longValue(), intValue(), shortValue(), byteValue()	Conversores con otros tipos primitivos
Integer decode(String), Integer parseInt(String), String toString(), Integer valueOf(String)	Conversores con String
String toBinaryString(int), String toHexString(int), String toOctalString(int)	Conversores a cadenas representando enteros en otros sistemas de numeración
Integer getInteger(String)	Determina el valor de una propiedad del sistema a partir del nombre de dicha propiedad
MAX_VALUE, MIN_VALUE, TYPE	Constantes predefinidas

Tabla 5.4. Métodos y constantes de la clase Integer.

Los Wrappers **Byte**, **Short** y **Long** son similares a **Integer**.

Los Wrappers son utilizados para convertir cadenas de caracteres (texto) en números. Esto es útil cuando se leen valores desde el teclado, desde un fichero de texto, etc. Los ejemplos siguientes muestran algunas conversiones:

```
String numDecimalString = "8.978";
float numFloat=Float.valueOf(numDecimalString).floatValue(); // numFloat = 8,979
double numDouble=Double.valueOf(numDecimalString).doubleValue(); // numDouble = 8,979
String numIntString = "1001";
int numInt=Integer.valueOf(numIntString).intValue(); // numInt = 1001
```

En el caso de que el texto no se pueda convertir directamente al tipo especificado se lanza una excepción de tipo **NumberFormatException**, por ejemplo si se intenta convertir directamente el texto "4.897" a un número entero. El proceso que habrá que seguir será convertirlo en primer lugar a un número **float** y posteriormente a número entero.

5.4. Clase Math

La clase ***java.lang.Math*** deriva de ***Object***. La clase ***Math*** proporciona métodos ***static*** para realizar las operaciones matemáticas más habituales. Proporciona además las constantes ***E*** y ***PI***, cuyo significado no requiere muchas explicaciones.

La Tabla 5.5 muestra los métodos matemáticos soportados por esta clase.

Métodos	Significado	Métodos	Significado
abs()	Valor absoluto	sin(double)	Calcula el seno
acos()	Arcocoseno	tan(double)	Calcula la tangente
asin()	Arcoseno	exp()	Calcula la función exponencial
atan()	Arcotangente entre -PI/2 y PI/2	log()	Calcula el logaritmo natural (base e)
atan2(,)	Arcotangente entre -PI y PI	max(,)	Máximo de dos argumentos
ceil()	Entero más cercano en dirección a infinito	min(,)	Mínimo de dos argumentos
floor()	Entero más cercano en dirección a -infinito	random()	Número aleatorio entre 0.0 y 1.0
round()	Entero más cercano al argumento	power(,)	Devuelve el primer argumento elevado al segundo
rint(double)	Devuelve el entero más próximo	sqrt()	Devuelve la raíz cuadrada
IEEERemainder(double, double)	Calcula el resto de la división	toDegrees(double)	Pasa de radianes a grados (Java 2)
cos(double)	Calcula el coseno	toRadians()	Pasa de grados a radianes (Java 2)

Tabla 5.5. Métodos matemáticos de la clase Math.

5.5. Colecciones

Java dispone también de clases e interfaces para trabajar con colecciones de objetos. En primer lugar se verán las clases ***Vector*** y ***Hashtable***, así como la interface ***Enumeration***. Estas clases están presentes en lenguaje desde la primera versión. Después se explicará brevemente la ***Java Collections Framework***, introducida en la versión JDK 1.2.

5.5.1. Clase Vector

La clase ***java.util.Vector*** deriva de ***Object***, implementa ***Cloneable*** (para poder sacar copias con el método ***clone()***) y ***Serializable*** (para poder ser convertida en cadena de caracteres).

Como su mismo nombre sugiere, ***Vector*** representa un ***array de objetos*** (referencias a objetos de tipo ***Object***) que puede crecer y reducirse, según el número de elementos. Además permite acceder a los elementos con un ***índice***, aunque no permite utilizar los corchetes ***[]***.

El método ***capacity()*** devuelve el tamaño o número de elementos que puede tener el vector. El método ***size()*** devuelve el número de elementos que realmente contiene, mientras que ***capacityIncrement*** es una variable que indica el salto que se dará en el tamaño cuando se necesite

crecer. La Tabla 5.6 muestra los métodos más importantes de la clase **Vector**. Puede verse que el gran número de métodos que existen proporciona una notable flexibilidad en la utilización de esta clase.

Además de **capacityIncrement**, existen otras dos variables miembro: **elementCount**, que representa el número de componentes válidos del vector, y **elementData[]** que es el array de **Objects** donde realmente se guardan los elementos del objeto **Vector** (**capacity** es el tamaño de este array). Las tres variables citadas son **protected**.

Métodos	Función que realizan
Vector(), Vector(int), Vector(int, int)	Constructores que crean un vector vacío, un vector de la capacidad indicada y un vector de la capacidad e incremento indicados
void addElement(Object obj)	Añade un objeto al final
boolean removeElement(Object obj)	Elimina el primer objeto que encuentra como su argumento y desplaza los restantes. Si no lo encuentra devuelve false
void removeAllElements()	Elimina todos los elementos
Object clone()	Devuelve una copia del vector
void copyInto(Object anArray[])	Copia un vector en un array
void trimToSize()	Ajusta el tamaño a los elementos que tiene
void setSize(int newSize)	Establece un nuevo tamaño
int capacity()	Devuelve el tamaño (capacidad) del vector
int size()	Devuelve el número de elementos
boolean isEmpty()	Devuelve true si no tiene elementos
Enumeration elements()	Devuelve una Enumeración con los elementos
boolean contains(Object elem)	Indica si contiene o no un objeto
int indexOf(Object elem, int index)	Devuelve la posición de la primera vez que aparece un objeto a partir de una posición dada
int lastIndexOf(Object elem, int index)	Devuelve la posición de la última vez que aparece un objeto a partir de una posición, hacia atrás
Object elementAt(int index)	Devuelve el objeto en una determinada posición
Object firstElement()	Devuelve el primer elemento
Object lastElement()	Devuelve el último elemento
void setElementAt(Object obj, int index)	Cambia el elemento que está en una determinada posición
void removeElementAt(int index)	Elimina el elemento que está en una determinada posición
void insertElementAt(Object obj, int index)	Inserta un elemento por delante de una determinada posición

Tabla 5.6. Métodos de la clase Vector.

5.5.2. Interface Enumeration

La interface **java.util.Enumeration** define métodos útiles para recorrer una colección de objetos. Puede haber distintas clases que implementen esta interface y todas tendrán un comportamiento similar.

La interface ***Enumeration*** declara dos métodos:

1. ***public boolean hasMoreElements()***. Indica si hay más elementos en la colección o si se ha llegado ya al final.
2. ***public Object nextElement()***. Devuelve el siguiente objeto de la colección. Lanza una *NoSuchElementException* si se llama y ya no hay más elementos.

Ejemplo: Para imprimir los elementos de un vector ***vec*** se pueden utilizar las siguientes sentencias:

```
for (Enumeration e = vec.elements(); e.hasMoreElements(); ) {
    System.out.println(e.nextElement());
}
```

donde, como puede verse en la Tabla 5.6, el método ***elements()*** devuelve precisamente una referencia de tipo ***Enumeration***. Con los métodos ***hasMoreElements()*** y ***nextElement()*** y un bucle ***for*** se pueden ir imprimiendo los distintos elementos del objeto ***Vector***.

5.5.3. Clase ***Hashtable***

La clase ***java.util.Hashtable*** extiende ***Dictionary (abstract)*** e implementa ***Cloneable*** y ***Serializable***. Una ***hash table*** es una tabla que relaciona una ***clave*** con un ***valor***. Cualquier objeto distinto de ***null*** puede ser tanto ***clave*** como ***valor***.

La clase a la que pertenecen las ***claves*** debe implementar los métodos ***hashCode()*** y ***equals()***, con objeto de hacer búsquedas y comparaciones. El método ***hashCode()*** devuelve un entero único y distinto para cada clave, que es siempre el mismo en una ejecución del programa pero que puede cambiar de una ejecución a otra. Además, para dos claves que resultan iguales según el método ***equals()***, el método ***hashCode()*** devuelve el mismo entero. Las ***hash tables*** están diseñadas para mantener una colección de pares ***clave/valor***, permitiendo insertar y realizar búsquedas de un modo muy eficiente.

Cada objeto de ***Hashtable*** tiene dos variables: ***capacity*** y ***load factor*** (entre 0.0 y 1.0). Cuando el número de elementos excede el producto de estas variables, la ***Hashtable*** crece llamando al método ***rehash()***. Un ***load factor*** más grande apura más la memoria, pero será menos eficiente en las búsquedas. Es conveniente partir de una ***Hashtable*** suficientemente grande para no estar ampliando continuamente.

Ejemplo de definición de ***Hashtable***:

```
Hashtable numeros = new Hashtable();
numbers.put("uno", new Integer(1));
numbers.put("dos", new Integer(2));
numbers.put("tres", new Integer(3));
```

donde se ha hecho uso del método ***put()***. La Tabla 5.7 muestra los métodos de la clase ***Hashtable***.

Métodos	Función que realizan
Hashtable(), Hashtable(int nElements), Hashtable(int nElements, float loadFactor)	Constructores
int size()	Devuelve el tamaño de la tabla
boolean isEmpty()	Indica si la tabla está vacía
Enumeration keys()	Devuelve una Enumeration con las claves
Enumeration elements()	Devuelve una Enumeration con los valores
boolean contains(Object value)	Indica si hay alguna clave que se corresponde con el valor
boolean containsKey(Object key)	Indica si existe esa clave en la tabla
Object get(Object key)	Devuelve un valor dado la clave
void rehash()	Amplía la capacidad de la tabla
Object put(Object key, Object value)	Establece una relación clave-valor
Object remove(Object key)	Elimina un valor por la clave
void clear()	Limpia la tabla
Object clone()	Hace una copia de la tabla
String toString()	Devuelve un string representando la tabla

Tabla 5.7. Métodos de la clase Hashtable.

5.5.4. El Collections Framework de Java 1.2

En la versión 1.2 del JDK se introdujo el **Java Framework Collections** o “estructura de colecciones de Java” (en adelante JCF). Se trata de un conjunto de clases e interfaces que mejoran notablemente las capacidades del lenguaje respecto a estructuras de datos. Además, constituyen un excelente ejemplo de aplicación de los conceptos propios de la *programación orientada a objetos*. Dada la amplitud de **Java** en éste y en otros aspectos se va a optar por insistir en la descripción general, dejando al lector la tarea de buscar las características concretas de los distintos métodos en la documentación de **Java**. En este apartado se va a utilizar una forma -más breve que las tablas utilizadas en otros apartados- de informar sobre los métodos disponibles en una clase o interface.

La Figura 5.1 muestra la jerarquía de interfaces de la **Java Collection Framework** (JCF). En letra cursiva se indican las clases que implementan las correspondientes interfaces. Por ejemplo, hay dos clases que implementan la interface **Map**: **HashMap** y **Hashtable**.

Las clases vistas en los apartados anteriores son clases “históricas”, es decir, clases que existían antes de la versión JDK 1.2. Dichas clases se denotan en la Figura 5.1 con la letra “h” entre paréntesis. Aunque dichas clases se han mantenido por motivos de compatibilidad, sus métodos no siguen las reglas del diseño general del JCF; en la medida de lo posible se recomienda utilizar las nuevas clases.

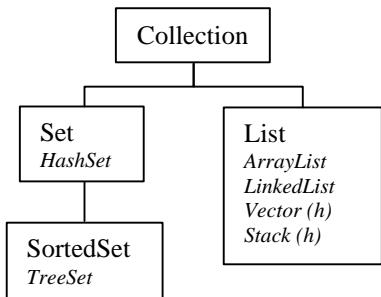


Figura 5.1. Interfaces de la Collection Framework.

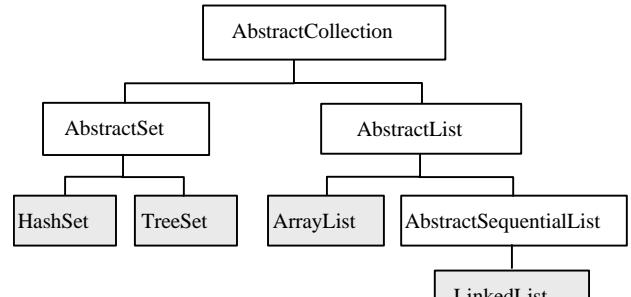


Figura 5.2. Jerarquía de clases de la Collection Framework.

En el diseño de la JCF las **interfaces** son muy importantes porque son ellas las que determinan las capacidades de las clases que las implementan. Dos clases que implementan la misma interface se pueden utilizar exactamente de la misma forma. Por ejemplo, las clases **ArrayList** y **LinkedList** disponen exactamente de los mismos métodos y se pueden utilizar de la misma forma. La diferencia está en la implementación: mientras que **ArrayList** almacena los objetos en un array, la clase **LinkedList** los almacena en una lista vinculada. La primera será más eficiente para acceder a un elemento arbitrario, mientras que la segunda será más flexible si se desea borrar e insertar elementos.

La Figura 5.2 muestra la jerarquía de clases de la JCF. En este caso, la jerarquía de clases es menos importante desde el punto de vista del usuario que la jerarquía de interfaces. En dicha figura se muestran con fondo blanco las clases abstractas, y con fondo gris claro las clases de las que se pueden crear objetos.

Las clases **Collections** y **Arrays** son un poco especiales: no son **abstract**, pero no tienen constructores públicos con los que se puedan crear objetos. Fundamentalmente contienen métodos **static** para realizar ciertas operaciones de utilidad: ordenar, buscar, introducir ciertas características en objetos de otras clases, etc.

5.5.4.1. Elementos del Java Collections Framework

Interfaces de la JCF: Constituyen el elemento central de la JCF.

- **Collection:** define métodos para tratar una colección genérica de elementos
- **Set:** colección que no admite elementos repetidos
- **SortedSet:** set cuyos elementos se mantienen ordenados según el criterio establecido
- **List:** admite elementos repetidos y mantiene un orden inicial
- **Map:** conjunto de pares clave/valor, sin repetición de claves
- **SortedMap:** map cuyos elementos se mantienen ordenados según el criterio establecido

Interfaces de soporte:

- **Iterator:** sustituye a la interface **Enumeration**. Dispone de métodos para recorrer una colección y para borrar elementos.
- **ListIterator:** deriva de **Iterator** y permite recorrer lists en ambos sentidos.

- **Comparable**: declara el método `compareTo()` que permite ordenar las distintas colecciones según un orden natural (`String`, `Date`, `Integer`, `Double`, ...).
- **Comparator**: declara el método `compare()` y se utiliza en lugar de **Comparable** cuando se desea ordenar objetos no estándar o sustituir a dicha interface.

Clases de propósito general: Son las implementaciones de las interfaces de la JFC.

- **HashSet**: Interface `Set` implementada mediante una hash table.
- **TreeSet**: Interface `SortedSet` implementada mediante un árbol binario ordenado.
- **ArrayList**: Interface `List` implementada mediante un array.
- **LinkedList**: Interface `List` implementada mediante una lista vinculada.
- **HashMap**: Interface `Map` implementada mediante una hash table.
- **WeakHashMap**: Interface `Map` implementada de modo que la memoria de los pares clave/valor pueda ser liberada cuando las claves no tengan referencia desde el exterior de la `WeakHashMap`.
- **TreeMap**: Interface `SortedMap` implementada mediante un árbol binario

Clases Wrapper: Colecciones con características adicionales, como no poder ser modificadas o estar sincronizadas. No se accede a ellas mediante constructores, sino mediante métodos "factory" de la clase `Collections`.

Clases de utilidad: Son mini-implementaciones que permiten obtener sets especializados, como por ejemplo `sets` constantes de un sólo elemento (`singleton`) o `lists` con `n` copias del mismo elemento (`nCopies`). Definen las constantes `EMPTY_SET` y `MPTY_LIST`. Se accede a través de la clase `Collections`.

Clases históricas: Son las clases `Vector` y `Hashtable` presentes desde las primeras versiones de `Java`. En las versiones actuales, implementan respectivamente las interfaces `List` y `Map`, aunque conservan también los métodos anteriores.

Clases abstractas: Son las clases abstract de la Figura 5.2. Tienen total o parcialmente implementados los métodos de la interface correspondiente. Sirven para que los usuarios deriven de ellas sus propias clases con un mínimo de esfuerzo de programación.

Algoritmos: La clase `Collections` dispone de métodos `static` para ordenar, desordenar, invertir orden, realizar búsquedas, llenar, copiar, hallar el mínimo y hallar el máximo.

Clase Arrays: Es una clase de utilidad introducida en el JDK 1.2 que contiene métodos `static` para ordenar, llenar, realizar búsquedas y comparar los arrays clásicos del lenguaje. Permite también ver los `arrays` como `lists`.

Después de esta visión general de la *Java Collections Framework*, se verán algunos detalles de las clases e interfaces más importantes.

5.5.4.2. Interface Collection

La interface `Collection` es implementada por los `conjuntos` (sets) y las `listas` (lists). Esta interface declara una serie de métodos generales utilizables con `Sets` y `Lists`. La declaración o header de dichos métodos se puede ver ejecutando el comando > `javap java.util.Collection` en una ventana de MS-DOS. El resultado se muestra a continuación:

```
public interface java.util.Collection {
    public abstract boolean add(java.lang.Object);                                // opcional
    public abstract boolean addAll(java.util.Collection);                          // opcional
    public abstract void clear();                                                 // opcional
    public abstract boolean contains(java.lang.Object);
    public abstract boolean containsAll(java.util.Collection);
    public abstract boolean equals(java.lang.Object);
    public abstract int hashCode();
    public abstract boolean isEmpty();
    public abstract java.util.Iterator iterator();
    public abstract boolean remove(java.lang.Object);                            // opcional
```

```

        public abstract boolean removeAll(java.util.Collection);      // opcional
        public abstract boolean retainAll(java.util.Collection);      // opcional
        public abstract int size();
        public abstract java.lang.Object toArray();
        public abstract java.lang.Object toArray(java.lang.Object[][]);
    }

```

A partir del nombre, de los argumentos y del valor de retorno, la mayor parte de estos métodos resultan autoexplicativos. A continuación se introducen algunos comentarios sobre los aspectos que pueden resultar más novedosos de estos métodos. Los detalles se pueden consultar en la documentación de **Java**.

Los métodos indicados como “// opcional” (estos caracteres han sido introducidos por los autores de este manual) no están disponibles en algunas implementaciones, como por ejemplo en las clases que no permiten modificar sus objetos. Por supuesto, dichos métodos deben ser definidos, pero lo que hacen al ser llamados es lanzar una **UnsupportedOperationException**.

El método **add()** trata de añadir un objeto a una colección, pero puede que no lo consiga si la colección es un **set** que ya tiene ese elemento. Devuelve **true** si el método ha llegado a modificar la colección. Lo mismo sucede con **addAll()**. El método **remove()** elimina un único elemento (si lo encuentra); devuelve **true** si la colección ha sido modificada.

El método **iterator()** devuelve una referencia **Iterator** que permite recorrer una colección con los métodos **next()** y **hasNext()**. Permite también borrar el elemento actual con **remove()**.

Los dos métodos **toArray()** permiten convertir una colección en un array.

5.5.4.3. Interfaces Iterator y ListIterator

La interface **Iterator** sustituye a **Enumeration**, utilizada en versiones anteriores del JDK. Dispone de los métodos siguientes:

```

Compiled from Iterator.java
public interface java.util.Iterator {
    public abstract boolean hasNext();
    public abstract java.lang.Object next();
    public abstract void remove();
}

```

El método **remove()** permite borrar el último elemento accedido con **next()**. Es la única forma segura de eliminar un elemento mientras se está recorriendo una colección.

Los métodos de la interface **ListIterator** son los siguientes:

```

Compiled from ListIterator.java
public interface java.util.ListIterator extends java.util.Iterator {
    public abstract void add(java.lang.Object);
    public abstract boolean hasNext();
    public abstract boolean hasPrevious();
    public abstract java.lang.Object next();
    public abstract int nextIndex();
    public abstract java.lang.Object previous();
    public abstract int previousIndex();
    public abstract void remove();
    public abstract void set(java.lang.Object);
}

```

La interface **ListIterator** permite recorrer una lista en ambas direcciones, y hacer algunas modificaciones mientras se recorre. Los elementos se numeran desde 0 a $n-1$, pero los valores válidos para el índice son de 0 a n . Puede suponerse que el índice i está en la frontera entre los elementos $i-1$ e i : en ese caso **previousIndex()** devolvería $i-1$ y **nextIndex()** devolvería i . Si el índice es 0, **previousIndex()** devuelve -1 y si el índice es n **nextIndex()** devuelve el resultado de **size()**.

5.5.4.4. Interfaces Comparable y Comparator

Estas interfaces están orientadas a mantener ordenadas las **listas**, y también los **sets** y **maps** que deben mantener un orden. Para ello se dispone de las interfaces **java.lang.Comparable** y **java.util.Comparator** (obsérvese que pertenecen a packages diferentes).

La interface **Comparable** declara el método **compareTo()** de la siguiente forma:

```
public int compareTo(Object obj)
```

que compara su argumento implícito con el que se le pasa por ventana. Este método devuelve un entero **negativo**, **cero** o **positivo** según el argumento implícito (**this**) sea **anterior**, **igual** o **posterior** al objeto **obj**. Las listas de objetos de clases que implementan esta interface tienen un **orden natural**. En **Java 1.2** esta interface está implementada -entre otras- por las clases **String**, **Character**, **Date**, **File**, **BigDecimal**, **BigInteger**, **Byte**, **Short**, **Integer**, **Long**, **Float** y **Double**. Téngase en cuenta que la implementación estándar de estas clases no asegura un orden alfabético correcto con mayúsculas y minúsculas, y tampoco en idiomas distintos del inglés.

Si se redefine, el método **compareTo()** debe ser programado con cuidado: es muy conveniente que sea coherente con el método **equals()** y que cumpla la **propiedad transitiva**. Para más información, consultar la documentación del JDK 1.2.

Las **listas** y los **arrays** cuyos elementos implementan **Comparable** pueden ser ordenadas con los métodos static **Collections.sort()** y **Arrays.sort()**.

La interface **Comparator** permite ordenar listas y colecciones cuyos objetos pertenecen a clases de tipo cualquiera. Esta interface permitiría por ejemplo ordenar figuras geométricas planas por el área o el perímetro. Su papel es similar al de la interface **Comparable**, pero el usuario debe siempre proporcionar una implementación de esta clase. Sus dos métodos se declaran en la forma:

```
public int compare(Object o1, Object o2)
public boolean equals(Object obj)
```

El objetivo del método **equals()** es comparar **Comparators**.

El método **compare()** devuelve un entero **negativo**, **cero** o **positivo** según su primer argumento sea **anterior**, **igual** o **posterior** al segundo. Los objetos que implementan **Comparator** pueden pasarse como argumentos al método **Collections.sort()** o a algunos **constructores** de las clases **TreeSet** y **TreeMap**, con la idea de que las mantengan ordenadas de acuerdo con dicho **Comparator**. Es muy importante que **compare()** sea compatible con el método **equals()** de los objetos que hay que mantener ordenados. Su implementación debe cumplir unas condiciones similares a las de **compareTo()**.

Java 1.2 dispone de clases capaces de ordenar cadenas de texto en diferentes lenguajes. Para ello se puede consultar la documentación sobre las clases **CollationKey**, **Collator** y sus clases derivadas, en el package **java.text**.

5.5.4.5. Sets y SortedSets

La interface **Set** sirve para acceder a una colección sin elementos repetidos. La colección puede estar o no ordenada (con un orden natural o definido por el usuario, se entiende). La interface **Set** no declara ningún método adicional a los de **Collection**.

Como un **Set** no admite elementos repetidos es importante saber cuándo dos objetos son considerados iguales (por ejemplo, el usuario puede o no deseiar que las palabras **Mesa** y **mesa** sean consideradas iguales). Para ello se dispone de los métodos **equals()** y **hashcode()**, que el usuario puede redefinir si lo desea.

Utilizando los métodos de **Collection**, los **Sets** permiten realizar operaciones algebraicas de **unión**, **intersección** y **diferencia**. Por ejemplo, **s1.containsAll(s2)** permite saber si **s2** está contenido en **s1**:

s1.addAll(s2) permite convertir **s1** en la unión de los dos conjuntos; **s1.retainAll(s2)** permite convertir **s1** en la intersección de **s1** y **s2**; finalmente, **s1.removeAll(s2)** convierte **s1** en la diferencia entre **s1** y **s2**.

La interface **SortedSet** extiende la interface **Set** y añade los siguientes métodos:

```
Compiled from SortedSet.java
public interface java.util.SortedSet extends java.util.Set {
    public abstract java.util.Comparator comparator();
    public abstract java.lang.Object first();
    public abstract java.util.SortedSet headSet(java.lang.Object);
    public abstract java.lang.Object last();
    public abstract java.util.SortedSet subSet(java.lang.Object, java.lang.Object);
    public abstract java.util.SortedSet tailSet(java.lang.Object);
}
```

que están orientados a trabajar con el “orden”. El método **comparator()** permite obtener el objeto pasado al constructor para establecer el orden. Si se ha utilizado el orden natural definido por la interface **Comparable**, este método devuelve **null**. Los métodos **first()** y **last()** devuelven el primer y último elemento del conjunto. Los métodos **headSet()**, **subSet()** y **tailSet()** sirven para obtener sub-conjuntos al principio, en medio y al final del conjunto original (los dos primeros no incluyen el límite superior especificado).

Existen dos implementaciones de conjuntos: la clase **HashSet** implementa la interface **Set**, mientras que la clase **TreeSet** implementa **SortedSet**. La primera está basada en una hash table y la segunda en un **TreeMap**.

Los elementos de un **HashSet** no mantienen el orden natural, ni el orden de introducción. Los elementos de un **TreeSet** mantienen el orden natural o el especificado por la interface **Comparator**. Ambas clases definen constructores que admiten como argumento un objeto **Collection**, lo cual permite convertir un **HashSet** en un **TreeSet** y viceversa.

5.5.4.6. Listas

La interface **List** define métodos para operar con colecciones ordenadas y que pueden tener elementos repetidos. Por ello, dicha interface declara métodos adicionales que tienen que ver con el orden y con el acceso a elementos o rangos de elementos. Además de los métodos de Collection, la interface **List** declara los métodos siguientes:

```
Compiled from List.java
public interface java.util.List extends java.util.Collection {
    public abstract void add(int, java.lang.Object);
    public abstract boolean addAll(int, java.util.Collection);
    public abstract java.lang.Object get(int);
    public abstract int indexOf(java.lang.Object);
    public abstract int lastIndexOf(java.lang.Object);
    public abstract java.util.ListIterator listIterator();
    public abstract java.util.ListIterator listIterator(int);
    public abstract java.lang.Object remove(int);
    public abstract java.lang.Object set(int, java.lang.Object);
    public abstract java.util.List subList(int, int);
}
```

Los nuevos métodos **add()** y **addAll()** tienen un argumento adicional para insertar elementos en una posición determinada, desplazando el elemento que estaba en esa posición y los siguientes. Los métodos **get()** y **set()** permiten obtener y cambiar el elemento en una posición dada. Los métodos **indexOf()** y **lastIndexOf()** permiten saber la posición de la primera o la última vez que un elemento aparece en la lista; si el elemento no se encuentra se devuelve -1.

El método **subList(int fromIndex, toIndex)** devuelve una “vista” de la lista, desde el elemento **fromIndex** inclusive hasta el **toIndex** exclusive. Un cambio en esta “vista” se refleja en la lista original, aunque no conviene hacer cambios simultáneamente en ambas. Lo mejor es eliminar la “vista” cuando ya no se necesita.

Existen dos implementaciones de la interface **List**, que son las clases **ArrayList** y **LinkedList**. La diferencia está en que la primera almacena los elementos de la colección en un **array** de **Objects**, mientras que la segunda los almacena en una **lista vinculada**. Los **arrays** proporcionan una forma de acceder a los elementos mucho más eficiente que las listas vinculadas. Sin embargo tienen dificultades para crecer (hay que reservar memoria nueva, copiar los elementos del array antiguo y liberar la memoria) y para insertar y/o borrar elementos (hay que desplazar en un sentido u en otro los elementos que están detrás del elemento borrado o insertado). Las **listas vinculadas** sólo permiten acceso secuencial, pero tienen una gran flexibilidad para crecer, para borrar y para insertar elementos. El optar por una implementación u otra depende del caso concreto de que se trate.

5.5.4.7. Maps y SortedMaps

Un **Map** es una estructura de datos agrupados en parejas **clave/valor**. Pueden ser considerados como una tabla de dos columnas. La **clave** debe ser única y se utiliza para acceder al **valor**.

Aunque la interface **Map** no deriva de **Collection**, es posible ver los **Maps** como colecciones de **claves**, de **valores** o de parejas **clave/valor**. A continuación se muestran los métodos de la interface **Map** (comando > **javap java.util.Map**):

```
Compiled from Map.java
public interface java.util.Map {
    public abstract void clear();
    public abstract boolean containsKey(java.lang.Object);
    public abstract boolean containsValue(java.lang.Object);
    public abstract java.util.Set entrySet();
    public abstract boolean equals(java.lang.Object);
    public abstract java.lang.Object get(java.lang.Object);
    public abstract int hashCode();
    public abstract boolean isEmpty();
    public abstract java.util.Set keySet();
    public abstract java.lang.Object put(java.lang.Object, java.lang.Object);
    public abstract void putAll(java.util.Map);
    public abstract java.lang.Object remove(java.lang.Object);
    public abstract int size();
    public abstract java.util.Collection values();
    public static interface java.util.Map.Entry
    {
        public abstract boolean equals(java.lang.Object);
        public abstract java.lang.Object getKey();
        public abstract java.lang.Object getValue();
        public abstract int hashCode();
        public abstract java.lang.Object setValue(java.lang.Object);
    }
}
```

Muchos de estos métodos tienen un significado evidente, pero otros no tanto. El método **entrySet()** devuelve una “vista” del **Map** como **Set**. Los elementos de este **Set** son referencias de la interface **Map.Entry**, que es una **interface interna** de **Map**. Esta “vista” del **Map** como **Set** permite modificar y eliminar elementos del **Map**, pero no añadir nuevos elementos.

El método **get(key)** permite obtener el valor a partir de la clave. El método **keySet()** devuelve una “vista” de las claves como **Set**. El método **values()** devuelve una “vista” de los valores del **Map** como **Collection** (porque puede haber elementos repetidos). El método **put()** permite añadir una pareja clave/valor, mientras que **putAll()** vuela todos los elementos de un **Map** en otro **Map** (los pares con clave nueva se añaden; en los pares con clave ya existente los valores nuevos sustituyen a los antiguos). El método **remove()** elimina una pareja clave/valor a partir de la clave.

La interface **SortedMap** añade los siguientes métodos, similares a los de **SortedSet**:

```
Compiled from SortedMap.java
```

```

public interface java.util.SortedMap extends java.util.Map {
    public abstract java.util.Comparator comparator();
    public abstract java.lang.Object firstKey();
    public abstract java.util.SortedMap headMap(java.lang.Object);
    public abstract java.lang.Object lastKey();
    public abstract java.util.SortedMap subMap(java.lang.Object, java.lang.Object);
    public abstract java.util.SortedMap tailMap(java.lang.Object);
}

```

La clase **HashMap** implementa la interface **Map** y está basada en una hash table, mientras que **TreeMap** implementa **SortedMap** y está basada en un árbol binario.

5.5.4.8. Algoritmos y otras características especiales: Clases Collections y Arrays

La clase **Collections** (no confundir con la interface **Collection**, en singular) es una clase que define un buen número de métodos static con diversas finalidades. No se detallan o enumeran aquí porque exceden del espacio disponible. Los más interesantes son los siguientes:

- Métodos que definen algoritmos:

Ordenación mediante el método mergesort

```

public static void sort(java.util.List);
public static void sort(java.util.List, java.util.Comparator);

```

Eliminación del orden de modo aleatorio

```

public static void shuffle(java.util.List);
public static void shuffle(java.util.List, java.util.Random);

```

Inversión del orden establecido

```
public static void reverse(java.util.List);
```

Búsqueda en una lista

```

public static int binarySearch(java.util.List, java.lang.Object);
public static int binarySearch(java.util.List, java.lang.Object,
                             java.util.Comparator);

```

Copiar una lista o reemplazar todos los elementos con el elemento especificado

```

public static void copy(java.util.List, java.util.List);
public static void fill(java.util.List, java.lang.Object);

```

Cálculo de máximos y mínimos

```

public static java.lang.Object max(java.util.Collection);
public static java.lang.Object max(java.util.Collection, java.util.Comparator);
public static java.lang.Object min(java.util.Collection);
public static java.lang.Object min(java.util.Collection, java.util.Comparator);

```

- Métodos de utilidad

Set inmutable de un único eleento

```
public static java.util.Set singleton(java.lang.Object);
```

Lista inmutable con n copias de un objeto

```
public static java.util.List nCopies(int, java.lang.Object);
```

Constantes para representar el conjunto y la lista vacía

```

public static final java.util.Set EMPTY_SET;
public static final java.util.List EMPTY_LIST;

```

Además, la clase **Collections** dispone de dos conjuntos de métodos "factory" que pueden ser utilizados para convertir objetos de distintas colecciones en objetos "**read only**" y para convertir distintas colecciones en objetos "**synchronized**" (por defecto las clases vistas anteriormente no están sincronizadas, lo cual quiere decir que se puede acceder a la colección desde distintas threads sin que se produzcan problemas. Los métodos correspondientes son los siguientes:

```

public static java.util.Collection synchronizedCollection(java.util.Collection);
public static java.util.List synchronizedList(java.util.List);
public static java.util.Map synchronizedMap(java.util.Map);
public static java.util.Set synchronizedSet(java.util.Set);
public static java.util.SortedMap synchronizedSortedMap(java.util.SortedMap);
public static java.util.SortedSet synchronizedSortedSet(java.util.SortedSet);

```

```

public static java.util.Collection unmodifiableCollection(java.util.Collection);
public static java.util.List unmodifiableList(java.util.List);
public static java.util.Map unmodifiableMap(java.util.Map);
public static java.util.Set unmodifiableSet(java.util.Set);
public static java.util.SortedMap unmodifiableSortedMap(java.util.SortedMap);
public static java.util.SortedSet unmodifiableSortedSet(java.util.SortedSet);

```

Estos métodos se utilizan de una forma muy sencilla: se les pasa como argumento una referencia a un objeto que no cumple la característica deseada y se obtiene como valor de retorno una referencia a un objeto que sí la cumple.

5.5.4.9. Desarrollo de clases por el usuario: clases abstract

Las clases abstractas indicadas en la Figura 5.2, en la página 71, pueden servir como base para que los programadores con necesidades no cubiertas por las clases vistas anteriormente desarrollen sus propias clases.

5.5.4.10. Interfaces Cloneable y Serializable

Las clases **HashSet**, **TreeSet**, **ArrayList**, **LinkedList**, **HashMap** y **TreeMap** (al igual que **Vector** y **Hashtable**) implementan las interfaces **Cloneable** y **Serializable**, lo cual quiere decir que es correcto sacar copias bit a bit de sus objetos con el método **Object.clone()**, y que se pueden convertir en cadenas o flujos (streams) de caracteres.

Una de las ventajas de implementar la interface **Serializable** es que los objetos de estas clases pueden ser impresos con los métodos **System.out.print()** y **System.out.println()**.

5.6. Otras clases del package `java.util`

El package **java.util** tiene otras clases interesantes para aplicaciones de distinto tipo, entre ellas algunas destinadas a considerar todo lo relacionado con fechas y horas. A continuación se consideran algunas de dichas clases.

5.6.1. Clase Date

La clase **Date** representa un instante de tiempo dado con precisión de milisegundos. La información sobre fecha y hora se almacena en un entero long de 64 bits que contiene los milisegundos transcurridos desde las 00:00:00 del 1 de enero de 1970 GMT (*Greenwich mean time*). Ya se verá que otras clases permiten a partir de un objeto **Date** obtener información del año, mes, día, horas, minutos y segundos. A continuación se muestran los métodos de la clase **Date**, habiéndose eliminado los métodos declarados obsoletos (*deprecated*) en el JDK 1.2:

```

Compiled from Date.java
public class java.util.Date extends java.lang.Object implements
    java.io.Serializable, java.lang.Cloneable, java.lang.Comparable {
    public java.util.Date();
    public java.util.Date(long);
    public boolean after(java.util.Date);
    public boolean before(java.util.Date);
    public java.lang.Object clone();
    public int compareTo(java.lang.Object);
    public int compareTo(java.util.Date);
    public boolean equals(java.lang.Object);
    public long getTime();
    public int hashCode();
    public void setTime(long);
    public java.lang.String toString();
}

```

El constructor por defecto **Date()** crea un objeto a partir de la fecha y hora actual del ordenador. El segundo constructor crea el objeto a partir de los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT. Los métodos **after()** y **before()** permiten saber si la fecha indicada como argumento implícito (**this**) es posterior o anterior a la pasada como argumento. Los métodos **getTime()** y **setTime()** permiten obtener o establecer los milisegundos transcurridos desde el 01/01/1970, 00:00:00 GMT para un

determinado objeto **Date**. Otros métodos son consecuencia de las interfaces implementadas por la clase **Date**.

Los objetos de esta clase no se utilizan mucho directamente, sino que se utilizan en combinación con las clases que se van a ver a continuación.

5.6.2. Clases *Calendar* y *GregorianCalendar*

La clase **Calendar** es una clase **abstract** que dispone de métodos para convertir objetos de la clase **Date** en enteros que representan fechas y horas concretas. La clase **GregorianCalendar** es la única clase que deriva de **Calendar** y es la que se utilizará de ordinario.

Java tiene una forma un poco particular para representar las fechas y horas:

1. Las horas se representan por enteros de 0 a 23 (la hora o va de las 00:00:00 hasta la 1:00:00), y los minutos y segundos por enteros entre 0 y 59.
2. Los días del mes se representan por enteros entre 1 y 31 (lógico).
3. Los meses del año se representan mediante enteros de 0 a 11 (no tan lógico).
4. Los años se representan mediante enteros de cuatro dígitos. Si se representan con dos dígitos, se resta 1900. Por ejemplo, con dos dígitos el año 2000 es para Java el año 00.

La clase **Calendar** tiene una serie de variables miembro y constantes (variables **final**) que pueden resultar muy útiles:

- La variable **int** AM_PM puede tomar dos valores: las constantes enteras AM y PM.
- La variable **int** DAY_OF_WEEK puede tomar los valores **int** SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY y SATURDAY.
- La variable **int** MONTH puede tomar los valores **int** JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER. Para hacer los programas más legibles es preferible utilizar estas constantes simbólicas que los correspondientes números del 0 al 11.
- La variable miembro HOUR se utiliza en los métodos get() y set() para indicar la hora de la mañana o de la tarde (en relojes de 12 horas, de 0 a 11). La variable HOUR_OF_DAY sirve para indicar la hora del día en relojes de 24 horas (de 0 a 23).
- Las variables DAY_OF_WEEK, DAY_OF_WEEK_IN_MONTH, DAY_OF_MONTH (o bien DATE), DAY_OF_YEAR, WEEK_OF_MONTH, WEEK_OF_YEAR tienen un significado evidente.
- Las variables ERA, YEAR, MONTH, HOUR, MINUTE, SECOND, MILLISECOND tienen también un significado evidente.
- Las variables ZONE_OFFSET y DST_OFFSET indican la zona horaria y el desafío en milisegundos respecto a la zona GMT.

La clase **Calendar** dispone de un gran número de métodos para establecer u obtener los distintos valores de la fecha y/u hora. Algunos de ellos se muestran a continuación. Para más información, se recomienda utilizar la documentación de JDK 1.2.

```
Compiled from Calendar.java
public abstract class java.util.Calendar extends java.lang.Object implements
java.io.Serializable, java.lang.Cloneable {
    protected long time;
    protected boolean isTimeSet;
    protected java.util.Calendar();
    protected java.util.Calendar(java.util.TimeZone, java.util.Locale);
```

```

public abstract void add(int, int);
public boolean after(java.lang.Object);
public boolean before(java.lang.Object);
public final void clear();
public final void clear(int);
protected abstract void computeTime();
public boolean equals(java.lang.Object);
public final int get(int);
public int getFirstDayOfWeek();
public static synchronized java.util.Calendar getInstance();
public static synchronized java.util.Calendar getInstance(java.util.Locale);
public static synchronized java.util.Calendar getInstance(java.util.TimeZone);
public static synchronized java.util.Calendar getInstance(java.util.TimeZone,
                                                       java.util.Locale);

public final java.util.Date getTime();
protected long getTimeInMillis();
public java.util.TimeZone getTimeZone();
public final boolean isSet(int);
public void roll(int, int);
public abstract void roll(int, boolean);
public final void set(int, int);
public final void set(int, int, int);
public final void set(int, int, int, int, int);
public final void set(int, int, int, int, int, int);
public final void setTime(java.util.Date);
public void setFirstDayOfWeek(int);
protected void setTimeInMillis(long);
public void setTimeZone(java.util.TimeZone);
public java.lang.String toString();
}

```

La clase ***GregorianCalendar*** añade las constante BC y AD para la ERA, que representan respectivamente antes y después de Jesucristo. Añade además varios constructores que admiten como argumentos la información correspondiente a la fecha/hora y -opcionalmente- la zona horaria.

A continuación se muestra un ejemplo de utilización de estas clases. Se sugiere al lector que cree y ejecute el siguiente programa, observando los resultados impresos en la consola.

```

import java.util.*;

public class PruebaFechas {
    public static void main(String arg[]) {
        Date d = new Date();
        GregorianCalendar gc = new GregorianCalendar();
        gc.setTime(d);
        System.out.println("Era:           " + gc.get(Calendar.ERA));
        System.out.println("Year:          " + gc.get(Calendar.YEAR));
        System.out.println("Month:         " + gc.get(Calendar.MONTH));
        System.out.println("Dia del mes:   " + gc.get(Calendar.DAY_OF_MONTH));
        System.out.println("D   de   la   S   en   mes: "
                           + gc.get(Calendar.DAY_OF_WEEK_IN_MONTH));
        System.out.println("No de semana:  " + gc.get(Calendar.WEEK_OF_YEAR));
        System.out.println("Semana del mes: " + gc.get(Calendar.WEEK_OF_MONTH));
        System.out.println("Fecha:          " + gc.get(Calendar.DATE));
        System.out.println("Hora:           " + gc.get(Calendar.HOUR));
        System.out.println("Tiempo del dia: " + gc.get(Calendar.AM_PM));
        System.out.println("Hora del dia:   " + gc.get(Calendar.HOUR_OF_DAY));
        System.out.println("Minuto:         " + gc.get(Calendar.MINUTE));
        System.out.println("Segundo:        " + gc.get(Calendar.SECOND));
        System.out.println("Dif. horaria:   " + gc.get(Calendar.ZONE_OFFSET));
    }
}

```

5.6.3. Clases *DateFormat* y *SimpleDateFormat*

DateFormat es una clase ***abstract*** que pertenece al package ***java.text*** y no al package ***java.util***, como las vistas anteriormente. La razón es para facilitar todo lo referente a la internacionalización, que

es un aspecto muy importante en relación con la conversión, que permite dar formato a fechas y horas de acuerdo con distintos criterios locales. Esta clase dispone de métodos **static** para convertir **Strings** representando fechas y horas en objetos de la clase **Date**, y viceversa.

La clase **SimpleDateFormat** es la única clase derivada de **DateFormat**. Es la clase que conviene utilizar. Esta clase se utiliza de la siguiente forma: se le pasa al constructor un **String** definiendo el formato que se desea utilizar. Por ejemplo:

```
import java.util.*;
import java.text.*;

class SimpleDateForm {
    public static void main(String arg[]) throws ParseException {
        SimpleDateFormat sdf1 = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");
        SimpleDateFormat sdf2 = new SimpleDateFormat("dd-MM-yy");
        Date d = sdf1.parse("12-04-1968 11:23:45");
        String s = sdf2.format(d);
        System.out.println(s);
    }
}
```

La documentación de la clase **SimpleDateFormat** proporciona abundante información al respecto, incluyendo algunos ejemplos.

5.6.4. Clases **TimeZone** y **SimpleTimeZone**

La clase **TimeZone** es también una clase abstracta que sirve para definir la zona horaria. Los métodos de esta clase son capaces de tener en cuenta el cambio de la hora en verano para ahorrar energía. La clase **SimpleTimeZone** deriva de **TimeZone** y es la que conviene utilizar.

El valor por defecto de la zona horaria es el definido en el ordenador en que se ejecuta el programa. Los objetos de esta clase pueden ser utilizados con los constructores y algunos métodos de la clase **Calendar** para establecer la zona horaria.

Parte 2

Internet

6. Internet y HTML

6.1. Protocolo TCP/IP

Lo que consigue que los ordenadores remotos se entiendan y que Internet funcione es un conjunto de instrucciones complicadas llamadas protocolo. La Internet utiliza varios protocolos, los más importantes son el Transport Control Protocol (TCP) y el llamado Internet Protocol (IP), o TCP/IP para abreviar. Son una serie de reglas para mover los datos electrónicos en paquetes y asegurarse de que son reensamblados correctamente cuando llegan al destino. Todos los ordenadores en Internet hablan TCP/IP, y gracias a ello se consigue eliminar la barrera de la heterogeneidad de los ordenadores.

6.2. Protocolo HTTP y lenguaje HTML

Gracias a los protocolos se puede recibir cualquier tipo de fichero de cualquier ordenador conectado a Internet, se puede enviar correo, se puede conectar a un servidor remoto, etc. Pero ninguno de estos servicios permiten la posibilidad de colaborar en la creación de un entorno multimedia, es decir, no se pueden pedir datos a un ordenador remoto para visualizarlos localmente utilizando TCP/IP. Es por ello que en 1991 se creó un nuevo protocolo llamado HTTP (HyperText Transport Protocol).

Una de las características del protocolo HTTP es que no es *permanente*, es decir, una vez que el servidor ha respondido a la petición del cliente la conexión se pierde y se queda en espera, al contrario de lo que ocurre con los servicios de **ftp** o **telnet**, en los cuales la conexión es permanente hasta que el usuario transmite la orden de desconexión. Esto tiene la ventaja de que el servidor no se colapsa, y el inconveniente de que complica la seguridad cuando los accesos se hacen con password, pues no se puede pedir el password cada vez que se realiza una conexión (sólo se pide la primera vez).

La grandeza del HTTP es que se pueden crear recursos multimedia localmente, transferirlos fácilmente a un ordenador remoto y visionarlos donde se han enviado. HTTP es una herramienta muy poderosa y que es la esencia del World Wide Web.

Para la creación de un Web en Internet se utiliza el lenguaje llamada HTML (HyperText Markup Language). Es un lenguaje muy simple cuyo código se puede escribir con cualquier editor de texto. Se basa en comandos reconocibles por el browser y que van entre los símbolos '<' y '>'. El lenguaje HTML junto con aplicaciones **CGI** (Common Gateway Interface) y **Java** hacen posible la creación de páginas Web muy vistosas.

6.3. URL (Uniform Resource Locator)

Todo ordenador en Internet y toda persona que use Internet tiene su propia dirección electrónica. Todas estas direcciones siguen un mismo formato. Para el alumno Pedro Gómez de la Escuela Superior de Ingenieros de San Sebastián su dirección puede ser: pgomez@gaviota.tecnun.es donde pgomez es el identificador ID que Pedro utiliza para conectarse a la red. Es así como el ordenador le conoce. La parte de la dirección que sigue al símbolo de arroba (@) se llama el **dominio**. El dominio es la dirección específica del ordenador, en este caso el ordenador se llama **gaviota** adjunto a TECNUN en España (es). Nunca hay espacios en una dirección de Internet.

El dominio está dividido en este caso en tres subdominios que se leen de derecha a izquierda. Entre estos dominios se pueden encontrar los siguientes (utilizados en Estados Unidos):

com	organizaciones comerciales.	mil	militar.
gov	gobierno.	net	organización de redes.
int	organización internacional.	org	organizaciones sin ánimo de lucro.

Estos subdominios se refieren al tipo de organización al que pertenece el servidor. También podemos encontrar información sobre el país en el que se encuentra el ordenador al que pertenece la dirección:

at	Austria	au	Australia	es	España	fi	Finlandia
ca	Canadá	ch	Suiza	fr	Francia	gr	Grecia
de	Alemania	dk	Dinamarca	jp	Japón	uk	Reino Unido

Los ordenadores también son conocidos por un número llamado la **dirección IP** y es lo que el ordenador realmente entiende. Los nombres son para facilitar la tarea a los usuarios. Por ejemplo el número que corresponde al dominio *gaviota.tecnun.es* es 193.145.249.21.

Así, ¿qué es un URL? Pues podría concebirse como la extensión del concepto de archivo: no sólo puede apuntarse a un archivo en un directorio, sino que además tal archivo y tal directorio existen de hecho en cualquier ordenador de la red. Los URLs posibilitan el direccionamiento, tanto de gente como de aplicaciones de software a una gran variedad de información, disponible en distintos protocolos de Internet. El más conocido es el HTTP, pero los FTP también pueden ser referenciados por medio de URLs. Incluso la dirección de e-mail de Internet de alguien puede ser referida con un URL. Un URL es como la dirección *completa* de correo: proporciona toda los datos necesarios para que la información llegue al lugar de destino.

En resumen, un URL es una manera muy conveniente y sucinta de referirse a un archivo u otro recurso electrónico.

La sintaxis genérica de los URLs es la que sigue:

método://ordenador.dominio/ruta-completa-del-fichero

donde **método** puede ser una de las siguientes palabras: **http**, **ftp**, **telnet**, ... Enseguida se verá la sintaxis específica de cada método.

Pero antes, unas breves observaciones:

- Hay ocasiones en las que se verá que el URL empleado tiene la misma sintaxis que arriba, pero acabada la ruta completa del fichero con una barra (/). Esto no quiere decir más que no se apunta a un archivo específico, sino a un directorio. El servidor generalmente devolverá el índice por defecto de ese directorio (un listado de archivos y subdirectorios de ese directorio para acceder al que se desee), o un archivo por defecto que el servidor busca automáticamente en el directorio.
- ¿Qué hacer una vez que se tiene un URL y se quiere utilizar? Muy sencillo: como se verá más adelante, todos los browsers poseen una ventana (*location box*) donde es posible escribir o pegar un URL.
- ¿Cómo presentar un URL a otros? Se suele recomendar la siguiente manera:

<URL: método://ordenador.dominio/ruta-completa-del-fichero>

para distinguir los URLs de los URIs (**Uniform Resource Identification**), que representan un concepto similar.

- Finalmente, destacar que hay caracteres considerados reservados o inseguros. Para poder usarlos, habrá que emplear las secuencias de escape, que utilizan el formato "%+US-ASCII-valor de un carácter hexadecimal" como puede verse en las tablas que hay a continuación:

1. **Caracteres inseguros:** son aquellos considerados como tales por alguna razón determinada, normalmente porque tienen su propio significado en algún sistema operativo.

Espacio	%20	<	%3C	>	%3E	#	%23
%	%25	{	%7B	}	%7D		%7C
\	%74	^	%5E	~	%7E	[%5B
]	%5D	`	%60				

2. **Caracteres reservados:** son aquellos considerados como tales por tener un significado especial en los URLs; para evitar tal significado también habrá que codificarlos en hexadecimal.

;	%3B	/	%2F	@	%40	=	%3D
?	%3F	:	%3A	&	%26		

A continuación se muestran las distintas formas de construir los URLs según los distintos servicios de Internet:

6.3.1. URLs del protocolo HTTP

HTTP es el protocolo específicamente diseñado para usar con la World Wide Web. Su sintaxis es:

```
http://<host>:<puerto>/<ruta>#<parte a buscar>
```

donde *host* es la dirección del servidor WWW, el *puerto* puede ser omitido (valor por defecto, 80), la *ruta* indica al servidor el fichero que se desea y *#parte a buscar* nos sitúa en una parte del texto que esté apuntada por una TAG ancla (anchor), que se explicarán más adelante.

Así, por ejemplo, *http://www.msn.com/index/prev/welcome.htm#offers* accede al Web de Microsoft Network, al archivo *welcome.htm* (cuya ruta de acceso es *index/prev*) y dentro de éste, al lugar marcado con el ancla *offers*.

6.3.2. URLs del protocolo FTP

La sintaxis específica del ftp es:

```
ftp://<usuario>:<password>@<host>:<puerto>/<cwd1>/<cwd2>/.../< cwdN>/<nombre>
```

Los comandos *usuario* y *password* pueden ser omitidos. *Host* es la dirección del ftp. El *puerto*, como antes, puede ser omitido. Las series *< cwd1>/.../< cwdN>* son los comandos que el cliente deberá emplear para moverse al directorio en el que reside el documento.

Así, por ejemplo, *ftp://www.msn.com/index/prev/welcome.htm* traerá el fichero *welcome.htm* (cuya ruta de acceso es *index/prev*) del web de **Microsoft Network**.

6.3.3. URLs del protocolo correo electrónico (mailto)

Su sintaxis difiere de las anteriores, y es la que sigue:

```
mailto:<cuenta@lugar>
```

siendo *cuenta@lugar* la dirección de correo electrónico a la cual se desea enviar un mensaje. Por ejemplo, *mailto:pgomez@ceit.es* manda un mensaje a P.Gómez, en el CEIT, en España.

6.3.4. URLs del protocolo Telnet

El URL para *Telnet* designa una sesión interactiva con un lugar remoto en Internet, mediante el protocolo *Telnet*. Su sintaxis es la siguiente:

```
telnet://<usuario>:<password>@<host>:<puerto>/
```

Los parámetros *user* y ***password*** pueden ser omitidos, *host* se refiere al lugar al que se va a realizar la conexión y *port* puede ser igualmente omitido (siendo su valor por defecto "23").

6.3.5. Nombres específicos de ficheros

El URL de método *file* supone que un fichero puede ser obtenido por un cliente. Esto suele confundirse con el método *ftp*. La diferencia radica en que *ftp* es un método específico para la transmisión de ficheros, mientras que el método *file* deja el método de traída de ficheros a elección del cliente que, en algunas circunstancias, bien podría ser el método *ftp*. La sintaxis para el método *file* es la que se ve a continuación:

```
file://<host>/<ruta-de-acceso>
```

Como siempre, *host* es la dirección y *ruta-de-acceso* es el camino jerárquico para acceder al documento (con una estructura directorio/directorio/.../nombre del archivo). Si el parámetro *host* se deja en blanco, se supondrá por defecto *localhost*, es decir, que se van a traer ficheros localmente.

6.4. HTML

HTML (*HyperText Markup Language*) es el lenguaje utilizado en Internet para definir las páginas del **World Wide Web**. Los ficheros HTML son ficheros de texto puramente ASCII, que pueden ser escritos con cualquier editor básico, tal como **Notepad** en **Windows** o **vi** en **Unix**. También se pueden utilizar procesadores de texto más complicados como **Microsoft Word**, pero en este caso hay que asegurarse que el fichero es guardado en disco como "text only". En este fichero de texto se introducen unas marcas o

caracteres de control llamadas TAGs (en esto, HTML se parece a los primeros procesadores de texto), que son interpretadas por el browser. Cuando éste lee un fichero ASCII con extensión ***.htm** o ***.html** interpreta estas TAGs y formatea el texto de acuerdo con ellas.

En general puede decirse que HTML es un lenguaje sencillo y eficiente. Aunque no puede competir con los procesadores de texto en capacidades de formato, es universal, es hipertexto e hipermedia, es muy accesible, sus ficheros ocupan poco espacio en disco; por otra parte es fácil de interpretar y de enviar a través de las redes. De hecho, es uno de los estándares en los cuales las empresas están basando sus **Intranets** y sus servicios de información interna.

En la página web de la asignatura está disponible la especificación HTML 4.01 (última versión).

6.5. Tags generales

El formato con el que imprime o visualiza un fichero HTML depende de unas marcas especiales llamadas TAGs. Todas las TAGs de HTML van encerradas entre los caracteres "<" y ">", como por ejemplo <HTML>. Además, la mayor parte de ellas son **dobles**: hay una TAG de comienzo y otra de final; entre ambas hay un **contenido** o texto afectado por dichas TAGs. La TAG de final es como la de comienzo, pero incluyendo una barra (/) antes del nombre, por ejemplo </HTML>. Normalmente las TAGs se escriben en letra mayúscula para hacer más legible el cuerpo del documento (distinguiéndolas del texto ordinario), pero de hecho también se pueden escribir en minúsculas. La forma general de estas TAGs dobles es la siguiente:

```
<COMANDO>Texto afectado</COMANDO>
```

Existen también TAGs **simples** (sin TAG de cierre), que no tienen un texto contenido al que se aplican. Por ejemplo, la marca de comienzo de párrafo <P> era un comando simple hasta la versión 3.0 de HTML, pues cada vez que empieza un párrafo se sobreentiende que ha terminado el anterior. La versión 3.0 de HTML ha introducido la TAG </P>, aunque su uso es opcional.

Una TAG, tanto si es *doble* o *simple*, puede tener uno o más **atributos**, que son parámetros que definen su forma de actuar. Los atributos se incluyen después del nombre de la TAG, antes del carácter ">", separados por uno o más blancos. Si la TAG es doble, los atributos se incluyen siempre en la TAG de apertura.

Los atributos que aún se soportan pero pueden quedar obsoletos en futuras versiones se indicarán como **[deprecated]**. Estos atributos han sido sustituidos por otras alternativas, generalmente utilizando hojas de estilo (*style sheets*), que no se verán en esta asignatura.

Se puede anidar unas TAGs dentro de otras. Se podría decir que en este caso los efectos son acumulativos, como en el siguiente ejemplo:

```
<TAG1>
    Texto afectado por el comando 1
    <TAG2>
        Texto afectado por las TAGs 1 y 2
    </TAG2>
</TAG1>
```

Advertencia: La *anidación* de TAGs debe hacerse de forma coherente, pues de lo contrario puede dar unos resultados que no son los esperados: la última TAG en abrirse debe ser la primera en ser cerrada, es decir que no se pueden entrecruzar los dominios de actuación de las TAGs.

Todos los ficheros HTML tienen una estructura similar a la siguiente:

```
<HTML>
    <HEAD>
        <TITLE>Título de la página</TITLE>
        ...
    </HEAD>
    <BODY>
        ...
    </BODY>
</HTML>
```

donde las marcas mostradas indican lo siguiente:

- <HTML>...</HTML>: Indica el comienzo y el fin de un documento HTML. Es necesario para que el browser sepa que tiene que interpretar un fichero en lenguaje HTML.
- <HEAD>...</HEAD>: Indica la cabecera del documento donde se guarda diversa información sobre el mismo, como por ejemplo el título de la ventana en la que aparecerá (<TITLE>Este es el título</TITLE>).
- <BODY>...</BODY>: En él aparece el cuerpo del documento Web propiamente dicho, que contiene el texto, las imágenes, los enlaces o *links* a otras páginas, etc.

6.6. Formato de texto

En este apartado se verá cómo organizar y formatear el texto de un documento.

6.6.1. TAGs generales de un documento.

Existen una serie de atributos para la TAG BODY, que afectarán a todo el documento, dado que todo el código de las páginas web se escribe dentro de la TAG doble BODY. Entre otros, los más utilizados son éstos:

- *[deprecated]* BGCOLOR="nombre_color" ó BGCOLOR="#00FF00", donde *nombre_color* es uno de los nombres reconocidos por los browsers (existe una tabla definida por Netscape, mayoritariamente aceptada), y "#00FF00" es el código RGB del color deseado (dos dígitos hexadecimales por cada color fundamental rojo, verde y azul, para representar hasta 256 intensidades de cada color, y unos 16 millones de colores distintos). Esta segunda opción es la más empleada, por ser la estándar (y por ello no da errores).
- *[deprecated]* BACKGROUND="imagen.ext": sirve para poner un dibujo como fondo de documento, siendo **imagen.ext** el nombre del fichero y la extensión. Si el dibujo no ocupa la pantalla entera, se dispondrá en mosaico hasta llenarla.

```
<BODY BACKGROUND="imagen.ext">
...
</BODY>
```

- *[deprecated]* TEXT="código de color": define el color del texto del documento.
- *[deprecated]* LINK="código de color": define el color del texto definido como *link*.
- *[deprecated]* VLINK="código de color": define el color de los links visitados (es decir, los links sobre los que ya se ha clicado en alguna ocasión).

6.6.2. Comentarios en HTML

El lenguaje HTML permite introducir comentarios, con el fin de que un documento sea más claro y comprensible cuando se analiza su código. Lógicamente, tales comentarios no aparecerán en el browser. La sintaxis de los comentarios, que pueden ocupar tantas líneas como se desee, es la que sigue:

```
<!-- Comentario -->
```

6.6.3. Caracteres de separación en HTML

Se consideran caracteres de separación el espacio en blanco, el tabulador y el salto de línea. Un punto importante a considerar es que, en principio, HTML considera varios caracteres de separación consecutivos como un único carácter de separación. Además, les otorga a todos ellos la misma categoría, es decir, considera todos como si fuesen espacios en blanco. Esto quiere decir que el aspecto del fichero ASCII que contiene el código HTML y el aspecto del documento visto en el browser pueden diferir notablemente. De ahí que al escribir un fichero luego haya que comprobarlo en el browser para ver si el resultado final es el esperado. Por ejemplo, introducir un salto de línea en un fichero HTML no implica, en principio, un salto de línea en la página Web.

Así pues, para organizar el texto HTML no se basa en los caracteres de separación citados sino en TAGs especiales para ello. Esas TAGs se considerán en el apartado siguiente. Los puntos suspensivos (...) entre las TAGs de comienzo y final representan un contenido a concretar

6.6.4. TAGs de organización o partición de un documento

Las TAGs más importantes de organización del contenido de un documento son las siguientes:

- **Tag <P>...</P> (de *paragraph*)**. Cada párrafo deberá ir incluido entre estas TAGs. Entre párrafo y párrafo se deja el espacio correspondiente a una línea en blanco (aproximadamente).
- Se puede añadir a <P> el atributo ALIGN [*deprecated*] para especificar cómo debe ser alineado un párrafo. Este atributo puede tomar los valores "LEFT", "CENTER" y "RIGHT".
- **Tag
 (de *break*)**. Salta de línea sin dejar una línea en blanco a continuación. Es un comando simple.
- **Tag <HR> (de *hard rule*)**. Este comando simple introduce una **Línea horizontal** que deja espacios antes y después, por lo que no hace falta añadir <P> o
 suplementarios.

Algunos atributos de este comando son:

- [*deprecated*] WIDTH: define la longitud de la línea horizontal en la pantalla, que puede especificarse usando un valor absoluto (número de pixels) o bien indicando un porcentaje (opción recomendada, ya que no es posible conocer a priori la anchura de la ventana del cliente Web), con lo que el resultado final podría no ser el esperado. El valor por defecto es del 100%.

```
<HR WIDTH=75%>
```

- [*deprecated*] ALIGN: alinea la línea donde se deseé. Puede tomar los valores LEFT, RIGHT y CENTER

```
<HR ALIGN=RIGHT>
```

- [*deprecated*] SIZE: especifica la anchura o grosor de la línea horizontal en número de pixels. El valor por defecto es 1 pixel

```
<HR SIZE=4>
```

- [*deprecated*] NOSHADE: Por defecto, estas líneas aparecen sombreadas, con un cierto efecto 3-D. Con este comando, tal efecto desaparecerá.

Puede observarse que no hay ningún atributo que haga referencia al color de la línea, que siempre se dibuja en negro. Conviene tener esto en cuenta a la hora de elegir colores de fondo.

6.6.5. Formateo de textos sin particiones

- **Tag <PRE>...</PRE>(de *preformatted*)**. Como ya se ha mencionado, el browser ignorará de ordinario todos los espaciados de línea, múltiples espacios consecutivos, tabuladores, etc. Por lo general esta práctica es muy adecuada, pero pueden presentarse ocasiones en las que se deseé un formato específico de texto, que aparezca en la pantalla tal y como se ha escrito (un caso típico son los listados de programas de ordenador, poesías, letras de canciones, etc.). Este objetivo se logra encerrando el texto entre el comando doble <PRE> *texto* </PRE>, tal como puede verse en la Figura 6.1.

Por ejemplo,
 si quisiera
 crear alguna
 figura con
 palabras,
 Necesitaría
 usar
 el TAG
 PRE
 para
 hacerlo.

Figura 6.1

- **Tag <BLOCKQUOTE>...</BLOCKQUOTE> (de *block quote*)**. Esta TAG de formateo de párrafos es muy útil. Inserta un salto de párrafo e indenta todo el texto que viene a continuación, hasta llegar a su TAG emparejada que, a su vez, insertará otro salto de párrafo y suprimirá la indentación anterior. Así la TAG BLOCKQUOTE debe emplearse para indentar bloques determinados de textos, pero en ningún caso con el único objetivo de mover el margen, ya que el comportamiento del texto afectado por este comando varía según los browsers.

6.6.6. Centrado de párrafos y cabeceras con <CENTER>

- **Tag <CENTER>...</CENTER>**. Esta TAG se emplea para centrar todo tipo de texto, figuras, tablas y otros elementos, como por ejemplo:

```
<CENTER> Texto afectado por el centrado </CENTER>
```

6.6.7. Efectos de formato en texto

- **Tag <H>...</H>(de *heading*)**. Es una tag doble que toma necesariamente un argumento, un número comprendido entre 1 y 6, de tal modo que <H1> es un título de primer nivel (normalmente el más grande) y <H6> es un título de sexto nivel (de ordinario el más pequeño, aunque el hecho de que sea más grande o pequeño que el estándar depende del browser). Inserta automáticamente un salto de párrafo, antes y después. Puede tomar como atributo ALIGN [deprecated], utilizado de la misma forma que en <P>
- **Tags ... , <I>... </I> y <U>... </U> (de *bold, italic y underlined* respectivamente)**. hace que el texto afectado esté en negrita (bold), <I> aplica el estilo cursiva y <U> [deprecated] lo subraya.
- **Tag <TT>...</TT>(de *teletype*)**. Esta TAG hace que la letra que se imprima sea de estilo teletipo, es decir, que su apariencia sea similar a la que de una máquina de escribir (fuente Courier o similar).
- **Tag ... [deprecated]**. Esta TAG sirve para controlar, por medio de sus atributos, el tamaño, el color y el tipo de letra con el que se representará el texto (si el browser es capaz de ello). Los atributos de esta TAG son los siguientes:
 - [deprecated] SIZE=n (siendo n un entero entre 1 y 7). Con este atributo se puede cambiar el tamaño de la letra. El tamaño base o tamaño por defecto es el 3.
 - [deprecated] SIZE=+n ó SIZE=-n. Con esta TAG se cambia el tamaño de la letra respecto al tamaño base, aumentando o reduciéndolo según se aplique un incremento positivo o negativo. Los tamaños deben estar comprendidos en la escala de 1 a 7.
 - [deprecated] COLOR="nombre_color" ó COLOR="#00FF00", donde nombre_color es uno de los nombres reconocidos por los browsers y "#00FF00" es el código RGB del color deseado.
 - [deprecated] FACE="tipo_letra", donde tipo_letra puede ser "Arial", "Times New Roman", etc.

Ejemplo: Texto a representar

- **Tag <BASEFONT SIZE=n> [deprecated]**. Establece el tamaño de letra base o por defecto. Es un TAG simple.
- **Tags ^{...} y _{...} (de *superscript* y *subscript*)**. <sup> y <sub> crean los estilos de superíndice o subíndice, respectivamente.
- **Tags <BIG>...</BIG> y <SMALL>...</SMALL>**. Las TAGs <BIG> y <SMALL> hacen al texto más grande y más pequeño respectivamente.

6.6.8. Caracteres especiales

Existen algunos caracteres, denominados especiales, que merecen una pequeña mención. Los que tienen más utilidad para el castellano son:

Las vocales acentuadas se escriben en HTML como sigue:

- Si son mayúsculas, &(LETRAMAYÚSC)**acute**. Por ejemplo, Á se escribe Á.
- Si son minúsculas, &(letramin)**acute**. Por ejemplo, é se escribe é.

- Ñ es Ñ y ñ es ñ.

6.7. Listas

6.7.1. Listas desordenadas

Una lista desordenada es una lista de elementos en líneas separadas precedidas de un guión o similar (en Inglés *bullet*).

- Tag **...** (de *unordered list*): comienzo de una lista no ordenada

Un atributo de esta Tag es TYPE [deprecated], que define la forma del *bullet* y puede tomar los valores: DISC(□), CIRCLE(●), que es el valor por defecto y SQUARE(■).

- Tag **** (de *line*): nuevo elemento de la lista.

```
<HTML>
<HEAD>
    <TITLE></TITLE>
</HEAD>
<BODY>
    Esto es una lista no ordenada:
    <UL>
        <LI>Elemento 1
        <LI>Elemento 2
        <LI>Elemento 3
    </UL>
</BODY>
</HTML>
```

Esto es una lista no ordenada:

- Elemento 1
- Elemento 2
- Elemento 3

6.7.2. Listas ordenadas

En este tipo de listas se emplean números en lugar de *bullets*.

- Tag **...** (de *ordered list*): comienzo de una lista ordenada.

Su atributo TYPE [deprecated] permite definir el tipo de numeración. Puede tomar los valores: A (para numerar A,B,C...), a (a,b,c...), I (I,II,III...), i(i,ii,iii), 1 (1,2,3...) que es el valor por defecto.

- Tag **** (de *line*): nuevo elemento de la lista .

Su atributo VALUE [deprecated] permite definir a partir de qué valor se empieza a numerar. Obviamente, por defecto comienza a numerarse por el principio. Por ejemplo, <LI VALUE=iii> numerará iii,iv,v... Cada vez que se utilice este atributo VALUE dentro de una misma lista ordenada se rompe con la numeración anterior.

```
<HTML>
<HEAD>
    <TITLE></TITLE>
</HEAD>
<BODY>
    Esto es una lista ordenada:
    <OL>
        <LI>Elemento 1
        <LI>Elemento 2
        <LI>Elemento 3
    </OL>
</BODY>
</HTML>
```

Esto es una lista ordenada:

1. Elemento 1
2. Elemento 2
3. Elemento 3

6.7.3. Listas de definiciones

Este tipo de lista es útil para obtener resultados como el que se ve a continuación:

```

Término 1
    Definición del término 1
Término 2
    Definición del término 2
...

```

- **Tag <DL>...</DL> (de *definition list*)**. comienzo de la definición de la lista.
- **Tag <DT> (de *definition term*)** - nueva definición.
- **Tag <DD> (de *definition description*)** - cuerpo o descripción de la definición.

6.8. Imágenes

El lenguaje HTML además de hipertexto, es hipermedia, es decir, que por lo tanto permite incluir información de tipo gráfico. Las imágenes se tratan de forma separada al texto, y, debido al tamaño que éstas suelen tener en general, se tarda más tiempo tanto en generar como en visualizar páginas HTML con imágenes.

- **Tag IMG (de *image*)**: Es un comando simple que se utiliza para insertar una imagen en el documento. Algunos atributos de esta tag son:
 - SRC="imagen.ext": es requerido y sirve para indicar dónde se va a encontrar la imagen, siendo **imagen.ext** el nombre del fichero y la extensión.
``
 - ALT="texto" donde *texto* es el texto que se verá en lugar de la imagen, bien porque se han desactivado los gráficos para navegar más rápidamente por Internet, bien porque el browser es "text only". Es requerido para evitar problemas en dichos casos.
 - WIDTH y HEIGHT: controlan el tamaño de la ventana, que puede especificarse usando un valor absoluto (número de pixels) o bien indicando un porcentaje (opción recomendada, ya que no es posible conocer a priori la anchura de la ventana del cliente Web).
``
 - ALIGN: alinea el texto donde se deseé respecto a la imagen. Puede tomar los valores TOP, BOTTOM, MIDDLE, LEFT, RIGHT, TEXTTOP, ABSMIDDLE, BASELINE, ABSBOTTOM.
 - BORDER=n. Permite añadir un borde o marco a la imagen, siendo *n* la anchura de éste en pixels. BORDER=0 implica la supresión del borde.
 - HSPACE=n y VSPACE=n. Estos atributos dejan un espacio horizontal y vertical respectivamente alrededor de la imagen, siendo *n* el valor en pixels.
 - LOWSRC=URL carga primero una imagen de baja resolución y cuando termina de cargar todo el documento la sustituye por la versión de alta resolución.

6.9. Links

Encerrando un texto o un gráfico entre las TAGs **<A> ... ** se consigue convertir dicho texto en hipertexto. Este link puede hacerse, entre otras posibilidades, a:

- Otra página Web.
- El correo electrónico de otra persona (se emplea el link para enviarles correo).
- Un archivo local (¡práctica a evitar si se pretende que los demás puedan acceder a ese link!).
- Una dirección relativa a documentos en directorios superiores o inferiores al directorio en que se está en ese momento o en el directorio actual.

Por lo general, los links tienen la forma **texto**. A su vez, los URLs tiene una estructura del tipo siguiente:

método://localización/archivo/

donde *método* hace referencia al modo de acceder al fichero (http, ftp, gopher, news, mailto), *localización* es el lugar donde se encuentra el fichero actualmente y *archivo* es el nombre completo del *archivo* requerido en el sistema especificado. Veamos algunos ejemplos:

```
<A HREF="http://www.ceit.es/">WEB CEIT</A>
```

Ileva al web del CEIT al clicar sobre las palabras WEB CEIT, que aparecerán subrayadas y en otro color (hiperlink).

```
<A HREF="mailto://president@whitehouse.gov">mail</A>
```

mandará un e-mail al Presidente de los EE.UU.

En el caso en que se quiera "saltar" a otro punto del propio documento, se procederá como se ve en el siguiente ejemplo: Establecemos el link (Atención: si es posible evitar los espacios, es mejor hacerlo, pues pueden originar problemas). A este tipo de links se les denomina anclas o "anchors" puesto que definen zonas fijas de un documento a las que se va a hacer referencia. Una vez efectuada esta operación, si ahora hacemos *texto* , al clicar sobre la palabra *texto* iremos al lugar donde se ha definido el link.

6.10. Tablas

- Las tablas se definen empleando los códigos pareados <TABLE> y </TABLE>.
- Las celdas se agrupan en filas, que se definen con los códigos pareados <TR> y </TR> (**de Table Row**).
- Una tabla se compone de celdas de datos. Una celda se define usando los códigos pareados <TD> y </TD> (**de Table Data**).
- Las celdas pueden contener cualquier elemento HTML: texto, imágenes, enlaces e incluso otras tablas anidadas.
- <TH>...</TH> (**de Table Header**): Para crear celdas cuyo texto sea resaltado (por ejemplo, en los encabezamientos) se sustituye el TD habitual por TH.

Algunos atributos que pueden añadirse a TABLE son:

- WIDTH: especifica la anchura de la tabla. Puede darse un valor absoluto en pixels o un valor relativo en porcentaje. Si no se especifica la anchura por defecto dependerá del browser.
- BORDER: Si se quiere que la tabla posea bordes: <TABLE BORDER(=n)> ... </TABLE>
- siendo n el grosor del borde en pixels. Por ser opcional (toma un valor por defecto) se ha denotado entre paréntesis.
- [deprecated] ALIGN: Posiciona la tabla respecto al documento. POSIC puede tomar los valores LEFT, CENTER y RIGHT.
- CELLSPACING: espaciado entre celdillas: <TABLE CELLSPACING(=n)> ... </TABLE>
- CELLPADDING(=n): permite especificar el ancho que debe existir desde los bordes de cada celdilla a los elementos en ella incluidos.
- [deprecated] BGCOLOR: indica el color de fondo de la tabla.

Algunos atributos de TR son:

- ALIGN: se utiliza para determinar la alineación del contenido de las celdas de la fila.
- VALIGN: se utiliza para determinar la posición vertical del contenido de las celdas de la fila.
- [deprecated] BGCOLOR: indica el color de fondo de la fila.

Algunos atributos que pueden añadirse a TH y TD son:

- ROWSPAN(=n): indica el número de filas que debe abarcar la celda actual.

- COLSPAN (=n): indica el número de columnas de la fila que abarcará la celda actual.
- ALIGN: se utiliza para determinar la alineación del contenido de la celda.
- VALIGN: se utiliza para determinar la posición vertical del contenido de la celda.
- WIDTH: especifica la anchura de la celda, en pixels o en porcentaje.
- HEIGHT: especifica la altura de la celda, en pixels o en porcentaje.
- [deprecated] BGCOLOR: indica el color de fondo de la celda.

Un ejemplo de tabla con varios de los elementos vistos sería la de la figura, que corresponde al siguiente código:

```
<HTML>
<HEAD><TITLE>Ejemplo de tabla</TITLE></HEAD>
<BODY>
<TABLE BORDER=3>
<CAPTION><STRONG>Ejemplo</STRONG></CAPTION>
<TR>
<TD>Elemento 1</TD>
<TD>Elemento 2</TD>
<TD ROWSPAN=3>Elemento</TD>
<TD>Total</TD>
</TR>
<TR>
<TD>Elemento 3</TD>
<TD>Elemento 4</TD>
</TR>
<TR>
<TD>Elemento 5</TD>
<TD>Elemento 6</TD>
</TR>
<TR>
<TD COLSPAN=3><CENTER>Elemento</CENTER></TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

Ejemplo			
Elemento 1	Elemento 2		Total
Elemento 3	Elemento 4	Elemento	
Elemento 5	Elemento 6		
Elemento			

6.11. Frames

6.11.1. Introducción a los frames

Una ventana HTML puede ser dividida en *subventanas* o **frames**, en cada una de las cuales se puede mostrar un documento o un URL distinto. Cada **frame** puede tener un *nombre* al que se puede dirigir el resultado (mediante el atributo TARGET, *objetivo*) de una acción (por ejemplo clicar un hiperlink). El tamaño de los **frames** se puede cambiar interactivamente con el ratón (salvo que se haya eliminado ésta posibilidad explícitamente, al definirlos.).

Puede haber **frames** de contenido fijo o estático (por ejemplo un logo de una empresa, un anuncio) y otros cuyo contenido vaya variando. Un **frame** puede contener un índice y otro (más grande) va mostrando los apartados a los que se hace referencia en el índice.

Se llama **frameset** a una ventana con **frames**. Un **frameset** puede contener otros **framesets**.

Dentro del TAG doble <FRAMESET>... </FRAMESET> sólo puede haber los siguientes contenidos:

1. Otro <FRAMESET>...</FRAMESET>
2. Definición de **frames** con el TAG simple <FRAME>

Para crear **frames** es necesario un fichero HTML con una estructura semejante a la siguiente (obsérvese que la TAG <FRAMESET> sustituye a <BODY>):

```
<HTML>
<HEAD><TITLE>Título</TITLE></HEAD>
<FRAMESET>
    Definición de marcos o frames
</FRAMESET>
</HTML>
```

La TAG <FRAMESET> tiene dos atributos: COLS y ROWS. Ambos atributos admiten una lista de valores separados por comas; estos pueden ser dados en valores absolutos (nº de pixels) o en porcentajes (muy recomendable). En este contexto el asterisco (*) representa el resto del tamaño disponible.

A continuación se exponen algunos ejemplos:

- Para dividir verticalmente la ventana en dos partes separadas por una línea vertical:
`<FRAMESET COLS="20%, 80%">`
- Si se desea reservar dos áreas horizontales de 80 y 160 pixels y una tercera con el resto de la ventana:
`<FRAMESET ROWS="80, 160, *">`
- Para crear una rejilla de 2x3 ventanas:
`<FRAMESET rows="30%,70%" cols="33%,34%,33%">`

Como no se sabe la resolución de la pantalla en la que se ejecutará el browser, conviene siempre utilizar tamaños relativos, o si se utilizan tamaños absolutos, disponer de una partición cuyo tamaño se determina con el asterisco (*). Hay que tener en cuenta que si se utilizan porcentajes relativos que no suman 100, a dichos porcentajes se les aplica un factor de escala de modo que representen correctamente los tamaños relativos de los distintos **frames**.

La TAG <FRAME> es simple y define las propiedades de cada **frame** en el **frameset**. Algunos de sus atributos son:

- <FRAME SRC="URL"> Define el contenido del **frame**, es decir el URL cuyo contenido se mostrará en él. Sin este atributo aparecerá en blanco.

```
<FRAMESET COLS="20%,60%,20%">
    <FRAME SRC="frame1.html">
    <FRAME SRC="frame2.html">
    <FRAME SRC="frame3.html">
</FRAMESET>
```

En este ejemplo se crea una división vertical de la ventana del browser en tres frames, en los que aparecen las páginas: **frame1.html**, **frame2.html** y **frame3.html**.

- NAME="nombre": Define un "nombre" para poder dirigir a ese **frame** contenidos poniendo ese nombre en el atributo TARGET de un hiperlink.
- MARGINWIDTH="n_pixels": es opcional, y determina la distancia horizontal del contenido a los márgenes de los **frames**.
- MARGINHEIGHT="n_pixels": Similar al anterior para la distancia vertical.
- SCROLLING="YES/NO/AUTO": Para que haya o no barras de desplazamiento en el **frame**. Con AUTO las habrá o no según quepa o no el contenido.
- NORESIZE: El usuario no podrá cambiar el tamaño del **frame** con el ratón.

Véase el siguiente ejemplo, correspondiente a la Figura 6.2 con frames anidados:

```
<HTML><HEAD><TITLE>Ejemplo de frames</TITLE></HEAD>
<FRAMESET ROWS="30%,30%,40%">
    <FRAME SRC="PRAC5.HTM" NORESIZE>
    <FRAMESET cols="25%,25%,50%">
        <FRAME SRC="prac1.htm">
        <FRAME SRC="prac2.htm">
        <FRAME SRC="prac3.htm">
```

```
</FRAMESET>
<FRAME SRC="PRAC4.HTM">
</FRAMESET>
</HTML>
```

Para los browsers que no admiten frames se puede emplear la TAG doble <NOFRAMES> y </NOFRAMES>, pudiendo visualizarse el documento sin problemas de forma alternativa:

```
<HTML><HEAD><TITLE> Ejemplo de FRAMES</TITLE></HEAD>
<FRAMESET COLS="80%, 20%">
    <FRAME SRC="Indice.htm">
    <FRAME SRC="Tema1.htm">
</FRAMESET>
<NOFRAMES>
    <A HREF="Tema1.htm">Tema 1</A><BR>
</NOFRAMES>
</HTML>
```

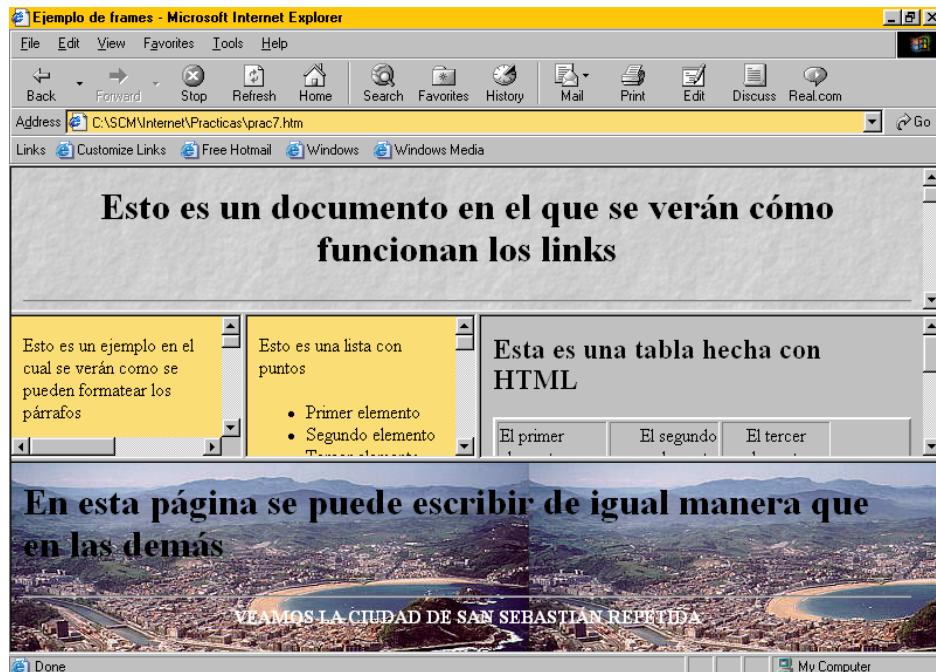


Figura 6.2.

6.11.2. Salida a ventanas o frames concretas de un browser.

Sin **frames**, al clicar en un *link* el nuevo documento aparecía en la misma ventana en que se estaba (o en una ventana nueva si así se había establecido en el browser).

Cuando se utilizan **frames** se puede especificar dónde se desea que aparezca el resultado cuando se clique en un *hiperlink*. Para ello se pueden asignar nombres (mediante el atributo NAME) a las ventanas o **frames** donde se mostrarán los documentos.

A su vez, hay que asignar un atributo TARGET que haga referencia al NAME de un frame o ventana. Si no existe, se creará una ventana con dicho nombre.

- TARGET en la TAG <A>...: Texto
donde el documento indicado en el URL aparecerá en la ventana cuyo nombre sea "nom_ventana".
- TARGET con la TAG <BASE>, para hacer que todos los *links* de un documento se dirijan al mismo TARGET. Así se establece un TARGET por defecto para todo el documento, que puede cambiarse con TARGETs específicos en TAGs tipo <A>. La TAG <BASE> debe incluirse en el <HEAD> del documento.

```
<HEAD>
```

```
<TITLE>Our Products</TITLE>
<BASE href="http://www.aviary.com/intro.html" TARGET="ventana">
</HEAD>
```

- TARGET con la TAG <FORM>, de modo que los resultados de un programa CGI (salida estándar) se dirijan a la ventana indicada. Los formularios <FORM> y los CGI se estudiarán más adelante.

```
<FORM ACTION="URL" TARGET="nom_ventana">
```

6.11.3. Nombres de TARGET

El atributo TARGET puede contener nombres de ventana. Los nombres válidos son los siguientes:

1. Cualquier nombre que empiece por carácter alfanumérico.
2. Los nombres que empiezan por el carácter (_) son especiales:

- TARGET="_blank": Salida dirigida a una ventana nueva en blanco y sin nombre.
- TARGET="_self". Salida dirigida a la propia ventana del *hiperlink*.
- TARGET="_parent". Salida dirigida al **frameset** padre del documento actual. Si no hay padre el resultado es como el de _self.
- TARGET="_top". Destruye todas las FRAMES que haya y la salida se dirige a la ventana principal del browser.

Advertencia: No se admiten **framesets** recursivos (cuando en un **frame** de un **frameset**, se carga el propio **frameset**); si se intenta actúa como FRAME=" _blank".

6.12. Mapas de imágenes o imágenes sensibles

6.12.1. Cómo funciona un mapa de imágenes

Los **mapas de imágenes** son imágenes clicables que permiten al usuario acceder a un URL o a otro dependiendo de dónde se clica sobre dicha imagen. Inicialmente las imágenes sensibles eran procesadas en el servidor remoto http, por lo que se perdía mucho tiempo y además no se indicaba la dirección a la que se iba a acceder. La solución a estos problemas fue procesar las imágenes sensibles en el browser, acelerando el proceso en gran medida y con la ventaja adicional de que el browser informa en la barra de estado acerca de la dirección URL a la que se va a acceder si se clica en ese punto.

6.12.2. Mapas de imágenes procesados por el browser

Un mapa de imágenes se elabora mediante varias TAGs simples <AREA> entre las TAG dobles <MAP>...</MAP>. La TAG <MAP> debe tener un atributo NAME para identificar el mapa de imágenes.

- Las TAGs simples <AREA> definen tantas zonas geométricas activas sobre una imagen como se desee, y pueden tener varios atributos:
 - HREF="URL". Define el URL de destino del área.
 - ALT="texto". Contiene una descripción textual alternativa del área (como en). Se requiere para que los browsers que no tengan gráficos disponibles puedan utilizar satisfactoriamente el Web.
 - SHAPE="forma". Indica la forma de la zona activa; puede tomar los valores: POLY, CIRCLE, RECT (siendo este último el valor por defecto).
 - COORDS="x1,y1,x2,y2,...". Este atributo define una lista de coordenadas (en pixels) separadas por comas que determinan las zonas activas. Las coordenadas son relativas a la esquina superior izquierda de la imagen. El orden y el valor de estas coordenadas depende del atributo SHAPE:
 - En el caso de RECT: X izda., Y sup. X dcha., Y inf.
 - En el caso de CIRCLE: X del centro, Y del centro, radio.
 - En el caso de POLY: X e Y de cada uno de los vértices, siendo el último par de coordenadas coincidente con el primero para cerrar el polígono.

- NOHREF. Este es un atributo opcional que indica que no se debe realizar ninguna acción si el usuario clica en la zona correspondiente. En todo caso, las zonas de la imagen no incluidas en ninguna zona activa definida se consideran de tipo NOHREF.

Para hacer referencia a un determinado mapa de imágenes se utiliza el atributo USEMAP a la TAG , indicando el NAME del mapa deseado como valor del atributo USEMAP. A continuación se expone un ejemplo:

```
<HTML><BODY>
<IMG SRC="concha.gif" USEMAP="#FOTO" ALT="Bahía de San Sebastián">
<MAP NAME="FOTO">
<AREA SHAPE="RECT" COORDS=20,25,155,83 HREF="historia.html" ALT="Historia">
<AREA SHAPE="CIRCLE" COORDS=60,150,40 HREF="plano.html" ALT="Planos">
<AREA SHAPE="POLY" COORDS=106,100,126,170,254,170,254,49,222,100
HREF="fotos.html" ALT="Fotos">
<AREA SHAPE="POLY" COORDS=169,26,169,93,267,26 NOHREF ALT="Idioma">
</MAP>
</BODY></HTML>
```

Para que funcione este código se ha creado el gráfico **concha.gif** incluyendo el dibujo de los contornos de las futuras zonas activas. Se han obtenido las coordenadas de las zonas activas y se han escrito en el atributo COORDS tal y como se ha explicado. Se han creado también los archivos **plano.html**, **historia.html** y **fotos.html**, a los que se accederá tras clicar sobre esas áreas.

En este ejemplo, se supone que la opción del idioma no está aún preparada. Por ello, la TAG AREA no hace referencia a ningún archivo o link. Se ha empleado el atributo NOHREF para indicar expresamente que no se acceda a ningún URL al clicar en esta zona. De este modo, cuando esté ya preparado el archivo **idiomas.html**, no habrá más que cambiar esta parte de código. Finalmente, podrá decirse que la imagen sensible funciona si al desplazar el ratón sin clicar sobre un área activa aparece en la barra de estado la dirección a la que se accederá si se clica sobre ella. Obviamente, esto último no se cumple para las áreas definidas con NOHREF.

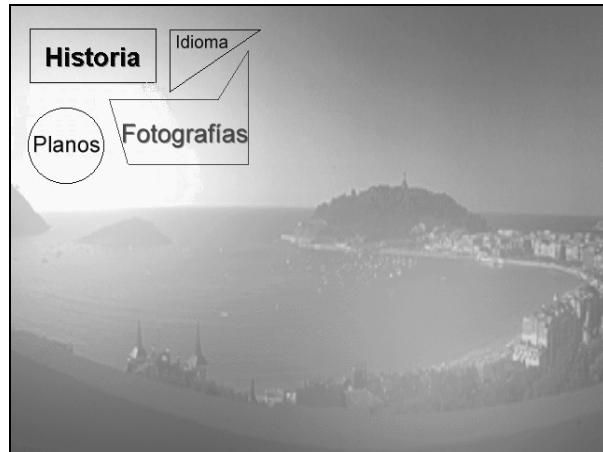


Figura 6.3.

6.13. Editores y conversores HTML

Con un simple editor de texto como **Notepad** se pueden crear los ficheros HTML más sofisticados. Sin embargo, hay que señalar que la tarea se simplifica mucho si se dispone de un buen *editor* o *conversor* especialmente preparado para producir este tipo de ficheros.

Los *editores* permiten introducir de modo automático o semiautomático las TAGs de HTML a medida que se va tecleando el texto. Los hay de dos tipos: editores automáticos **WYSIWYG** (*What You See Is What You Get*) en los que al introducir el texto se muestra tal como se verá en el browser, y editores semiautomáticos más sencillos que simplemente proporcionan ayuda para ir introduciendo los TAGs; de ordinario estos editores permiten llamar al browser preferido pulsando un simple botón, para comprobar cómo se ve el fichero que se está creando. Son editores semiautomáticos **CMED** y **HotDog** y automáticos los editores **Netscape Navigator Gold** y **FrontPage HTML Editor**.

Los *conversores* son programas que convierten los formatos propios de una aplicación (por ejemplo **Word**, **Excel** y **Powerpoint**) a formato HTML. Es posible que en la conversión se pierdan o cambien algunas características de formato, pero eso no resta utilidad a los conversores. Cada vez más, los conversores son programas integrados, es decir, es la propia aplicación original la que tiene la posibilidad

de crear ficheros ***.html**. **Internet Assistant for Word**, **Excel** y **Powerpoint** son conversores integrados en los programas de **Microsoft Office**.

7. Formularios y CGIs

7.1. Formularios (Forms)

Hasta el momento se ha visto cómo el servidor puede crear páginas Web que son visualizadas por medio de un browser en el ordenador del usuario remoto, y cómo se mantiene cierta interactividad mediante el carácter de **hipertexto**, mediante el cual el usuario va solicitando distintos contenidos según sus preferencias. Sin embargo, en lo visto hasta ahora se echa en falta la posibilidad de que el usuario envíe datos al servidor, tales como sugerencias, nombres, datos personales, etc. Esta posibilidad se consigue por medio de los **formularios** (FORMS) que permiten el envío de datos mediante diversos tipos de botones, cuadros de diálogo y ventanas de texto. En este apartado los formularios y sus distintos elementos se estudiarán con un cierto detenimiento. En la Figura 7.1. se presenta un **formulario** con distintos *elementos*: cajas de texto, botones de selección o de opción, listas desplegables, etc.

Figura 7.1.

Los formularios tienen un modo propio de funcionar que conviene entender correctamente, primero de un modo más general y luego más en detalle. Cada uno de los *elementos* del formulario tiene un **nombre**, determinado por la persona que lo ha programado. Ese nombre se asociará posteriormente a la opción elegida o al texto introducido por el usuario. Un elemento particularmente importante (por lo decisivo, más que por su complejidad) es el botón **Enviar** (o *Submit*, *Send*, *O.K.*, o como se le haya querido llamar), cuya misión es dar por concluida la recogida de datos en el browser y enviar al servidor toda la información que el usuario haya introducido. Cuando se pulsa este botón se envía al servidor (en la forma que más adelante se verá) una larga cadena de caracteres que contiene los nombres de todos los elementos del formulario junto con la información que el usuario ha introducido en cada uno de ellos.

¿Qué hace el servidor con toda la información que le llega de un formulario debidamente cumplimentado? Pues algo muy sencillo: arrancar un programa cuyo nombre está especificado en el propio formulario, pasarle toda la información que ha recibido, recoger la información que dicho programa le envía como respuesta, y enviarla al cliente o browser que rellenó el formulario. Además de responder al cliente, este programa podrá realizar cualquier otra acción, como por ejemplo guardar la información recibida en un fichero o en una base de datos. Ya se ve que en todo este proceso el servidor actúa como una especie de intermediario entre el browser y el programa que recibe la información del formulario y la procesa. La forma en la que el servidor se entiende con el programa al que llama para que procese los datos del formulario recibe el nombre de CGI (*Common Gateway Interface*). La Figura 7.2. representa de forma simbólica todo este proceso.

Así pues todo formulario se define mediante la doble TAG **<FORM>...</FORM>**, dentro de las cuales se definirán los distintos elementos. Al crear un formulario se debe definir el **método de envío de datos** (GET o POST) que el servidor utilizará para enviar al programa CGI los datos que debe procesar. También se debe especificar la **acción** que el servidor deberá emprender cuando reciba los datos, y que será

arrancar dicho programa CGI cuyo nombre debe conocer. Estas dos tareas se definen, respectivamente, con los atributos METHOD y ACTION.

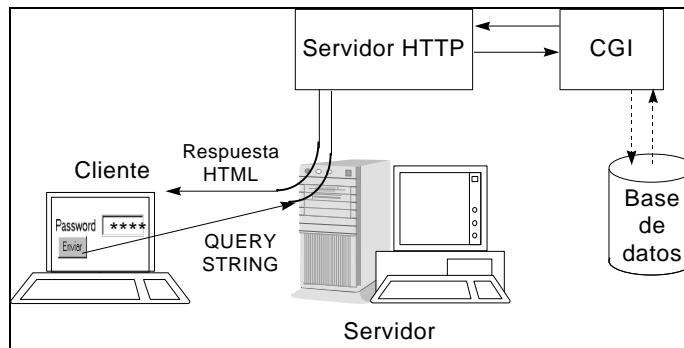


Figura 7.2.

A continuación se presenta un ejemplo simple que permite ver cómo se define un formulario y cómo aparece en el browser. El código HTML es el siguiente:

```

<FORM ACTION="/cgi-bin/form1.exe" METHOD="GET">
    Introduzca su nombre:
    <INPUT TYPE="text" NAME="nombre"><br>
    <INPUT TYPE="SUBMIT" VALUE="Enviar">
</FORM>
    
```

En la Figura 7.3. aparece representado el formulario correspondiente, que sólo tiene dos elementos: un *cuadro de texto* para que el usuario teclee el nombre y un *botón* para enviar los datos al servidor cuando el nombre haya sido ya introducido. Todos los elementos de un formulario que recogen información se definen con la TAG simple **<INPUT>**. El atributo TYPE define el tipo de elemento de que se trata (si se omite, por defecto se supone que el tipo es **text**). A cada elemento (excepto a los singulares -que no se pueden repetir- como el botón **Enviar**) hay que añadirle un atributo NAME que defina el nombre con el que se identificará luego su contenido.



Figura 7.3.

En el ejemplo anterior, el cuadro de texto se ha definido con la TAG INPUT; en esa TAG se ha introducido un parámetro NAME definiendo que ese cuadro se llama "*nombre*".

Se ha introducido el atributo TYPE="SUBMIT" para el segundo elemento del formulario. Este elemento es un botón cuya función es enviar los datos de los elementos del formulario al programa CGI que lo va a procesar. Dicho programa se especifica mediante el atributo ACTION, y resulta ser "/cgi-bin/form1.exe". En general **cgi-bin** es un directorio localizado en el servidor, que contiene los programas CGI.

También se puede crear un botón (llamado en inglés RESET) que reinicie el formulario a su estado inicial, anulando todos los cambios que hubiera podido introducir el usuario. La manera de crearlo se verá en el siguiente ejemplo, que contiene algunos elementos y atributos nuevos con los que se trata de recoger los datos del domicilio de una persona. La Figura 7.4. muestra como se ve en el browser el formulario definido. Gracias a la TAG <PRE>...</PRE> que impone paso constante y respeta los espacios en blanco se consigue que los cuadros de texto queden alineados. El atributo NAME permite definir un nombre para cada elemento. El atributo SIZE determina la longitud del cuadro de texto, mientras que MAXLENGTH limita el número de caracteres que se pueden introducir en cada cuadro. Inicialmente todos

los cuadros de texto están en blanco, tal como se ve en la figura 7.4. Si el usuario introduce datos y pulsa luego el botón **Borrar**, el formulario volverá a la condición inicial. Los botones SUBMIT y BORRAR no necesitan NAME porque quedan identificados por su atributo TYPE. Por su parte los cuadros de texto no necesitan atributo TYPE porque el tipo por defecto es TEXT.

```
<FORM ACTION="/cgi-bin/form3" METHOD="POST">
<PRE>
Nombre:      <INPUT NAME="nombre" SIZE=30>
Calle:        <INPUT NAME="calle" SIZE=40>
Localidad:    <INPUT NAME="localid" SIZE=35>
Provincia:    <INPUT NAME="prov" SIZE=25>
Cód. Postal:  <INPUT NAME="cp" SIZE=5 MAXLENGTH=5>
</PRE>
<INPUT TYPE="SUBMIT" VALUE="Enviar">
<INPUT TYPE="RESET" VALUE="Borrar">
</FORM>
```

Figura 7.4.

Considérese un nuevo ejemplo correspondiente al formulario que se muestra en la Figura 7.5. En él se ve que también se pueden crear **áreas de texto** en las cuales no se limita la cantidad de texto a introducir, y que pueden servir para pedir sugerencias u opiniones. La forma de crearlas es con la TAG doble **<TEXTAREA>...</TEXTAREA>**. Los parámetros que se le envían son el nombre (atributo NAME) y los números de filas y de columnas de la ventana (atributos ROWS=n y COLS=m). En este caso la TAG doble **<TEXTAREA>** sustituye a la TAG simple **<INPUT>**. Más adelante se verán otros TAGs alternativas para crear distintos elementos del formulario. A continuación se muestra un ejemplo (Figura 7.5).

```
<H2>Sugerencias y críticas</H2>
<P><FORM ACTION="/cgi-bin/form3" METHOD="POST">
  Por favor, introduzca cualquier sugerencia o crítica positiva:<BR>
  <TEXTAREA ROWS=5 COLS=30 NAME="positivo">Me encanta este Web porque...
  </TEXTAREA></P>
  <P>Introduzca los comentarios negativos:<BR>
  <TEXTAREA ROWS=2 COLS=15 NAME="negativo"></TEXTAREA></P>
  <P><INPUT TYPE="SUBMIT" VALUE="Enviar">
</FORM></P>
```

Sugerencias y críticas

Por favor, introduzca cualquier sugerencia o crítica positiva:

Me encanta este Web porque...

Introduzca los comentarios negativos:

...

Figura 7.5.

Introduzca el nombre del usuario:

Introduzca código de acceso:

Figura 7.6.

Indique qué sistemas operativos conoce:

Unix
 OS/2
 MacOS
 DOS
 Windows NT

Figura 7.7.

En el siguiente ejemplo se va a ver que es posible crear cuadros de texto en los que lo que se escribe no aparezca (que aparezca como caracteres *). Estos elementos se emplean para que el usuario introduzca un **password**. En la Figura 7.6. se muestra el resultado del código HTML que sigue:

```
<FORM>
  Introduzca el nombre del usuario: <INPUT NAME="login" SIZE=25><BR>
  Introduzca código de acceso:
  <INPUT TYPE="PASSWORD" NAME="acceso" SIZE=6 MAXLENGTH=10><P>
  <INPUT TYPE="SUBMIT" VALUE="Enviar">
  <INPUT TYPE="RESET" VALUE="Borrar">
</FORM>
```

Como se puede ver, el modo de crear este tipo de cuadros de texto es análogo a los anteriores: únicamente hay que introducir el atributo TYPE="password".

Una de las opciones más interesantes de los formularios es la de poder crear **casillas de verificación**. En la Figura 7.7 aparece un ejemplo de este tipo de elemento. Para poder crear estas casillas se debe utilizar la TAG INPUT, poniendo el atributo TYPE="CHECKBOX" y el mismo NAME para todas las casillas del grupo. La característica de estas casillas es que se pueden seleccionar varias a la vez. Si al definirlas se introduce además el atributo CHECKED, aparecerán inicialmente seleccionadas (Pese a que luego puedan no seleccionarse). El código HTML necesario para crear este formulario es el que sigue a continuación:

```
<FORM ACTION="/cgi-bin/form" METHOD="POST">
  Indique qué sistemas operativos conoce:<P>
  <INPUT TYPE="checkbox" NAME="sisoper" VALUE="Unix" CHECKED>Unix<BR>
  <INPUT TYPE="checkbox" NAME="sisoper" VALUE="OS/2">OS/2<BR>
  <INPUT TYPE="checkbox" NAME="sisoper" VALUE="MacOS">MacOS<BR>
  <INPUT TYPE="checkbox" NAME="sisoper" VALUE="DOS" CHECKED>DOS<BR>
  <INPUT TYPE="checkbox" NAME="sisoper" VALUE="WinNT" CHECKED>Windows NT<BR>
  <INPUT TYPE="SUBMIT" VALUE="Enviar">
  <INPUT TYPE="RESET" VALUE="Borrar">
</FORM>
```

En el siguiente ejemplo se va a ver que también se pueden crear **botones de radio**, que son similares a los CHECKBOX, pero con la condición de que sólo se puede seleccionar uno de ellos a la vez. La Figura 7.8. muestra el resultado en el browser del código que se incluye a continuación:

```
<FORM ACTION="/cgi-bin/form" METHOD="POST">
  Indique su estado civil:<P>
  <INPUT TYPE="radio" NAME="estado" VALUE="Soltero">Soltero/a <BR>
  <INPUT TYPE="radio" NAME="estado" VALUE="Casado">Casado/a <BR>
  <INPUT TYPE="radio" NAME="estado" VALUE="Viudo">Viudo/a <BR>
  <INPUT TYPE="radio" NAME="estado" VALUE="Otros">Otros <BR>
```

```
<INPUT TYPE="SUBMIT" VALUE="Enviar">
<INPUT TYPE="RESET" VALUE="Borrar">
</FORM>
```

Finalmente se presentan ejemplos de otra de las opciones que permiten los formularios: las **ventanas de selección desplegables** y las **ventanas de scroll** (o ventanas con barras de deslizamiento). Las primeras están basadas en una TAG doble llamado **<SELECT>...</SELECT>**. En el contenido de esta TAG se pueden incluir tantos TAGs simples **<OPTION>** como se deseé. El TAG **<SELECT>** tiene el atributo NAME, lo que no ocurre con las TAGs **<OPTION>**. Cada TAG **<OPTION>** contiene un atributo VALUE que describe el texto que aparecerá en la ventana.

Indique su estado civil:

Soltero/a
 Casado/a
 Viudo/a
 Otros

Figura 7.8.

Indique cuál es el medio de transporte que usa con mayor frecuencia para ir al trabajo:

Coche particular ▾
 Coche particular
 Autobús
 Taxi
 Tren
Metro
 Bicicleta

Figura 7.9.

El aspecto visual del primero de los ejemplos se muestra en la Figura 7.9 y su código HTML:

```
<FORM>
  Indique cuál es el medio de<BR>
  transporte que usa con mayor<BR>
  frecuencia para ir al trabajo:<P>
  <SELECT NAME="Transp">
    <OPTION VALUE="coche"> Coche particular </OPTION>
    <OPTION VALUE="bus"> Autobús </OPTION>
    <OPTION VALUE="taxi"> Taxi </OPTION>
    <OPTION VALUE="tren"> Tren </OPTION>
    <OPTION VALUE="metro"> Metro </OPTION>
    <OPTION VALUE="bici"> Bicicleta </OPTION>
  </SELECT>
  <INPUT TYPE="SUBMIT" VALUE="Enviar">
</FORM>
```

Como se puede ver cada una de las opciones de la ventana se introduce con la TAG simple **<OPTION VALUE="nombre">**, y es necesario que todas las opciones estén entre las TAGs **<SELECT>** y **</SELECT>**

En el siguiente ejemplo se va a utilizar, en vez de una **ventana desplegable**, una ventana con barras de deslizamiento o **ventana de scroll**. En la Figura 7.10 se muestra el resultado del código que sigue:

```
<FORM>
  Indique cuál es el medio de<BR>
  transporte que usa con mayor<BR>
  frecuencia para ir al trabajo:<P>
  <SELECT NAME="Transp" SIZE=3>
    <OPTION VALUE="coche"> Coche particular </OPTION>
    <OPTION VALUE="bus"> Autobús </OPTION>
    <OPTION VALUE="taxi"> Taxi </OPTION>
    <OPTION VALUE="tren"> Tren </OPTION>
    <OPTION VALUE="metro"> Metro </OPTION>
    <OPTION VALUE="bici"> Bicicleta </OPTION>
  </SELECT>
  <INPUT TYPE="SUBMIT" VALUE="Enviar">
</FORM>
```

Indique cuál es el medio de transporte que usa con mayor frecuencia para ir al trabajo:

Autobús
Taxi
Tren

Figura 7.10.

Indique las áreas que sean de su interés:

Internet
Telecomunicaciones
Sistemas Operativos
Software

Figura 7.11.

La única diferencia respecto al ejemplo anterior es el atributo SIZE que define el tamaño vertical (número de líneas) de la ventana. Si SIZE=1 la ventana será desplegable, y en caso contrario aparecerá una ventana con barras de deslizamiento.

En los dos ejemplos anteriores sólo se podía seleccionar uno de las opciones de la ventana. En el siguiente ejemplo se va a crear una **ventana de selección múltiple**. El resultado se muestra en la Figura 7.11. y el código HTML es como sigue:

```
<FORM>
  Indique las áreas que sean<BR>
  de su interés:<P>
  <SELECT NAME="Areas" SIZE=4 MULTIPLE>
    <OPTION VALUE="lan"> Redes de Área Local
    <OPTION VALUE="inet"> Internet
    <OPTION VALUE="teleco"> Telecomunicaciones
    <OPTION VALUE="sop"> Sistemas Operativos
    <OPTION VALUE="soft"> Software
    <OPTION VALUE="hard"> Hardware
  </SELECT>
  <INPUT TYPE="SUBMIT" VALUE="Enviar" >
</FORM>
```

Ni que decir tiene que también es posible crear ventanas desplegables de selección múltiple: pruébese copiando éste mismo código eliminando el parámetro SIZE=4 o, simplemente, haciendo SIZE= 1) En los ejemplos anteriores se han presentado las posibilidades abiertas por los formularios. En el apartado siguiente se considerará cómo se debe procesar esta información cuando llega al servidor.

7.2. Programas CGI

Ya se ha visto en el apartado anterior cómo crear formularios con los que recoger datos de diverso tipo para enviarlos al servidor, pero ¿cómo se envían y cómo se procesan estos datos? La respuesta está en que los datos se envían como cadena de caracteres añadida a un URL como *query string*, precedida con un carácter (?); para procesarlos se utilizan los programas CGI (*Common Gateway Interface*), que el servidor arranca y a los que pasa los datos recibidos del browser.

Los CGI son programas hechos por lo general en **Perl** (lenguaje interpretado muy popular en el entorno UNIX, pero que también existe en **Windows NT**) o en **C/C++**.

A grandes rasgos un programa CGI debe realizar las siguientes tareas:

- Recibir los datos que son enviados por el browser e interpretarlos, separando la entrada correspondiente a cada elemento del formulario.
- Realizar las tareas necesarias para procesar esos datos, que de ordinario consistirán en almacenarlos en algún archivo o base de datos y enviar una respuesta al browser donde se encuentra el usuario. Esta respuesta debe tener la forma de página HTML incluyendo todos los elementos necesarios (<HEAD>, <BODY>, etc.).

Para que un programa CGI funcione correctamente debe estar en un directorio /cgi-bin/ del servidor HTTP (servidor Web). En caso que se utilice Visual C++ (u otro compilador compatible) es importante, para que el ejecutable CGI funcione correctamente, el compilarlo para que funcione en modo MS-DOS (o modo

Consola). Para un programa de este tipo la *entrada de datos estándar* es el teclado y la *salida de datos estándar* es la pantalla. Para un programa CGI el servidor HTTP asume el papel de entrada y salida de datos estándar.

Parte 3

Aplicaciones Web

8. Servlets

8.1. Clientes y Servidores

8.1.1. Clientes (clients)

Por su versatilidad y potencialidad, en la actualidad la mayoría de los usuarios de **Internet** utilizan en sus comunicaciones con los servidores de datos, los **browsers** o **navegadores**. Esto no significa que no puedan emplearse otro tipo de programas como clientes e-mail, news, etc. para aplicaciones más específicas. De hecho, los browsers más utilizados incorporan lectores de mail y de news.

En la actualidad los browsers más extendidos son **Netscape Communicator** y **Microsoft Internet Explorer**. Ambos acaparan una cuota de mercado que cubre prácticamente a todos los usuarios.

A pesar de que ambos cumplen con la mayoría de los estándares aceptados en la **Internet**, cada uno de ellos proporciona soluciones adicionales a problemas más específicos. Por este motivo, muchas veces será necesario tener en cuenta qué tipo de browser se va a comunicar con un servidor, pues el resultado puede ser distinto dependiendo del browser empleado, lo cual puede dar lugar a errores.

Ambos browsers soportan **Java**, lo cual implica que disponen de una **Java Virtual Machine** en la que se ejecutan los ficheros ***.class** de las **Applets** que traen a través de **Internet**. Netscape es más fiel al estándar de **Java** tal y como lo define **Sun**, pero ambos tienen la posibilidad de sustituir la **Java Virtual Machine** por medio de un mecanismo definido por **Sun**, que se llama **Java Plug-in** (los **plug-ins** son aplicaciones que se ejecutan controladas por los browsers y que permiten extender sus capacidades, por ejemplo para soportar nuevos formatos de audio o video).

8.1.2. Servidores (servers)

Los **servidores** son programas que se encuentran permanentemente esperando a que algún otro ordenador realice una solicitud de conexión. En un mismo ordenador es posible tener simultáneamente servidores de los distintos servicios anteriormente mencionados (**HTTP**, **FTP**, **TELNET**, etc.). Cuando a dicho ordenador llega un requerimiento de servicio enviado por otro ordenador de la red, se interpreta el tipo de llamada, y se pasa el control de la conexión al **servidor** correspondiente a dicho requerimiento. En caso de no tener el **servidor** adecuado para responder a la comunicación, está será rechazada.

Como ya se ha apuntado, no todos los servicios actúan de igual manera. Algunos, como **TELNET** y **FTP**, una vez establecida la conexión, la mantienen hasta que el cliente o el servidor explícitamente la cortan. Por ejemplo, cuando se establece una conexión con un servidor de **FTP**, los dos ordenadores se mantienen en contacto hasta que el cliente cierra la conexión mediante el comando correspondiente (**quit**, **exit**, ...) o pase un tiempo establecido en la configuración del servidor **FTP** o del propio cliente, sin ninguna actividad entre ambos.

La comunicación a través del protocolo **HTTP** es diferente, ya que es necesario establecer una comunicación o conexión distinta para cada elemento que se desea leer. Esto significa que en un documento **HTML** con 10 imágenes son necesarias 11 conexiones distintas con el servidor **HTTP**, esto es, una para el texto del documento **HTML** con las **tags** y las otras 10 para traer las imágenes referenciadas en el documento **HTML**.

La mayoría de los usuarios de **Internet** son **clientes** que acceden mediante un **browser** a los distintos **servidores WWW** presentes en la red. El servidor no permite acceder indiscriminadamente a todos sus ficheros, sino únicamente a determinados directorios y documentos previamente establecidos por el **administrador** de dicho servidor.

8.2. Tendencias Actuales para las aplicaciones en Internet

En la actualidad, la mayoría de aplicaciones que se utilizan en entornos empresariales están construidos en torno a una arquitectura **cliente-servidor**, en la cual uno o varios computadores (generalmente de una potencia considerable) son los **servidores**, que proporcionan servicios a un número mucho más grande de **clientes** conectados a través de la red. Los **clientes** suelen ser PCs de propósito general, de ordinario menos potentes y más orientados al usuario final. A veces los servidores son intermediarios entre los clientes y otros servidores más especializados (por ejemplo los grandes servidores de bases de datos corporativos basados en **mainframes** y/o sistemas **Unix**. En este caso se habla de *aplicaciones de varias capas*).

Con el auge de **Internet**, la arquitectura **cliente-servidor** ha adquirido una mayor relevancia, ya que la misma es el principio básico de funcionamiento de la **World Wide Web**: un usuario que mediante un **browser (cliente)** solicita un servicio (páginas **HTML**, etc.) a un computador que hace las veces de **servidor**. En su concepción más tradicional, los servidores **HTTP** se limitaban a enviar una página **HTML** cuando el usuario la requería directamente o clicaba sobre un enlace. La interactividad de este proceso era mínima, ya que el usuario podía pedir ficheros, pero no enviar sus datos personales de modo que fueran almacenados en el servidor u obtuviera una respuesta personalizada. La Figura 8.1 representa gráficamente este concepto.

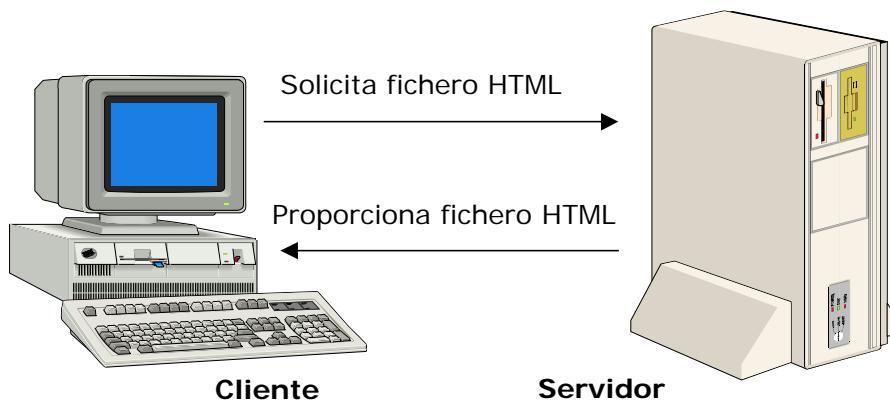


Figura 8.1. Arquitectura cliente-servidor tradicional.

Desde esa primera concepción del servidor **HTTP** como mero servidor de ficheros **HTML** el concepto ha ido evolucionando en dos direcciones complementarias:

1. Añadir más inteligencia en el **servidor**, y
2. Añadir más inteligencia en el **cliente**.

Las formas más extendidas de añadir inteligencia a los clientes (a las páginas **HTML**) han sido **Javascript** y las **applets de Java**. **Javascript** es un lenguaje relativamente sencillo, interpretado, cuyo código fuente se introduce en la página **HTML** por medio de los tags <SCRIPT> ... </SCRIPT>. Las **applets de Java** tienen mucha más capacidad de añadir inteligencia a las páginas **HTML** que se visualizan en el browser, ya que son verdaderas clases de **Java** (ficheros *.class) que se cargan y se ejecutan en el cliente.

De cara a estos apuntes tienen mucho más interés los caminos seguidos para añadir más inteligencia en el servidor **HTTP**. La primera y más empleada tecnología ha sido la de los **programas CGI (Common Gateway Interface)**, unida a los **formularios HTML**.

Los **formularios HTML** permiten de alguna manera invertir el sentido del flujo de la información. Cumplimentando algunos campos con cajas de texto, botones de opción y de selección, el usuario puede definir sus preferencias o enviar sus datos al servidor. Cuando en un formulario **HTML** se pulsa en el botón **Enviar** (o nombre equivalente, como **Submit**) los datos tecleados por el cliente se envían al servidor para su procesamiento.

¿Cómo recibe el servidor los datos de un formulario y qué hace con ellos? Éste es el problema que tradicionalmente han resuelto los **programas CGI**. Cada formulario lleva incluido un campo llamado

Action con el que se asocia el nombre de programa en el servidor. El servidor arranca dicho programa y le pasa los datos que han llegado con el formulario. Existen dos formas principales de pasar los datos del formulario al **programa CGI**:

1. Por medio de una variable de entorno del sistema operativo del servidor, de tipo String (método **GET**)
2. Por medio de un flujo de caracteres que llega a través de la entrada estándar (*stdin* o *System.in*), que de ordinario está asociada al teclado (método **POST**).

En ambos casos, la información introducida por el usuario en el formulario llega en la forma de una única cadena de caracteres en la que el nombre de cada campo del formulario se asocia con el valor asignado por el usuario, y en la que los blancos y ciertos caracteres especiales se han sustituido por secuencias de caracteres de acuerdo con una determinada codificación. Más adelante se verán con más detenimiento las reglas que gobiernan esta transmisión de información. En cualquier caso, lo primero que tiene que hacer el **programa CGI** es decodificar esta información y separar los valores de los distintos campos. Después ya puede realizar su tarea específica: escribir en un fichero o en una base de datos, realizar una búsqueda de la información solicitada, realizar comprobaciones, etc. De ordinario, el **programa CGI** termina enviando al cliente (el navegador desde el que se envió el formulario) una página **HTML** en la que le informa de las tareas realizadas, le avisa de si se ha producido alguna dificultad, le reclama algún dato pendiente o mal cumplimentado, etc. La forma de enviar esta página **HTML** al cliente es a través de la salida estándar (*stdout* o *System.out*), que de ordinario suele estar asociada a la pantalla. La página **HTML** tiene que ser construida elemento a elemento, de acuerdo con las reglas de este lenguaje. No basta enviar el contenido: hay que enviar también todas y cada una de las **tags**. En un próximo apartado se verá un ejemplo completo.

En principio, los **programas CGI** pueden estar escritos en cualquier lenguaje de programación, aunque en la práctica se han utilizado principalmente los lenguajes **Perl**⁵ y **C/C++**. Un claro ejemplo de un **programa CGI** sería el de un formulario en el que el usuario introdujera sus datos personales para registrarse en un sitio web. El **programa CGI** recibiría los datos del usuario, introduciéndolos en la base de datos correspondiente y devolviendo al usuario una página **HTML** donde se le informaría de que sus datos habían sido registrados. La Figura 8.2 muestra el esquema básico de funcionamiento de los **programas CGI**.

Es importante resaltar que estos procesos tienen lugar en el servidor. Esto a su vez puede resultar un problema, ya que al tener múltiples clientes conectados al servidor, el **programa CGI** puede estar siendo llamado simultáneamente por varios clientes, con el riesgo de que el servidor se llegue a saturar. Téngase en cuenta que cada vez que se recibe un requerimiento se arranca una nueva copia del **programa CGI**. Existen otros riesgos adicionales que se estudiarán más adelante.

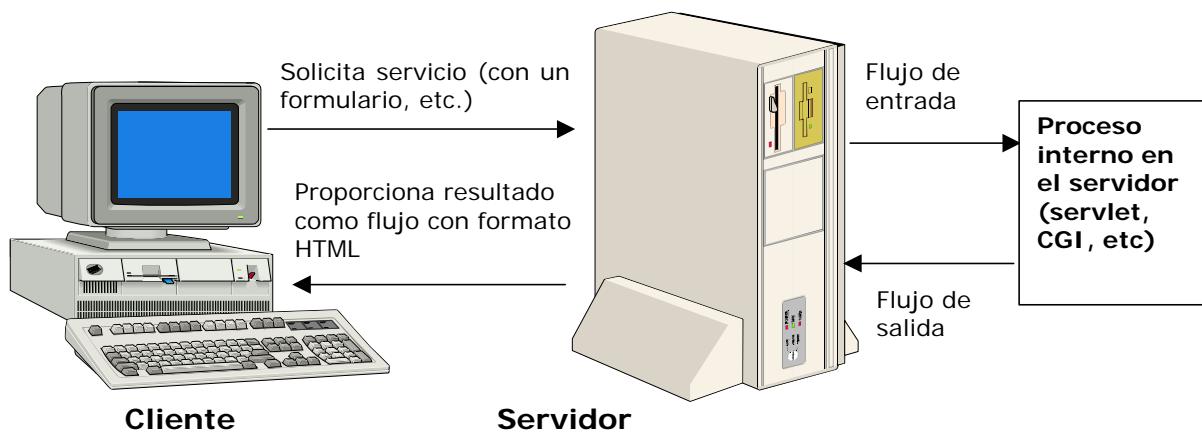


Figura 8.2. Arquitectura cliente-servidor interactiva para la WEB.

⁵ PERL es un lenguaje interpretado procedente del entorno Unix (aunque también existe en Windows NT), con grandes capacidades para manejar texto y cadenas de caracteres.

El objetivo de este capítulo es el estudio de la alternativa que **Java** ofrece a los **programas CGI**: los **servlets**, que son a los servidores lo que los **applets** a los browsers. Se podría definir un **servlet** como **un programa escrito en Java que se ejecuta en el marco de un servicio de red, (un servidor HTTP, por ejemplo), y que recibe y responde a las peticiones de uno o más clientes.**

8.3. Diferencias entre las tecnologías CGI y Servlet

La tecnología **Servlet** proporciona las mismas ventajas del lenguaje **Java** en cuanto a **portabilidad** ("write once, run anywhere") y **seguridad**, ya que un **servlet** es una **clase de Java** igual que cualquier otra, y por tanto tiene en ese sentido todas las características del lenguaje. Esto es algo de lo que carecen los **programas CGI**, ya que hay que compilarlos para el sistema operativo del servidor y no disponen en muchos casos de técnicas de comprobación dinámica de errores en tiempo de ejecución.

Otra de las principales ventajas de los **servlets** con respecto a los **programas CGI**, es la del rendimiento, y esto a pesar de que **Java** no es un lenguaje particularmente rápido. Mientras que los es necesario cargar los **programas CGI** tantas veces como peticiones de servicio existan por parte de los clientes, los **servlets**, una vez que son llamados por primera vez, **quedan activos en la memoria del servidor hasta que el programa que controla el servidor los desactiva**. De esta manera se minimiza en gran medida el tiempo de respuesta.

Además, los **servlets** se benefician de la gran capacidad de **Java** para ejecutar métodos en ordenadores remotos, para conectar con bases de datos, para la seguridad en la información, etc. Se podría decir que las **clases estándar de Java** ofrecen resueltos mucho problemas que con otros lenguajes tiene que resolver el programador.

8.4. Características de los servlets

Además de las características indicadas en el apartado anterior, los **servlets** tienen las siguientes características:

1. Son independientes del servidor utilizado y de su sistema operativo, lo que quiere decir que a pesar de estar escritos en **Java**, el servidor puede estar escrito en cualquier lenguaje de programación, obteniéndose exactamente el mismo resultado que si lo estuviera en **Java**.
2. Los **servlets** pueden llamar a otros **servlets**, e incluso a métodos concretos de otros **servlets**. De esta forma se puede distribuir de forma más eficiente el trabajo a realizar. Por ejemplo, se podría tener un **servlet** encargado de la interacción con los clientes y que llamaría a otro **servlet** para que a su vez se encargara de la comunicación con una base de datos. De igual forma, los **servlets** permiten **redireccionar** peticiones de servicios a otros **servlets** (en la misma máquina o en una máquina remota).
3. Los **servlets** pueden obtener fácilmente información acerca del **cliente** (la permitida por el protocolo **HTTP**), tal como su dirección **IP**, el **puerto** que se utiliza en la llamada, el método utilizado (**GET**, **POST**, ...), etc.
4. Permiten además la utilización de **cookies** y **sesiones**, de forma que se puede guardar información específica acerca de un usuario determinado, personalizando de esta forma la interacción cliente-servidor. Una clara aplicación es **mantener la sesión** con un cliente.
5. Los **servlets** pueden actuar como enlace entre el cliente y una o varias **bases de datos** en arquitecturas *cliente-servidor de 3 capas* (si la base de datos está en un servidor distinto).
6. Asimismo, pueden realizar tareas de **proxy** para un **applet**. Debido a las restricciones de seguridad, un **applet** no puede acceder directamente por ejemplo a un servidor de datos localizado en cualquier máquina remota, pero el **servlet** sí puede hacerlo de su parte.
7. Al igual que los **programas CGI**, los **servlets** permiten la generación dinámica de código **HTML** dentro de una propia página **HTML**. Así, pueden emplearse **servlets** para la creación de contadores, banners, etc.

8.5. Servidor de aplicaciones

Para ejecutar una aplicación java, es necesario el interprete java.exe. Para ejecutar un servlet es necesario un servidor de aplicaciones (application server), también conocido como contenedor de servlets (servlet container).

El servidor de aplicaciones es quién recoge la petición de un cliente y pasa esta llamada al servlet que se haya configurado en el servidor para responder. El servlet realiza las acciones programadas para dicha llamada y devuelve una página html de resultado al servidor que este envía al cliente que realizó la solicitud.

Los servidores de aplicaciones implementan las especificaciones de los servlets, recogidas en su API. El **API** de los servlets consta de dos **packages** cuyo funcionamiento será estudiado en detalle en el siguiente capítulo, y que se encuentran contenidos en **javax.servlet** y **javax.servlet.http**. Este último es una particularización del primero para el caso del protocolo **HTTP**, que es el que será utilizado en este manual, al ser el más extendido en la actualidad. Mediante este diseño lo que se consigue es que se mantenga una puerta abierta a la utilización de otros protocolos que existen en la actualidad (**FTP**, **POP**, **SMTP**, etc.), o vayan siendo utilizados en el futuro. Estos **packages** están almacenados en un fichero **JAR**.

Existen distintos servidores de aplicaciones. Sun distribuye GlassFish Application Server y la fundación Apache el servidor Apache Tomcat Servlet/JSP container.

La instalación de estos servidores proporciona los paquetes necesarios para programar y ejecutar los servlets.

8.5.1. Visión general del Servlet API

Es importante adquirir cuanto antes una visión general del **API** (*Application Programming Interface*) de los servlets, de qué clases e interfaces la constituyen y de cuál es la relación entre ellas.

El **Servlet API** contiene dos paquetes: **javax.servlet** y **javax.servlet.http**. Todas las clases e interfaces que hay que utilizar en la programación de **servlets** están en estos dos paquetes.

La relación entre las clases e interfaces de **Java**, muy determinada por el concepto de **herencia**, se entiende mucho mejor mediante una representación gráfica tal como la que puede verse en la Figura 8.3. En dicha figura se representan las **clases** con letra normal y las **interfaces** con cursiva.

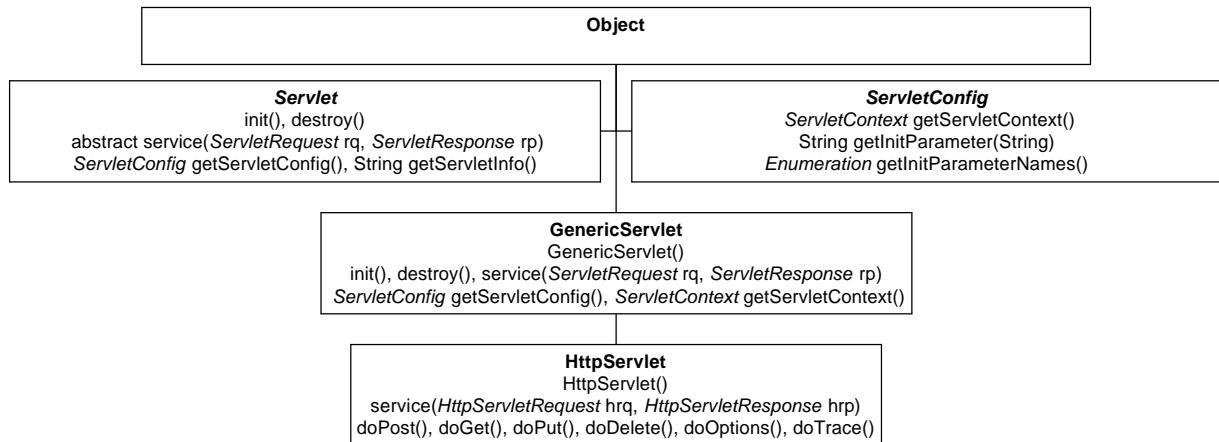


Figura 8.3. Jerarquía y métodos de las principales clases para crear servlets.

La clase **GenericServlet** es una clase **abstract** puesto que su método **service()** es **abstract**. Esta clase implementa dos interfaces, de las cuales la más importante es la interface **Servlet**.

La interface **Servlet** declara los métodos más importantes de cara a la vida de un servlet: **init()** que se ejecuta sólo al arrancar el servlet; **destroy()** que se ejecuta cuando va a ser destruido y **service()** que se ejecutará cada vez que el servlet deba atender una solicitud de servicio.

Cualquier clase que derive de **GenericServlet** deberá definir el método **service()**. Es muy interesante observar los dos argumentos que recibe este método, correspondientes a las interfaces **ServletRequest** y **ServletResponse**. La primera de ellas referencia a un objeto que describe por completo la solicitud de

servicio que se le envía al servlet. Si la solicitud de servicio viene de un formulario HTML, por medio de ese objeto se puede acceder a los nombres de los campos y a los valores introducidos por el usuario; puede también obtenerse cierta información sobre el cliente (ordenador y browser). El segundo argumento es un objeto con una referencia de la interface **ServletResponse**, que constituye el camino mediante el cual el método **service()** se conecta de nuevo con el cliente y le comunica el resultado de su solicitud. Además, dicho método deberá realizar cuantas operaciones sean necesarias para desempeñar su cometido: escribir y/o leer datos de un fichero, comunicarse con una base de datos, etc. El método **service()** es realmente el corazón del servlet.

En la práctica, salvo para desarrollos muy especializados, todos los servlets deberán construirse a partir de la clase **HttpServlet**, sub-clase de **GenericServlet**.

La clase **HttpServlet** ya no es **abstract** y dispone de una implementación o definición del método **service()**. Dicha implementación detecta el tipo de servicio o método **HTTP** que le ha sido solicitado desde el browser y llama al método adecuado de esa misma clase (**doPost()**, **doGet()**, etc.). Cuando el programador crea una sub-clase de **HttpServlet**, por lo general no tiene que redefinir el método **service()**, sino uno de los métodos más especializados (normalmente **doPost()**), que tienen los mismos argumentos que **service()**: dos objetos referenciados por las interfaces **ServletRequest** y **ServletResponse**.

En la Figura 8.3 aparecen también algunas otras **interfaces**, cuyo papel se resume a continuación.

1. La interface **ServletContext** permite a los **servlets** acceder a información sobre el entorno en que se están ejecutando.
2. La interface **ServletConfig** define métodos que permiten pasar al **servlet** información sobre sus parámetros de inicialización.
3. La interface **ServletRequest** permite al método **service()** de **GenericServlet** obtener información sobre una petición de servicio recibida de un cliente. Algunos de los datos proporcionados por **GenericServlet** son los nombres y valores de los parámetros enviados por el formulario HTML y una **input stream**.
4. La interface **ServletResponse** permite al método **service()** de **GenericServlet** enviar su respuesta al cliente que ha solicitado el servicio. Esta interface dispone de métodos para obtener un **output stream** o un **writer** con los que enviar al cliente datos binarios o caracteres, respectivamente.
5. La interface **HttpServletRequest** deriva de **ServletRequest**. Esta interface permite a los métodos **service()**, **doPost()**, **doGet()**, etc. de la clase **HttpServlet** recibir una petición de servicio **HTTP**. Esta interface permite obtener información del header de la petición de servicio **HTTP**.
6. La interface **HttpServletResponse** extiende **ServletResponse**. A través de esta interface los métodos de **HttpServlet** envían información a los clientes que les han pedido algún servicio.

El **Servlet API** dispone de clases e interfaces adicionales, no citadas en este apartado. Algunas de estas clases e interfaces serán consideradas en apartados posteriores.

8.5.2. Apache Tomcat

Apache Tomcat es uno de los servidores más utilizados para la ejecución de servlets. Es necesario su instalación en un ordenador, que actuará de servidor y por tanto las peticiones se dirigirán a dicho ordenador, bien sea por su nombre o por su IP. El servidor de aplicaciones se ejecuta como un programa o un servicio en el ordenador servidor, utilizando uno de los puertos disponibles mediante la configuración de Apache Tomcat.

La instalación de los servlets se realiza indicando en un fichero de configuración, la clase en que se define el servlet y su relación (mapping) con la petición que realiza el cliente.

En los tutoriales de Apache Tomcat y en el guión de las prácticas se detallan los pasos para realizar la instalación del servidor Apache Tomcat y de los servlets en él.

La configuración de Apache Tomcat no carga de nuevo de modo automático los **servlets** que hayan sido actualizados externamente; es decir, si se cambia algo en el código de un **servlet** y se vuelve a compilar, al hacer una nueva llamada al mismo utiliza la copia de la anterior versión del **servlet** que tiene cargada. Para que cargue la nueva es necesario cerrar del servidor (shutdown) y volverlo a arrancar (startup). Esta operación habrá que realizarla cada vez que se modifique el **servlet**.

El servidor Apache Tomcat utiliza las librerías que se encuentran en su instalación. Estas librerías pueden ser utilizadas para la compilación de los servletes. Para ello habrá que definir que la variable de entorno **CLASSPATH** contiene la ruta de acceso del fichero **servlet-api.jar** en el directorio **\lib**.

8.6. Ejemplo Introductorio

Para poder hacerse una idea del funcionamiento de un **servlet** y del aspecto que tienen los mismos, lo mejor es estudiar un ejemplo sencillo. Imagínese que en una página web se desea recabar la opinión de un visitante así como algunos de sus datos personales, con el fin de realizar un estudio estadístico. Dicha información podría ser almacenada en una base de datos para su posterior estudio.

La primera tarea sería diseñar un formulario en el que el visitante pudiera introducir los datos. Este paso es idéntico a lo que se haría al escribir un **programa CGI**, ya que bastará con utilizar los *tags* que proporciona el lenguaje **HTML** (**<FORM>**, **<ACTION>**, **<TYPE>**, etc.).

8.6.1. Formulario

El formulario contendrá dos campos de tipo TEXT donde el visitante introducirá su **nombre** y **apellidos**. A continuación, deberá indicar la opinión que le merece la página visitada eligiendo una entre tres posibles (**Buena**, **Regular** o **Mala**). Por último, se ofrece al usuario la posibilidad de escribir un **comentario** si así lo considera oportuno. En la Figura 8.4 puede observarse el diseño del formulario creado. El código correspondiente a la **página HTML** que contiene este formulario es el siguiente (fichero **MiServlet.htm**):

```

<HTML>
<HEAD>
    <TITLE>Envíe su opinión</TITLE>
</HEAD>

<BODY>
<H2>Por favor, envíenos su opinión acerca de este sitio web</H2>
<FORM ACTION="http://miServidor:8080/servlet/ServletOpinion" METHOD="POST">
    Nombre: <INPUT TYPE="TEXT" NAME="nombre" SIZE=15><BR>
    Apellidos: <INPUT TYPE="TEXT" NAME="apellidos" SIZE=30><P>
    Opinión que le ha merecido este sitio web<BR>
    <INPUT TYPE="RADIO" CHECKED NAME="opinion" VALUE="Buena">Buena<BR>
    <INPUT TYPE="RADIO" NAME="opinion" VALUE="Regular">Regular<BR>
    <INPUT TYPE="RADIO" NAME="opinion" VALUE="Mala">Mala<P>
    Comentarios <BR>
    <TEXTAREA NAME="comentarios" ROWS=6 COLS=40> </TEXTAREA><P>
    <INPUT TYPE="SUBMIT" NAME="botonEnviar" VALUE="Enviar">
    <INPUT TYPE="RESET" NAME="botonLimpiar" VALUE="Limpiar">
</FORM>
</BODY>
</HTML>

```

En el código anterior, hay algunas cosas que merecen ser comentadas. En primer lugar, es necesario asignar un identificador único (es decir, un valor de la propiedad **NAME**) a cada uno de los **campos** del formulario, ya que la información que reciba el **servlet** estará organizada en forma de **pares de valores**, donde uno de los elementos de dicho par será un **String** que contendrá el **nombre del campo**. Así, por ejemplo, si se introdujera como nombre del visitante "*Mikel*", el **servlet** recibiría del browser el par **nombre=Mikel**, que permitirá acceder de una forma sencilla al nombre introducido mediante el método **getParameter()**, tal y como se explicará posteriormente al analizar el **servlet** del ejemplo introductorio. Por este motivo es importante no utilizar nombres duplicados en los elementos de los formularios.

Por otra parte puede observarse que en el *tag* **<FORM>** se han utilizado dos propiedades, **ACTION** y **METHOD**. El método (**METHOD**) utilizado para la transmisión de datos es el método **HTTP POST**. También se podría haber utilizado el método **HTTP GET**, pero este método tiene algunas limitaciones en cuanto al volumen de datos transmisible, por lo que es recomendable utilizar el método **POST**. Mediante la propiedad **ACTION** deberá especificarse el **URL** del **servlet** que debe procesar los datos. Este **URL** contiene, en el ejemplo presentado, las siguientes características:

Por favor, envíenos su opinión acerca de este sitio web

Nombre:

Apellidos:

Opinión que le ha merecido este sitio web

Buena
 Regular
 Mala

Comentarios

Figura 8.4. Diseño del formulario de adquisición de datos.

- El **servlet** se encuentra situado en un servidor cuyo nombre es **miServidor** (un ejemplo más real podría ser **www.tecnun.es**). Este nombre dependerá del ordenador que proporcione los servicios de red. En cualquier caso, para poder hacer pruebas, se puede utilizar como nombre de servidor el *host local* o **localhost**, o su **número IP** si se conoce. Por ejemplo, se podría escribir:

```
<FORM ACTION="http://localhost:8080/servlet/ServletOpinion" METHOD="POST">
```

o de otra forma,

```
<FORM ACTION="http://Número_IP:8080/servlet/ServletOpinion" METHOD="POST">
```

- El **servidor HTTP** está “escuchando” por el puerto el **puerto** 8080. Todas las llamadas utilizando dicho puerto serán procesadas por el módulo del servidor encargado de la gestión de los **servlets**. En principio es factible la utilización de cualquier puerto libre del sistema, siempre que se indique al **servidor HTTP** cuál va a ser el puerto utilizado para dichas llamadas. Por diversos motivos, esto último debe ser configurado por el administrador del sistema.
- El **servlet** se encuentra situado en un subdirectorio del servidor llamado **servlet**. Este nombre es opcional.
- El nombre del **servlet** empleado es **ServletOpinion**, y es éste el que recibirá la información enviada por el cliente al servidor (el formulario en este caso), y quien se encargará de diseñar la respuesta, que pasará al servidor para que este a su vez la envíe de vuelta al cliente. Al final se ejecutará una clase llamada **ServletOpinion.class**.

8.6.2. Código del Servlet

Tal y como se ha mencionado con anterioridad, el **servlet** que gestionará toda la información del formulario se llamará **ServletOpinion**. Como un **servlet** es una clase de **Java**, deberá por tanto encontrarse almacenado en un fichero con el nombre **ServletOpinion.java**. En cualquier caso, por hacer lo más simple posible este ejemplo introductorio, este **servlet** se limitará a responder al usuario con una página **HTML** con la información introducida en el formulario, dejando para un posterior apartado el

estudio de cómo se almacenarían dichos datos. El código fuente de la clase **ServletOpinion** es el siguiente:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletOpinion extends HttpServlet {

    // Declaración de variables miembro correspondientes a
    // los campos del formulario
    private String nombre=null;
    private String apellidos=null;
    private String opinion=null;
    private String comentarios=null;

    // Este método se ejecuta una única vez (al ser inicializado el servlet)
    // Se suelen inicializar variables y realizar operaciones costosas en
    // tiempo de ejecución (abrir ficheros, bases de datos, etc)
    public void init(ServletConfig config) throws ServletException {
        // Llamada al método init() de la superclase (GenericServlet)
        // Así se asegura una correcta inicialización del servlet
        super.init(config);
        System.out.println("Iniciando ServletOpinion... ");
    } // fin del método init()

    // Este método es llamado por el servidor web al "apagarse" (al hacer
    // shutdown). Sirve para proporcionar una correcta desconexión de una
    // base de datos, cerrar ficheros abiertos, etc.
    public void destroy() {
        System.out.println("No hay nada que hacer... ");
    } // fin del método destroy()

    // Método llamado mediante un HTTP POST. Este método se llama
    // automáticamente al ejecutar un formulario HTML
    public void doPost (HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        // Adquisición de los valores del formulario a través del objeto req
        nombre=req.getParameter("nombre");
        apellidos=req.getParameter("apellidos");
        opinion=req.getParameter("opinion");
        comentarios=req.getParameter("comentarios");

        // Devolver al usuario una página HTML con los valores adquiridos
        devolverPaginaHTML(resp);
    } // fin del método doPost()

    public void devolverPaginaHTML(HttpServletRequest resp) {

        // Se establece el tipo de contenido MIME de la respuesta
        resp.setContentType("text/html");

        // Se obtiene un PrintWriter donde escribir (sólo para mandar texto)
        PrintWriter out = null;
        try {
            out=resp.getWriter();
        } catch (IOException io) {
            System.out.println("Se ha producido una excepcion");
        }
    }
}
```

```

// Se genera el contenido de la página HTML
out.println("<html>");
out.println("<head>");
out.println("<title>Valores recogidos en el formulario</title>");
out.println("</head>");
out.println("<body>");
out.println("<b><font size=+2>Valores recogidos del " );
out.println("formulario: </font></b>" );
out.println("<p><font size=+1><b>Nombre: </b>" + nombre + "</font>" );
out.println("<br><font size=+1><b>Apellido: </b>" +
           +apellidos + "</font><b><font size=+1></font></b>" );
out.println("<p><font size=+1> <b>Opini&acute;n: </b><i>" + opinion +
           "</i></font>" );
out.println("<br><font size=+1><b>Comentarios: </b>" + comentarios
           + "</font>" );
out.println("</body>" );
out.println("</html>");

// Se fuerza la descarga del buffer y se cierra el PrintWriter,
// liberando recursos de esta forma. IMPORTANTE
out.flush();
out.close();
} // fin de devolverPaginaHTML()

}

// Función que permite al servidor web obtener una descripción del servlet
// Qué cometido tiene, nombre del autor, comentarios adicionales, etc.
public String getServletInfo() {
    return "Este servlet lee los datos de un formulario" +
           " y los muestra en pantalla";
} // fin del método getServletInfo()
}

```

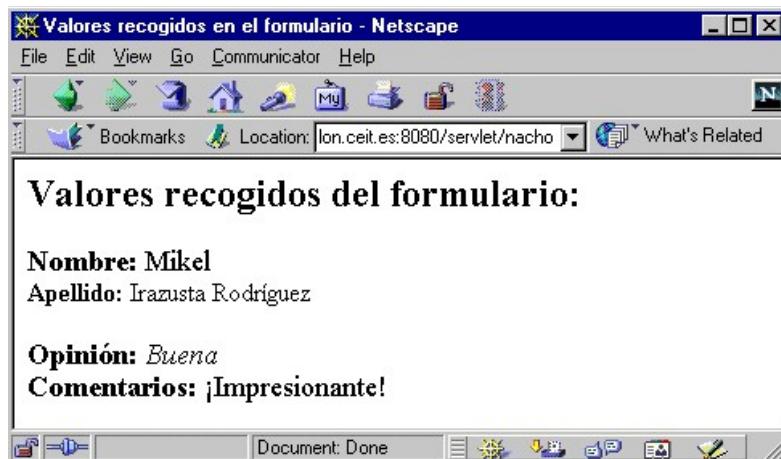


Figura 8.5. Página HTML devuelta por el servlet.

El resultado obtenido en el browser tras la ejecución del **servlet** puede apreciarse en la Figura 8.5. En aras de una mayor simplicidad en esta primera aproximación a los **servlets**, se ha evitado tratar de conseguir un código más sólido, que debería realizar las comprobaciones pertinentes (verificar que los **String** no son **null** después de leer el parámetro, excepciones que se pudieran dar, etc.) e informar al usuario acerca de posibles errores en caso de que fuera necesario.

En cualquier caso, puede observarse que el aspecto del código del **servlet** es muy similar al de cualquier otra clase de **Java**. Sin embargo, cabe destacar algunos aspectos particulares:

- La clase **ServletOpinion** hereda de la clase **HttpServlet**, que a su vez hereda de **GenericServlet**. La forma más sencilla (y por tanto la que debería ser siempre empleada) de crear un **servlet**, es heredar de la clase **HttpServlet**. De esta forma se está identificando la clase como un **servlet** que se conectará con un **servidor HTTP**. Más adelante se estudiará esto con más detalle.
- El método **init()** es el primero en ser ejecutado. Sólo es ejecutado **la primera vez** que el **servlet** es llamado. Se llama al método **init()** de la super-clase **GenericServlet** a fin de que la inicialización sea completa y correcta. La interface **ServletConfig** proporciona la información que necesita el **servlet** para inicializarse (parámetros de inicialización, etc.).
- El método **destroy()** no tiene ninguna función en este **servlet**, ya que no se ha utilizado ningún recurso adicional que necesite ser cerrado, pero tiene mucha importancia si lo que se busca es proporcionar una descarga correcta del **servlet** de la memoria, de forma que no queden recursos ocupados indebidamente, o haya conflictos entre recursos en uso. Tareas propias de este método son, por ejemplo, el cierre de las conexiones con otros ordenadores o con bases de datos.
- Como el formulario **HTML** utiliza el método **HTTP POST** para la transmisión de sus datos, habrá que redefinir el método **doPost()**, que se encarga de procesar la respuesta y que tiene como argumentos el objeto que contiene la petición y el que contiene la respuesta (pertenecientes a las clases **HttpServletRequest** y **HttpServletResponse**, respectivamente). Este método será llamado tras la inicialización del **servlet** (en caso de que no haya sido previamente inicializado), y contendrá el núcleo del código del **servlet** (llamadas a otros **servlets**, llamadas a otros métodos, etc.).
- El método **getServletInfo()** proporciona datos acerca del **servlet** (autor, fecha de creación, funcionamiento, etc.) al servidor web. No es en ningún caso obligatoria su utilización aunque puede ser interesante cuando se tienen muchos **servlets** funcionando en un mismo servidor y puede resultar compleja la identificación de los mismos.
- Por último, el método **devolverPaginaHTML()** es el encargado de mandar los valores recogidos del cliente. En primer lugar es necesario tener un *stream* hacia el cliente (**PrintWriter** cuando haya que mandar texto, **ServletOutputStream** para datos binarios). Posteriormente debe indicarse el tipo de contenido **MIME** de aquello que va dirigido al cliente (**text/html** en el caso presentado). Estos dos pasos son necesarios para poder enviar correctamente los datos al cliente. Finalmente, mediante el método **println()** se va generando la página **HTML** propiamente dicha (en forma de **String**).
- Puede sorprender la forma en que ha sido enviada la página **HTML** como **String**. Podría parecer más lógico generar un **String** en la forma (concatenación de **Strings**),

```
String texto = "<html><head> + . . . + <b>Nombre:</b>" + nombre + "</font>..."
```

y después escribirlo en el *stream* o flujo de salida. Sin embargo, uno de los parámetros a tener en cuenta en los **servlets** es el tiempo de respuesta, que tiene que ser el mínimo posible. En este sentido, la creación de un **String** mediante concatenación es bastante costosa, pues cada vez que se concatenan dos **Strings** mediante el signo + se están convirtiendo a **StringBuffers** y a su vez creando un nuevo **String**, lo que utilizado profusamente requiere más recursos que lo que se ha hecho en el ejemplo, donde se han escrito directamente mediante el método **println()**.

El esquema mencionado en este ejemplo se repite en la mayoría de los **servlets** y es el fundamento de esta tecnología. En posteriores apartados se efectuará un estudio más detallado de las clases y métodos empleados en este pequeño ejemplo.

9. Servlet API

El **Servlet API** es una extensión al **API de Java 1.1.x**, y también de **Java 2**. Contiene los paquetes **javax.servlet** y **javax.servlet.http**. El **API** proporciona soporte en cuatro áreas:

1. Control del ciclo de vida de un **servlet**: clase **GenericServlet**
2. Acceso al contexto del **servlet (servlet context)**
3. Clases de utilidades
4. Clases de soporte específicas para **HTTP**: clase **HttpServlet**

9.1. El ciclo de vida de un servlet: clase GenericServlet

La clase **GenericServlet** es una clase abstract porque declara el método **service()** como **abstract**. Aunque los **servlets** desarrollados en conexión con páginas web suelen derivar de la clase **HttpServlet**, puede ser útil estudiar el ciclo de vida de un **servlet** en relación con los métodos de la clase **GenericServlet**. Esto es lo que se hará en los apartados siguientes. Además, la clase **HttpServlet** hereda los métodos de **GenericServlet** y define el método **service()**.

Los **servlets** se ejecutan en el **servidor HTTP** como parte integrante del propio proceso del servidor. Por este motivo, el **servidor HTTP** es el responsable de la inicialización, llamada y destrucción de cada objeto de un **servlet**, tal y como puede observarse en la Figura 9.1.

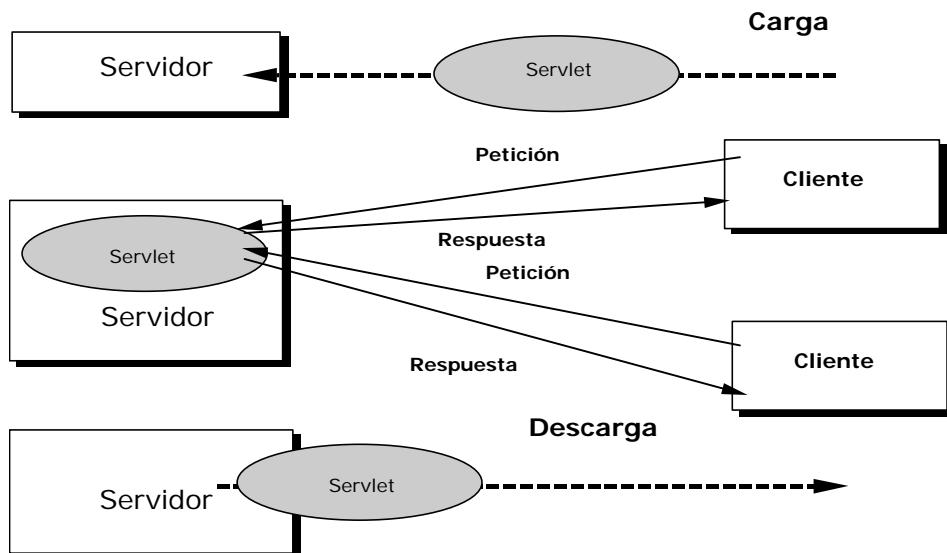


Figura 9.1. Ciclo de vida de un servlet.

Un **servidor web** se comunica con un **servlet** mediante los métodos de la interface **javax.servlet.Servlet**. Esta interface está constituida básicamente por tres métodos principales, alguno de los cuales ya se ha utilizado en el ejemplo introductorio:

- **init(), destroy() y service()**
y por dos métodos algo menos importantes:
- **getServletConfig(), getServletInfo()**

9.1.1. El método *init()* en la clase *GenericServlet*

Cuando un **servlet** es cargado por primera vez, el método ***init()*** es llamado por el **servidor HTTP**. Este método no será llamado nunca más mientras el **servlet** se esté ejecutando. Esto permite al **servlet** efectuar cualquier operación de inicialización potencialmente costosa en términos de CPU, ya que ésta sólo se ejecutará la primera vez. Esto es una ventaja importante frente a los **programas CGI**, que son cargados en memoria cada vez que hay una petición por parte del cliente. Por ejemplo, si en un día hay 500 consultas a una base de datos, mediante un **CGI** habría que abrir una conexión con la base de datos 500 veces, frente a una única apertura que sería necesaria con un **servlet**, pues dicha conexión podría quedar abierta a la espera de recibir nuevas peticiones.

Si el servidor permite pre-cargar los **servlets**, el método ***init()*** será llamado al iniciarse el servidor. Si el servidor no tiene esa posibilidad, será llamado la primera vez que haya una petición por parte de un cliente.

El método ***init()*** tiene un único argumento, que es una referencia a un objeto de la interface **ServletConfig**, que proporciona los argumentos de inicialización del **servlet**. Este objeto dispone del método ***getServletContext()*** que devuelve una referencia de la interface **ServletContext**, que a su vez contiene información acerca del entorno en el que se está ejecutando el **servlet**.

Siempre que se redefina el método ***init()*** de la clase base **GenericServlet** (o de **HttpServlet**, que lo hereda de **GenericServlet**), será preciso llamar al método ***init()*** de la super-clase, a fin de garantizar que la inicialización se efectúe correctamente. Por ejemplo:

```
public void init (ServletConfig config) throws ServletException {
    // Llamada al método init de la superclase
    super.init(config);
    System.out.println("Iniciando... ");

    // Definición de variables
    ...
    // Apertura de conexiones, ficheros, etc.
    ...
}
```

El servidor garantiza que el método ***init()*** termina su ejecución antes de que sea llamado cualquier otro método del **servlet**.

9.1.2. El método *service()* en la clase *GenericServlet*

Este método es el núcleo fundamental del **servlet**. Recuérdese que es **abstract** en **GenericServlet**, por lo que si el **servlet** deriva de esta clase deberá ser definido por el programador. Cada petición por parte del cliente se traduce en una llamada al método ***service()*** del **servlet**. El método ***service()*** lee la petición y debe producir una respuesta en base a los dos argumentos que recibe:

- Un objeto de la interface **ServletRequest** con datos enviados por el cliente. Estos incluyen parejas de parámetros clave/valor y un **InputStream**. Hay diversos métodos que proporcionan información acerca del cliente y de la petición efectuada por el mismo, entre otros los mostrados en la Tabla 9.1.
- Un objeto de la interface **ServletResponse**, que encapsula la respuesta del **servlet** al cliente. En el proceso de preparación de la respuesta, es necesario llamar al método ***setContentType()***, a fin de establecer el tipo de contenido **MIME** de la respuesta. La Tabla 9.2 indica los métodos de la interface **ServletResponse**.

Puede observarse en la Tabla 9.1 que hay dos formas de recibir la información de un formulario HTML en un **servlet**. La primera de ellas consiste en obtener los valores de los parámetros (métodos ***getParameterNames()*** y ***getParameterValues()***) y la segunda en recibir la información mediante un **InputStream** o un **Reader** y hacer por uno mismo su partición decodificación.

El cometido del método ***service()*** es conceptualmente bastante simple: genera una respuesta por cada petición recibida de un cliente. Es importante tener en cuenta que puede haber múltiples respuestas que están siendo procesadas al mismo tiempo, pues los **servlets** son **multithread**. Esto hace que haya que ser especialmente cuidadoso con los **threads**, para evitar por ejemplo que haya dos objetos de un **servlet** escribiendo simultáneamente en un mismo campo de una base de datos.

A pesar de la importancia del método **service()**, en general es no es aconsejable su definición (no queda más remedio que hacerlo si la clase del servlet deriva de **GenericServlet**, pero lo lógico es que el programador derive las clases de sus servlets de **HttpServlet**). El motivo es simple: la clase **HttpServlet** define **service()** de una forma más que adecuada, llamando a otros métodos (**doPost()**, **doGet()**, etc.) que son los que tiene que redefinir el programador. La forma de esta redefinición será estudiada en apartados posteriores.

9.1.3. El método **destroy()** en la clase **GenericServlet**:

Una buena implementación de este método debe permitir que el **servlet** concluya sus tareas de forma ordenada. De esta forma, es posible liberar recursos (ficheros abiertos, conexiones con bases de datos, etc.) de una forma limpia y segura. Cuando esto no es necesario o importante, no hará falta redefinir el método **destroy()**.

Puede suceder que al llamar al método **destroy()** haya peticiones de servicio que estén todavía siendo ejecutadas por el método **service()**, lo que podría provocar un fallo general del sistema. Por este motivo, es conveniente escribir el método **destroy()** de forma que se retrase la liberación de recursos hasta que no hayan concluido todas las llamadas al método **service()**. A continuación se presenta una forma de lograr una correcta descarga del **servlet**:

En primer lugar, es preciso saber si existe alguna llamada al método **service()** pendiente de ejecución, para lo cual se debe llevar un **contador** con las llamadas activas a dicho método.

Aunque en general es poco recomendable redefinir **service()** (en caso de tratarse de un **servlet** que derive de **HttpServlet**). Sin embargo, en este caso sí resulta conveniente su redefinición, para poder saber cuándo ha sido llamado. El método redefinido deberá llamar al método **service()** de su super-clase para mantener íntegra la funcionalidad del **servlet**.

Los métodos de actualización del **contador** deben estar **sincronizados**, para evitar que dicho valor sea accedido simultáneamente por dos o más **threads**, lo que podría hacer que su valor fuera erróneo.

Además, no basta con que el **servlet** espere a que todos los métodos **service()** hayan acabado. Es preciso indicarle a dicho método que el servidor se dispone a apagarse. De otra forma, el **servlet** podría quedar esperando indefinidamente a que los métodos **service()** acabaran. Esto se consigue utilizando una variable **boolean** que establezca esta condición.

Métodos de ServletRequest	Comentarios
Public abstract int getContentLength()	Devuelve el tamaño de la petición del cliente o -1 si es desconocido.
Public abstract String getContentType()	Devuelve el tipo de contenido MIME de la petición o null si éste es desconocido.
Public abstract String getProtocol()	Devuelve el protocolo y la versión de la petición como un String en la forma <protocolo>/<versión mayor>. <versión menor>
public abstract String getScheme()	Devuelve el tipo de esquema de la URL de la petición: http, https, ftp...
public abstract String getServerName()	Devuelve el nombre del host del servidor que recibió la petición..
public abstract int getServerPort()	Devuelve el número del puerto en el que fue recibida la petición.
public abstract String getRemoteAddr()	Devuelve la dirección IP del ordenador que realizó la petición.
public abstract String getRemoteHost()	Devuelve el nombre completo del ordenador que realizó la petición.
public abstract ServletInputStream getInputStream() throws IOException	Devuelve un InputStream para leer los datos binarios que vienen dentro del cuerpo de la petición.
public abstract String getParameter(String)	Devuelve un String que contiene el valor del parámetro especificado, o null si dicho parámetro no existe. Sólo debe emplearse cuando se está seguro de que el parámetro tiene un único valor.
public abstract String[] getParameterValues(String)	Devuelve los valores del parámetro especificado en forma de un array de Strings , o null si el parámetro no existe. Útil cuando un parámetro puede tener más de un valor.
public abstract Enumeration getParameterNames()	Devuelve una enumeración en forma de String de los parámetros encapsulados en la petición. No devuelve nada si el InputStream está vacío.
public abstract BufferedReader getReader() throws IOException	Devuelve un BufferedReader que permite leer el texto contenido en el cuerpo de la petición.
public abstract String getCharacterEncoding()	Devuelve el tipo de codificación de los caracteres empleados en la petición.

Tabla 9.1. Métodos de la interface **ServletRequest**.

Métodos de ServletResponse	Comentarios
ServletOutput Stream getOutputStream()	Permite obtener un ServletOutputStream para enviar datos binarios
PrintWriter getWriter()	Permite obtener un PrintWriter para enviar caracteres
setContentType(String)	Establece el tipo MIME de la salida
setContentLength(int)	Establece el tamaño de la respuesta

Tabla 9.2. Métodos de la interface **ServletResponse**.

Todas las consideraciones anteriores se han introducido en el siguiente código:

```
public class ServletSeguro extends HttpServlet {
    ...
    private int contador=0;
    private boolean apagandose=false;
    ...
    protected synchronized void entrandoEnService() {
        contador++;
    }

    protected synchronized void saliendoDeService() {
        contador--;
    }

    protected synchronized void numeroDeServicios() {
        return contador;
    }

    protected void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        entrandoEnService();
        try {
            super.service(req, resp);
        } finally {
            saliendoDeService();
        }
    } // fin del método service()

    protected void setApagandose(boolean flag){
        apagandose=flag;
    }

    protected boolean estaApagandose() {
        return apagandose;
    }
    ...
    public void destroy() {
        // Comprobar que hay servicios en ejecución y en caso afirmativo
        // ordernarles que paren la ejecución
        if(numeroDeServicios()>0)
            setApagandose(true);

        // Mientras haya servicios en ejecución, esperar
        while(numServices()>0) {
            try {
                Thread.sleep(intervalo);
            } catch(InterruptedException e) {
            } // fin del catch
        } // fin del while
    } // fin de destroy()
    ...
    // Servicio
    public void doPost(...) {
        ...
        // Comprobación de que el servidor no se está apagando
        for (i=0; ((i<numeroDeCosasAHacer)&& !estaApagandose()); i++) {
            try {
                ...
                // Aquí viene el código
            } catch(Exception e)
```

```

        } // fin del for
    } // fin de doPost()
} // fin de la clase ServletEjemplo

```

9.2. El contexto del servlet (servlet context)

Un **servlet** vive y muere dentro de los límites del proceso del servidor. Por este motivo, puede ser interesante en un determinado momento obtener información acerca del entorno en el que se está ejecutando el **servlet**. Esta información incluye la disponible en el momento de inicialización del **servlet**, la referente al propio servidor o la información contextual específica que puede contener cada petición de servicio.

9.2.1. Información durante la inicialización del servlet

Esta información es suministrada al **servlet** mediante el argumento **ServletConfig** del método **init()**. Cada **servidor HTTP** tiene su propia forma de pasar información al **servlet**. En cualquier caso, para acceder a dicha información habría que emplear un código similar al siguiente:

```

String valorParametro;
public void init(ServletConfig config) {
    valorParametro = config.getInitParameter(nombreParametro);
}

```

Como puede observarse, se ha empleado el método **getInitParameter()** de la interface **ServletConfig** (implementada por **GenericServlet**) para obtener el valor del parámetro. Asimismo, puede obtenerse una **enumeración** de todos los nombres de parámetros mediante el método **getInitParameterNames()** de la misma interface.

9.2.2. Información contextual acerca del servidor

La información acerca del servidor está disponible en todo momento a través de un objeto de la interface **ServletContext**. Un **servlet** puede obtener dicho objeto mediante el método **getServletContext()** aplicable a un objeto **ServletConfig**.

La interface **ServletContext** define los métodos descritos en la Tabla 9.3.

9.3. Clases de utilidades (Utility Classes)

El **Servlet API** proporciona una serie de utilidades que se describen a continuación.

- La primera de ellas es la interface ***javax.servlet.SingleThreadModel*** que puede hacer más sencillo el desarrollo de ***servlets***. Si un ***servlet*** implementa dicha interface, el servidor sabe que nunca debe llamar al método ***service()*** mientras esté procesando una petición anterior. Es decir, el servidor procesa todas las peticiones de servicio dentro de un mismo ***thread***. Sin embargo, a pesar de que esto puede facilitar el desarrollo de ***servlets***, puede ser un gran obstáculo en cuanto al rendimiento del

Métodos de <i>ServletContext</i>	Comentarios
public abstract Object getAttribute(String)	Devuelve información acerca de determinados atributos del tipo clave/valor del servidor. Es propio de cada servidor.
public abstract Enumeration getAttributeNames()	Devuelve una enumeración con los nombre de atributos disponibles en el servidor.
public abstract String getMimeType(String)	Devuelve el tipo MIME de un determinado fichero.
public abstract String getRealPath(String)	Traduce una ruta de acceso virtual a la ruta relativa al lugar donde se encuentra el directorio raíz de páginas HTML
public abstract String getServerInfo()	Devuelve el nombre y la versión del servicio de red en el que está siendo ejecutado el <i>servlet</i> .
public abstract Servlet getServlet(String) throws ServletException	Devuelve un objeto <i>servlet</i> con el nombre dado.
public abstract Enumeration getServletNames()	Devuelve un enumeración con los <i>servlets</i> disponibles en el servidor.
public abstract void log(String)	Escribe información en un fichero de log . El nombre del mismo y su formato son propios de cada servidor.

Tabla 9.3. Métodos de la interface ***ServletContext***.

servlet. Por ello, a veces es preciso explorar otras opciones. Por ejemplo, si un ***servlet*** accede a una base de datos para su modificación, existen dos alternativas para evitar conflictos por accesos simultáneos:

1. **Sincronizar los métodos** que acceden a los recursos, con la consiguiente complejidad en el código del ***servlet***.
 2. Implementar la ya citada interface ***SingleThreadModel***, solución más sencilla pero que trae consigo un aumento en el tiempo de respuesta. En este caso, no es necesario escribir ningún código adicional, basta con implementar la interface. Es una forma de **marcar** aquellos ***servlets*** que deben tener ese comportamiento, de forma que el servidor pueda identificarlos.
- El **Servlet API** incluye dos clases de **excepciones**:
 1. La excepción ***javax.servlet.ServletException*** puede ser empleada cuando ocurre un fallo general en el ***servlet***. Esto hace saber al servidor que hay un problema.
 2. La excepción ***javax.servlet.UnavailableException*** indica que un ***servlet*** no se encuentra disponible. Los ***servlets*** pueden notificar esta excepción en cualquier momento. Existen dos tipos de indisponibilidades:
 - a) **Permanente**: El ***servlet*** no podrá seguir funcionando hasta que el administrador del servidor haga algo. En este estado, el ***servlet*** debería escribir en el fichero de **log** una descripción del problema, y posibles soluciones.

- b) **Temporal:** El *servlet* se ha encontrado con un problema que es potencialmente temporal, como pueda ser un disco lleno, un servidor que ha fallado, etc. El problema puede arreglarse con el tiempo o puede requerir la intervención del administrador.

9.4. Clase HttpServlet: soporte específico para el protocolo HTTP

Los *servlets* que utilizan el protocolo **HTTP** son los más comunes. Por este motivo, **Sun** ha incluido un package específico para estos *servlets* en su **JSDK: javax.servlet.http**. Antes de estudiar dicho *package* en profundidad, se va a hacer una pequeña referencia al protocolo **HTTP**.

HTTP son las siglas de **HyperText Transfer Protocol**, que es un protocolo mediante el cual los browser y los servidores puedan comunicarse entre sí, mediante la utilización de una serie de **métodos: GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT y OPTIONS**. Para la mayoría de las aplicaciones, bastará con conocer los tres primeros.

9.4.1. Método GET: codificación de URLs

El método **HTTP GET** solicita **información** a un **servidor web**. Esta información puede ser un fichero, el resultado de un programa ejecutado en el servidor (como un *servlet*, un **programa CGI**, ...), etc.

En la mayoría de los servidores web los *servlets* son accedidos mediante un **URL** que comienza por `/servlet/`. El siguiente método **HTTP GET** solicita el servicio del *servlet MiServlet* al servidor `miServidor.com`, con lo cual petición **GET** tiene la siguiente forma (en **negrita** el contenido de la petición):

```
GET /servlet/MiServlet?nombre=Antonio&Apellido=Lopez%20de%20Romera HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.5 (
    compatible;
    MSIE 4.01;
    Windows NT)
Host: miServidor.com
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg
```

El **URL** de esta petición **GET** llama a un *servlet* llamado **MiServlet** y contiene dos parámetros, **nombre** y **apellido**. Cada parámetro es un par que sigue el formato **clave=valor**. Los parámetros se especifican poniendo un **signo de interrogación (?)** tras el nombre del *servlet*. Además, los distintos parámetros están separados entre sí por el símbolo **ampersand (&)**.

Obsérvese que la secuencia de caracteres **%20** aparece dos veces en el apellido. Es una forma de decir que hay un **espacio** entre "Lopez" y "de", y otro entre "de" y "Romera". Esto ocurre por la forma en que se codifican los **URL** en el protocolo **HTTP**. Sigue lo mismo con otros símbolos como las tildes u otros caracteres especiales. Esta codificación sigue el esquema:

%+<valor hexadecimal del código ASCII correspondiente al carácter>

Por ejemplo, el carácter **á** se escribiría como **%E1** (código ASCII 225). También se puede cambiar la secuencia **%20** por el signo **+**, obteniendo el mismo efecto.

En cualquier caso, los programadores de *servlets* no deben preocuparse en principio por este problema, ya que la clase **HttpServletRequest** se encarga de la decodificación, de forma que los valores de los parámetros sean accesibles mediante el método **getParameter(String parametro)** de dicha clase. Sin embargo, hay que tener cuidado con algunos caracteres a la hora de incluirlos en un **URL**, en concreto con aquellos caracteres no pertenecientes al código ASCII y con aquellos que tienen un significado concreto para el protocolo **HTTP**. Más en concreto, se pueden citar los siguientes caracteres especiales y su secuencia o código equivalente:

```
" (%22), # (%23), % (%25), & (%26), + (%2B), , (%2C), / (%2F),
: (%3A), < (%3C), = (%3D), > (%3E), ? (%3F) y @ (%40).
```

Adicionalmente, **Java** proporciona la posibilidad de codificar un **URL** de forma que cumpla con las anteriores restricciones. Para ello, se puede utilizar la clase **URLEncoder**, que se encuentra incluida en el package **java.net**, que es un package estándar de **Java**. Dicha clase tiene un único método, **String encode(String)**, que se encarga de codificar el **String** que recibe como argumento, devolviendo otro **String** con el **URL** debidamente codificado. Así, considérese el siguiente ejemplo:

```
import java.net.*;

public class Codificar {
    public static void main(String argv[]) {
        String URLcodificada = URLEncoder.encode(
            "/servlet/MiServlet?nombre=Antonio"&"Apellido=López de Romera");
        System.out.println(URLcodificada);
    }
}
```

que cuando es ejecutado tiene como resultado la siguiente secuencia de caracteres:

```
%2Fservlet%2FMiServlet%3Fnombre%3DAntonio%26Apellido%3DL%2Apez+de+Romera
```

Obsérvese además que, cuando sea necesario escribir una comilla dentro del **String** de **out.println(String)**, hay que precederla por el carácter **escape** (\). Así, la sentencia:

```
out.println("<A HREF=\"http://www.yahoo.com\">Yahoo</A>"); // INCORRECTA
```

es incorrecta y produce errores de compilación. Deberá ser sustituida por:

```
out.println("<A HREF=\\"http://www.yahoo.com\\">Yahoo</A>");
```

Las peticiones **HTTP GET** tienen una limitación importante (recuérdese que transmiten la información a través de las variables de entorno del sistema operativo) y es un límite en la cantidad de caracteres que pueden aceptar en el **URL**. Si se envían los datos de un formulario muy extenso mediante **HTTP GET** pueden producirse errores por este motivo, por lo que habría que utilizar el método **HTTP POST**.

Se suele decir que el método **GET** es **seguro** e **idempotente**:

- **Seguro**, porque no tiene ningún efecto secundario del cual pueda considerarse al usuario responsable del mismo. Es decir, por ejemplo, una llamada del método **GET** no debe ser capaz en teoría de alterar una base de datos. **GET** debería servir únicamente para obtener información.
- **Idempotente**, porque puede ser llamado tantas veces como se quiera de una forma segura.

Es como si **GET** fuera algo así como *ver pero no tocar*.

9.4.2. Método HEAD: información de ficheros

Este método es similar al anterior. La petición del cliente tiene la misma forma que en el método **GET**, con la salvedad de que en lugar de **GET** se utiliza **HEAD**. En este caso el servidor responde a dicha petición enviando únicamente **información acerca del fichero**, y no el fichero en sí. El método **HEAD** se suele utilizar frecuentemente para comprobar lo siguiente:

- La **fecha de modificación** de un documento presente en el servidor.
- El **tamaño del documento** antes de su descarga, de forma que el browser pueda presentar información acerca del progreso de descarga.
- El **tipo de servidor**.
- El **tipo de documento** solicitado, de forma que el cliente pueda saber si es capaz de soportarlo.

El método **HEAD**, al igual que **GET**, es **seguro** e **idempotente**.

9.4.3. Método POST: el más utilizado

El método **HTTP POST** permite al cliente **enviar información al servidor**. Se debe utilizar en lugar de **GET** en aquellos casos que requieran transferir una cantidad importante de datos (formularios).

El método **POST** no tiene la limitación de **GET** en cuanto a volumen de información transferida, pues ésta no va incluida en el **URL** de la petición, sino que viaja encapsulada en un **input stream** que llega al **servlet** a través de la entrada estándar.

El encabezamiento y el contenido (en **negrita**) de una petición **POST** tiene la siguiente forma:

```
POST /servlet/MiServlet HTTP/1.1
User-Agent: Mozilla/4.5 (
    compatible;
    MSIE 4.01;
    Windows NT)
Host: www.MiServidor.com
Accept: image/gif, image/x-bitmap, image/jpeg, image/jpeg, */
Content-type: application/x-www-form-urlencoded
Content-length: 39

nombre=Antonio&Apellido=Lopez%20de%20Romera
```

Nótese la existencia de una **Línea en blanco** entre el encabezamiento (*header*) y el comienzo de la información extendida. Esta línea en blanco indica el final del **header**.

A diferencia de los anteriores métodos, **POST** **no** es ni **seguro** ni **idempotente**, y por tanto es conveniente su utilización en aquellas aplicaciones que requieran operaciones más complejas que las de sólo-lectura, como por ejemplo modificar bases de datos, etc.

9.4.4. Clases de soporte HTTP

Una vez que se han presentado unas ciertas nociones sobre el protocolo **HTTP**, resulta más sencillo entender las funciones del package **javax.servlet.http**, que facilitan de sobremanera la creación de **servlets** que empleen dicho protocolo.

La clase abstracta **javax.servlet.http.HttpServlet** incluye un numero de importante de funciones adicionales e implementa la interface **javax.servlet.Servlet**. La forma más sencilla de escribir un **servlet HTTP** es heredando de **HttpServlet** como puede observarse en la Figura 9.2.

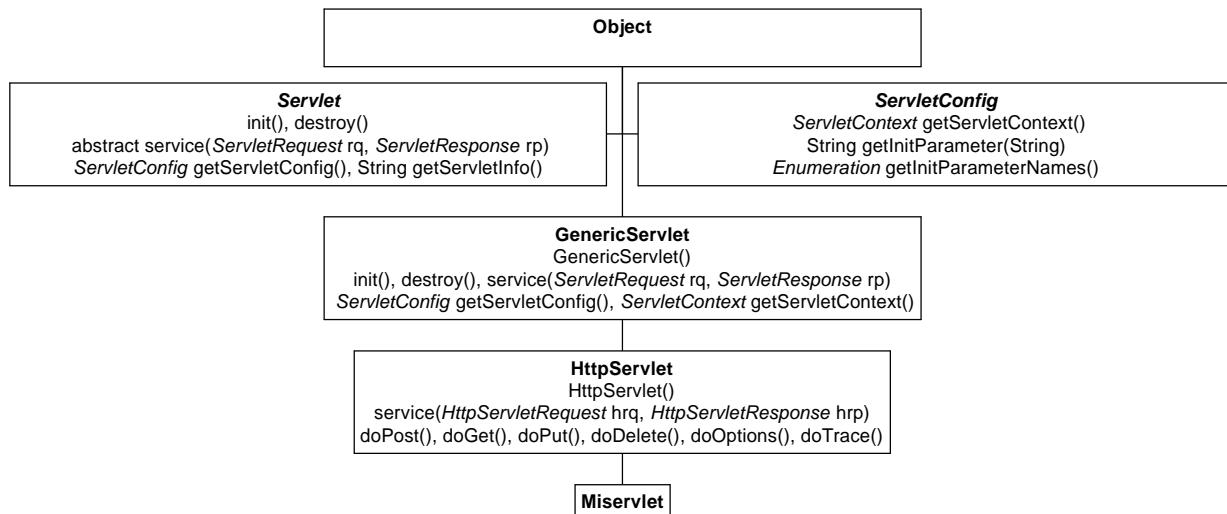


Figura9.2. Jerarquía de clases en servlets.

La clase **HttpServlet** es también una clase **abstract**, de modo que es necesario definir una clase que derive de ella y redefinir en la clase derivada **al menos uno** de sus métodos, tales como **doGet()**, **doPost()**, etc.

Como ya se ha comentado, la clase **HttpServlet** proporciona una implementación del método **service()** en la que distingue qué método se ha utilizado en la petición (**GET**, **POST**, etc.), llamando seguidamente al método adecuado (**doGet()**, **doHead()**, **doDelete()**, **doOptions()**, **doPost()** y **doTrace()**). Estos métodos e corresponden con los métodos **HTTP** anteriormente citados.

Así pues, la clase **HttpServlet** no define el método **service()** como **abstract**, sino como **protected**, al igual que los métodos **init()**, **destroy()**, **doGet()**, **doPost()**, etc., de forma que ya no es necesario escribir una implementación de **service()** en un **servlet** que herede de dicha clase. Si por algún motivo es necesario redefinir el método **service()**, es muy conveniente llamar desde él al método **service()** de la super-clase (**HttpServlet**).

La clase **HttpServlet** es bastante “inteligente”, ya que es también capaz de saber qué métodos han sido redefinidos en una sub-clase, de forma que puede comunicar al cliente qué tipos de métodos soporta el **servlet** en cuestión. Así, si en la clase **MiServlet** sólo se ha redefinido el método **doPost()**, si el cliente realiza una petición de tipo **HTTP GET** el servidor lanzará automáticamente un mensaje de error similar al siguiente:

501 Method GET Not Supported

donde el número que aparece antes del mensaje es un código empleado por los **servidores HTTP** para indicar su estado actual. En este caso el código es el 501.

No siempre es necesario redefinir todos los métodos de la clase **HttpServlet**. Por ejemplo, basta definir el método **doGet()** para que el **servlet** responda por sí mismo a peticiones del tipo **HTTP HEAD** o **HTTP OPTIONS**.

9.4.5. Modo de empleo de la clase **HttpServlet**

Todos los métodos de clase **HttpServlet** que debe o puede redefinir el programador (**doGet()**, **doPost()**, **doPut()**, **doOptions()**, etc.) reciben como argumentos un objeto **HttpServletRequest** y otro **HttpServletResponse**.

La interface **HttpServletRequest** proporciona numerosos métodos para obtener información acerca de la petición del cliente (así como de la identidad del mismo). Consultar la documentación del **API** para mayor información. Por otra parte, el objeto de la interface **HttpServletResponse** permite enviar desde el **servlet** al cliente información acerca del estado del servidor (métodos **sendError()** y **setStatus()**), así como establecer los valores del *header* del mensaje saliente (métodos **setHeader()**, **setDateHeader()**, etc.).

Recuérdese que tanto **HttpServletRequest** como **HttpServletResponse** son interfaces que derivan de las interfaces **ServletRequest** y **ServletResponse** respectivamente, por lo que se pueden también utilizar todos los métodos declarados en estas últimas.

Recuérdese a modo de recapitulación que el método **doGet()** debería:

1. Leer los datos de la solicitud, tales como los nombres de los parámetros y sus valores
2. Establecer el *header* de la respuesta (longitud, tipo y codificación)
3. Escribir la respuesta en formato HTML para enviarla al cliente.

Recuérdese que la implementación de este método debe ser **segura** e **idempotente**.

El método **doPost()** por su parte, debería realizar las siguientes funciones:

1. Obtener input stream del cliente y leer los parámetros de la solicitud.
2. Realizar aquello para lo que está diseñado (actualización de bases de datos, etc.).
3. Informar al cliente de la finalización de dicha tarea o de posibles imprevistos. Para ello hay que establecer primero el tipo de la respuesta, obtener luego un PrintWriter y enviar a través suyo el mensaje HTML.

10. Sesión en Servlets

10.1. Formas de seguir la trayectoria de los usuarios

Los **servlets** permiten seguir la trayectoria de un cliente, es decir, obtener y mantener una determinada información acerca del cliente. De esta forma se puede **tener identificado a un cliente** (usuario que está utilizando un browser) durante un determinado tiempo. Esto es muy importante si se quiere disponer de aplicaciones que impliquen la ejecución de varios **servlets** o la ejecución repetida de un mismo **servlet**. Un claro ejemplo de aplicación de esta técnica es el de los **comercios vía Internet** que permiten llevar un **carrito de la compra** en el que se van guardando aquellos productos solicitados por el cliente. El cliente puede ir navegando por las distintas secciones del comercio virtual, es decir realizando distintas conexiones **HTTP** y ejecutando diversos **servlets**, y a pesar de ello no se pierde la información contenida en el carrito de la compra y se sabe en todo momento que es un mismo cliente quien está haciendo esas conexiones diferentes.

El mantener información sobre un cliente a lo largo de un proceso que implica múltiples conexiones se puede realizar de tres formas distintas:

- Mediante **cookies**
- Mediante **seguimiento de sesiones (Session Tracking)**
- Mediante la **reescritura** de URLs

10.2. Cookies

Estrictamente hablando “cookie” significa galleta. Parece ser que dicha palabra tiene otro significado: se utilizaría también para la ficha que le dan a un cliente en un guardarropa al dejar el abrigo y que tiene que entregar para que le reconozcan y le devuelvan dicha prenda. Éste sería el sentido de la palabra **cookie** en el contexto de los **servlets**: algo que se utiliza para que un **servidor HTTP** reconozca a un cliente como alguien que ya se había conectado anteriormente. Como era de esperar, los **cookies** en **Java** son objetos de la clase **Cookie**, en el package **javax.servlet.http**.

El empleo de **cookies** en el seguimiento de un cliente requiere que dicho cliente sea capaz de soportarlas. Sin embargo, puede ocurrir que a pesar de estar disponible, dicha opción esté **desactivada** por el usuario, por lo que puede ser necesario emplear otras alternativas de seguimiento de clientes como la reescritura de **URLs**. Esto es debido a que los **servlets** envían **cookies** a los clientes junto con la respuesta, y los clientes las devuelven junto con una petición. Así, si un cliente tiene activada la opción **No cookies** o similar en su navegador, no le llegará la **cookie** enviada por el **servlet**, por lo que el seguimiento será imposible.

Cada **cookie** tiene un nombre que puede ser el mismo para varias **cookies**, y se almacenan en un directorio o fichero predeterminado en el disco duro del cliente. De esta forma, puede mantenerse información acerca del cliente durante días, ya que esa información queda almacenada en el ordenador del cliente (aunque no indefinidamente, pues las **cookies** tienen una fecha de caducidad).

La forma en que se envían **cookies** es bastante sencilla en concepto. Añadiendo una **clave** y un **valor** al **header** del mensaje es posible enviar **cookies** al cliente, y desde éste al servidor. Adicionalmente, es posible incluir otros parámetros adicionales, tales como **comentarios**. Sin embargo, estos no suelen ser tratados correctamente por los browsers actuales, por lo que su empleo es desaconsejable. Un servidor puede enviar más de una **cookie** al cliente (hasta veinte **cookies**).

Las **cookies** almacenadas en el cliente son enviadas en principio sólo al servidor que las originó. Por este motivo (porque las **cookies** son enviadas al **servidor HTTP** y **no** al **servlet**), los **servlets** que se ejecutan en un mismo servidor comparten las mismas **cookies**.

La forma de implementar todo esto es relativamente simple gracias a la clase **Cookie** incluida en el **Servlet API**. Para enviar una **cookie** es preciso:

- Crear un objeto **Cookie**
- Establecer sus atributos
- Enviar la **cookie**

Por otra parte, para obtener información de una **cookie**, es necesario:

- Recoger todas las **cookies** de la petición del cliente
- Encontrar la **cookie** precisa
- Obtener el valor recogido en la misma

10.2.1. Crear un objeto Cookie

La clase **javax.servlet.http.Cookie** tiene un constructor que presenta como argumentos un **String** con el **nombre** de la **cookie** y otro **String** con su **valor**. Es importante hacer notar que toda la información almacenada en **cookies** lo es en forma de **String**, por lo que será preciso convertir cualquier valor a **String** antes de añadirlo a una **cookie**.

Hay que ser cuidadoso con los **nombres** empleados, ya que aquellos que contengan caracteres especiales pueden no ser válidos. Adicionalmente, aquellos que comienzan por el símbolo de dólar (\$) no pueden emplearse, por estar reservados.

Con respecto al **valor** de la **cookie**, en principio puede tener cualquier forma, aunque hay que tener cautela con el valor **null**, que puede ser incorrectamente manejado por los browsers, así como espacios en blanco o los siguientes caracteres:

```
[ ] ( ) = , " / ? @ : ;
```

Por último, es importante saber que es necesario crear la **cookie** antes de acceder al **Writer** del objeto **HttpServletResponse**, pues como las **cookies** son enviadas al cliente en el **header** del mensaje, y éstas deben ser escritas antes de crear el **Writer**.

Por ejemplo, el siguiente código crea una **cookie** con el nombre "**Compra**" y el valor de **IdObjetoAComprar**, que es una variable que contiene la identificación de un objeto a comprar (301):

```
...
String IdObjetoAComprar = new String("301");
if(IdObjetoAComprar!=null)
    Cookie miCookie=new Cookie("Compra", IdObjetoAComprar);
```

10.2.2. Establecer los atributos de la cookie

La clase **Cookie** proporciona varios métodos para establecer los valores de una **cookie** y sus atributos. Entre otros, los mostrados en la Tabla 10.1.

Todos estos métodos tienen sus métodos **getXXX()** correspondientes incluidos en la misma clase.

Por ejemplo, se puede cambiar el valor de una **cookie** de la siguiente forma:

```
...
Cookie miCookie=new Cookie("Nombre", "ValorInicial");
miCookie.setValue("ValorFinal");
```

o hacer que sea eliminada al cerrar el browser:

```
miCookie.setMaxAge(-1);
...
```

Métodos de la clase Cookie	Comentarios
public void setComment(String)	Si un browser presenta esta cookie al usuario, el cometido de la cookie será descrito mediante este comentario.
public void setDomain(String)	Establece el patrón de dominio a quien permitir el acceso a la información contenida en la cookie . Por ejemplo .yahoo.com permite el acceso a la cookie al servidor www.yahoo.com pero no a a.b.yahoo.com
public void setMaxAge(int)	Establece el tiempo de caducidad de la cookie en segundos. Un valor -1 indica al browser que borre la cookie cuando se apague. Un valor 0 borra la cookie de inmediato.
public void setPath(String)	Establece la ruta de acceso del directorio de los servlets que tienen acceso a la cookie . Por defecto es aquel que originó la cookie .
Public void setSecure(boolean)	Indica al browser que la cookie sólo debe ser enviada utilizando un protocolo seguro (https). Sólo debe utilizarse en caso de que el servidor que haya creado la cookie lo haya hecho de forma segura.
public void setValue(String)	Establece el valor de la cookie
public void setVersion(int)	Establece la versión del protocolo de la cookie .

Tabla 10.1. Métodos de la clase **Cookie**.

10.2.3. Enviar la cookie

Las **cookies** son enviadas como parte del *header* de la respuesta al cliente. Por ello, tienen que ser añadidas a un objeto **HttpServletResponse** mediante el método **addCookie(Cookie)**. Tal y como se ha explicado con anterioridad, esto debe realizarse antes de llamar al método **getWriter()** de ese mismo objeto. Sirva como ejemplo el siguiente código:

```
...
public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    ...
    Cookie miCookie=new Cookie("Nombre","Valor");
    miCookie.setMaxAge(-1);
    miCookie.setComment("Esto es un comentario");
    resp.addCookie(miCookie);

    PrintWriter out=resp.getWriter();
    ...
}
```

10.2.4. Recoger las cookies

Los clientes devuelven las **cookies** como parte integrante del *header* de la petición al servidor. Por este motivo, las **cookies** enviadas deberán recogerse del objeto **HttpServletRequest** mediante el método **getCookies()**, que devuelve un array de objetos **Cookie**. Véase el siguiente ejemplo:

```
...
Cookie miCookie = null;
Cookie[] arrayCookies = req.getCookies();
miCookie = arrayCookies[0];
...
```

El anterior ejemplo recoge la primera **cookie** del **array de cookies**.

Por otra parte, habrá que tener cuidado, pues tal y como se ha mencionado con anterioridad, puede haber más de una **cookie** con el mismo nombre, por lo que habrá que detectar de alguna manera cuál es la **cookie** que se necesita.

10.2.5. Obtener el valor de la cookie

Para obtener el valor de una **cookie** se utiliza el método **getValue()** de la clase **Cookie**. Obsérvese el siguiente ejemplo. Supóngase que se tiene una tienda virtual de libros y que un usuario ha decidido eliminar un libro del carro de la compra. Se tienen dos **cookies** con el mismo nombre (**compra**), pero con dos valores (**libro1**, **libro2**). Si se quiere eliminar el valor **libro1**:

```
...
String libroABorrar=req.getParameter("Borrar");
...
if(libroABorrar!=null) {
    Cookie[] arrayCookies=req.getCookies();
    for(i=0;i<arrayCookies.length;i++) {
        Cookie miCookie=arrayCookies[i];
        if(miCookie.getName().equals("compra")
            &&miCookie.getValue().equals("libro1")) {
            miCookie.setMaxAge(0); // Elimina la cookie
        } // fin del if
    } // fin del for
} // fin del if
...
```

Tal y como se ha dicho con anterioridad, una **cookie** contiene como valor un **String**. Este **String** debe ser tal que signifique algo para el **servlet**. Con esto se quiere decir que es responsabilidad exclusiva del programador establecer que formato o codificación va a tener ese **String** que almacena la **cookie**. Por ejemplo, si se tiene una tienda on-line, pueden establecerse tres posibles tipos de status de un producto (con referencia al interés de un cliente determinado por dicho producto): el cliente se ha solicitado información sobre el producto (**A**), el cliente lo tiene contenido en el carro de la compra (**B**) o el cliente ya ha comprado uno anteriormente (**C**). Así, si por ejemplo el código del producto fuera el **301** y estuviera contenido en el carro de la compra, podría enviarse una **cookie** con el siguiente valor:

```
Cookie miCookie = new Cookie("NombreDeCookie", "301_B");
```

El programador deberá establecer una codificación propia y ser capaz de descodificarlo posteriormente.

10.3. Sesiones (Session Tracking)

Una **sesión** es una conexión continuada de un mismo browser a un servidor durante un tiempo prefijado de tiempo. Este tiempo depende habitualmente del servidor, aunque a partir de la versión **2.1** del **Servlet API** puede establecerse mediante el método **setMaxInactiveInterval(int)** de la interface **HttpSession**. Esta interface es la que proporciona los métodos necesarios para mantener **sesiones**.

Al igual que las **cookies**, las **sesiones** son compartidas por todos los **servlets** de un mismo servidor. De hecho, por defecto se utilizan **cookies** de una forma implícita en el mantenimiento de **sesiones**. Por ello, si el browser no acepta **cookies**, habrá que emplearse las **sesiones** en conjunción con la reescritura de URLs (Ver apartado 10.4).

La forma de obtener una **sesión** es mediante el método **getSession(boolean)** de un objeto **HttpServletRequest**. Si este **boolean** es **true**, se crea una sesión nueva si es necesario mientras que si es **false**, el método devolverá la **sesión** actual. Por ejemplo:

```
...
HttpSession miSesion = req.getSession(true);
...
```

crea una nueva **sesión** con el nombre **miSesion**.

Una vez que se tiene un objeto ***HttpSession***, es posible mantener una colección de pares ***nombre de dato/valor de dato***, de forma que pueda almacenarse todo tipo de información sobre la ***sesión***. Este valor puede ser cualquier objeto de la clase ***Object*** que se desee. La forma de añadir valores a la ***sesión*** es mediante el método ***putValue(String , Object)*** de la clase ***HttpSession*** y la de obtenerlos es mediante el método ***getValue(String , Object)*** del mismo objeto. Esto puede verse en el siguiente ejemplo:

```
...
HttpSession miSesion=req.getSession(true);
CarritoCompras compra = (CarritoCompras) miSesion.getValue(miSesion.getId());
if(compra==null) {
    compra = new CarritoCompras();
    miSesion.putValue(miSesion.getId(), compra);
}
...

```

En este ejemplo, se supone la existencia de una clase llamada ***CarritoCompras***. En primer lugar se obtiene una nueva ***sesión*** (en caso de que fuera necesario, si no se mantendrá una creada previamente), y se trata de obtener el objeto ***CarritoCompras*** añadido a la ***sesión***. Obsérvese que para ello se hace una llamada al método ***getId()*** del objeto ***miSesion***. Cada ***sesión*** se encuentra identificada por un identificador único que la diferencia de las demás. Este método devuelve dicho identificador. Esta es una buena forma de evitar confusiones con el nombre de las ***sesiones*** y el de sus valores. En cualquier caso, al objeto ***CarritoCompras*** se le podía haber asociado cualquier otra clave. Si no se hubiera añadido previamente el objeto ***CarritoCompras*** a la ***sesión***, la llamada al método ***getValue()*** tendría como resultado ***null***. Obsérvese además, que es preciso hacer un ***cast*** para pasar el objeto ***Object*** a objeto ***CarritoCompras***.

En caso de que compra sea ***null***, es decir, que no existiera un objeto añadido previamente, se crea un nuevo objeto ***CarritoCompras*** y se añade a la sesión ***miSesion*** mediante el método ***putValue()***, utilizando de nuevo el identificador de la ***sesión*** como nombre.

Además de estos métodos mencionados, la interface ***HttpSession*** define los siguientes métodos:

- ***getCreationTime()***: devuelve el momento en que fue creado la ***sesión*** (en milisegundos).
- ***getLastAccessedTime()***:devuelve el último momento en que el cliente realizó una petición con el identificador asignado a una determinada ***sesión*** (en milisegundos)
- ***getValueNames()***: devuelve un array con todos los nombres de los objetos asociados con la ***sesión***.
- ***invalidate()***: invalida la ***sesión*** en curso.
- ***isNew()***: devuelve un ***boolean*** indicando si la ***sesión*** es "nueva".
- ***removeValue(String)***: elimina el objeto asociado con una determinada clave.

De todos los anteriores métodos conviene comentar dos en especial: ***invalidate()*** y ***isNew()***.

El método ***invalidate()*** invalida la sesión en curso. Tal y como se ha mencionado con anterioridad, una sesión puede ser invalidada por el propio servidor si en el transcurso de un intervalo prefijado de tiempo no ha recibido peticiones de un cliente. *Iniciar* quiere decir eliminar el objeto ***HttpSession*** y los valores asociados con él del sistema.

El método ***isNew()*** sirve para conocer si una sesión es "nueva". El servidor considera que una sesión es nueva hasta que el cliente se une a la sesión. Hasta ese momento ***isNew()*** devuelve ***true***. Un valor de retorno ***true*** puede darse en las siguientes circunstancias:

- El cliente todavía no sabe nada acerca de la ***sesión***
- La ***sesión*** todavía no ha comenzado.
- El cliente no quiere unirse a la ***sesión***. Ocurre cuando el browser tiene la aceptación de ***cookies*** desactivada.

10.4. Reescritura de URLs

A pesar de que la mayoría de los browser más extendidos soportan las **cookies** en la actualidad, para poder emplear **sesiones** con clientes que o bien no soportan **cookies** o bien las rechazan, debe utilizarse la reescritura de **URLs**. No todos los servidores soportan la reescritura de **URLs**.

Para emplear esta técnica lo que se hace es incluir el código identificativo de la **sesión (sessionId)** en el **URL** de la petición. Los métodos que se encargan de reescribir el **URL** si fuera necesario son **HttpServletResponse.encodeUrl()** y **HttpServletResponse.encodeRedirectUrl()** (sustituidas en el **API 2.1** por **encodeURL()** y **encodeRedirectURL()** respectivamente). El primero de ellos lee un **String** que representa un **URL** y si fuera necesario la reescribe añadiendo el identificativo de la **sesión**, dejándolo inalterado en caso contrario. El segundo realiza lo mismo sólo que con **URLs** de redirección, es decir, permite reenviar la petición del cliente a otro **URL**.

Véase el siguiente ejemplo:

```
...
HttpSession miSesion=req.getSession(true);
CarritoCompras compra = (CarritoCompras)miSesion.getValue(miSesion.getId());

if(compra==null) {
    compra = new CarritoCompras();
    miSesion.putValue(miSesion.getId(), compra);
}
...

PrintWriter out = resp.getWriter();
resp.setContentType("text/html");
...
out.println("Esto es un enlace reescrito");
out.println("<a href=\"" +
resp.encodeUrl("/servlet/buscador?nombre=Pedro")+"\"</a>");
...

```

En este caso, como hay una **sesión**, la llamada al método **encodeUrl()** tendría como consecuencia la reescritura del enlace incluyendo el identificativo de la **sesión** en él.

Parte 4

Interfaz gráfico de usuario

11. JavaScript

11.1. Introducción

JavaScript es un lenguaje de programación que permite el script de eventos, clases y acciones para el desarrollo de aplicaciones Internet entre el cliente y el usuario. *JavaScript* permite con nuevos elementos dinámicos ir más allá de clicar y esperar en una página Web. Los usuarios no leerán únicamente las páginas sino que además las páginas ahora adquieren un carácter interactivo. Esta interacción permite cambiar las páginas dentro de una aplicación: poner botones, cuadros de texto, código para hacer una calculadora, un editor de texto, un juego, o cualquier otra cosa que pueda imaginarse.

Los navegadores interpretan las sentencias de *JavaScript* incluidas directamente en una página HTML, permitiendo la creación de aplicaciones similares a los CGI.

Aún no hay definición clara del *scripting language* ("lenguaje interpretado de comandos"). A veces el término se usa para distinguir este tipo de lenguaje de los lenguajes compilados como el C++. Quizá, algunos lenguajes como el C o C++ puedan ser usados para *scripts* de aplicaciones. *JavaScript* es en muchos aspectos un lenguaje de programación parecido al C o C++.

Como otros lenguajes *script*, *JavaScript* extiende las capacidades de la aplicación con la que trabajan, así *JavaScript* extiende la página Web más allá de su uso normal. Hay numerosas maneras de dar vida al Web y dar flexibilidad al lenguaje. El único límite es la imaginación.

11.1.1. Propiedades del Lenguaje JavaScript

Las propiedades más importantes de *JavaScript* son las siguientes:

- Se interpreta por el ordenador que recibe el programa, no se compila.
- Tiene una programación orientada a objetos. El código de los objetos está predefinido y es expandible. No usa clases ni herencia.
- El código está integrado (incluido) en los documentos HTML.
- Trabaja con los elementos del HTML.
- No se declaran los tipos de variables.
- Ejecución dinámica: los programas y funciones no se chequean hasta que se ejecutan.
- Los programas de *JavaScript* se ejecutan cuando sucede algo, a ese algo se le llama evento.

11.1.2. El lenguaje JavaScript

JavaScript está basado en un modelo orientado al WWW. Elementos de una página como un botón o un cuadro de selección, pueden causar un evento que ejecutará una acción. Cuando ocurre alguno de estos eventos se ejecuta una función en *JavaScript*. Esta función está compuesta de varias sentencias que examinan o modifican el contenido de la página Web, o hacen otras tareas para dar respuesta de algún modo al evento.

Por lo general, los comandos de un programa en *JavaScript* se dividen en 5 categorías:

- Variables y sus valores.
- Expresiones, que manipulan los valores de las variables.
- Estructuras de control, que modifican cómo las sentencias son ejecutadas.
- Funciones, que ejecutan un bloque de sentencias

- Clases y arrays (vectores), que son maneras de agrupar datos.

A continuación se presenta una breve introducción sobre estas categorías.

11.1.3. Variables y valores

En *JavaScript*, a diferencia de la mayoría de los lenguajes de programación, no se debe especificar el tipo de datos. No hay manera de especificar que una variable representa un entero, una cadena de caracteres, un número con decimales (que se escriben con punto y no con coma), o un valor lógico booleano. De hecho, la misma variable puede ser interpretada de diferentes modos en diferentes momentos.

Todas las variables se declaran usando el comando ***var***. Una variable puede ser inicializada cuando se da un valor al ser declarada, o puede no ser inicializada. Además, varias variables pueden ser declaradas a la vez separadas por comas.

Ejemplo 1:

```
var variable1= "coche"
var cuaderno
var mi_variable = 123456, decimal =2342.89
var n_casas, n_habitaciones, n_cuadros, nombre = "Franklin"
```

11.1.4. Sentencias, Expresiones y Operadores

Como en la mayoría de los lenguajes de programación, la unidad básica de trabajo en *JavaScript* es la **sentencia**. Una sentencia de *JavaScript* hace que algo sea evaluado. Esto puede ser el resultado de dar valor a una variable, llamar a una función, etc. Cualquiera de las líneas del ejemplo 1 es una sentencia.

Los programas de *JavaScript* son un grupo de sentencias, normalmente organizadas en funciones que manipulan las variables y el entorno HTML en el cual el *script* trabaja.

Los operadores hacen que en una sentencia las variables sean evaluadas y se les asigne un valor o un resultado. Los operadores pueden actuar de distinto modo en diferentes situaciones. Algunos operadores de *JavaScript* pueden ser sobrecargados, es decir, pueden tener diversas interpretaciones según su modo de uso.

No hay ningún carácter especial o signo que marque el final de una sentencia en un programa. Por defecto se considera que una sentencia ha acabado cuando se llega al final de la línea, aunque se puede especificar el fin con el carácter punto y coma (;). Ésto hace posible poner varias sentencias en una sola línea separadas entre sí por un punto y coma.

Las dos siguientes sentencias realizan la misma operación, para *JavaScript* no tienen ninguna diferencia

Ejemplo 2:

```
var variable1= "coche"
var variable1= "coche";
```

y estos dos grupos de sentencias también:

Ejemplo 3:

```
var a = 0; a = 2+4; var c = a / 3 ;;
var a = 0
a = 2+4
var c = a / 3
```

Los segunda sentencia es una expresión.

11.1.5. Estructuras de Control.

Con lo explicado, aún no es posible escribir código para un programa completo; hay que conocer construcciones de nivel más elevado. Existen varios métodos para controlar el modo de ejecución de sentencias que se verán más adelante.

11.1.6. Funciones y Objetos

Las sentencias, expresiones y operadores básicos se agrupan en bloques más complejos dentro de un mismo programa llamadas funciones. El control de estructuras representa el siguiente nivel de organización de *JavaScript*. Las funciones y los objetos representan el nivel más alto de organización del lenguaje.

11.1.6.1. Funciones

Una función es un bloque de código con un nombre. Cada vez que se usa el nombre, se llama a la función y el código de la función es ejecutado. Las funciones pueden llamarse con valores, conocidos como parámetros, que se usan en la función. Las funciones tienen dos objetivos: organización del programa (archivo o documento) y ejecución del código de la función. Al clicar con el ratón, apretar un botón, seleccionar texto y otras acciones pueden llamar a funciones.

El nombre de una función se escribe inmediatamente después del comando **function**. Todos los nombres de funciones deben ser únicos y diferentes de los nombres de los comandos que usa *JavaScript*. No puede haber dos funciones con el mismo nombre. La lista de parámetros de una función se separa por comas. La función usa esos parámetros en las sentencias de su cuerpo que la configuran. Los argumentos que se le pasan a una función no pueden ser cambiados en su interior.

Ejemplo 4:

```
function valor_abs (num){  
    if (num >= 0)  
        return num  
    else  
        return -num  
}
```

num es el argumento que se utiliza dentro de la función. El código de la función va entre llaves.

11.1.6.2. Objetos

Las funciones se usan para organizar el código. Los objetos tienen el mismo propósito pero con datos. Los tipos de datos conocidos hasta ahora son variables declaradas o inicializadas con **var**. Cada uno de estos tipos puede tener un solo valor. Los objetos permiten la capacidad de tener varios valores, de tal manera que un grupo de datos pueda estar relacionado con otro.

Lo que en *JavaScript* se llama objeto en otros lenguajes se llama estructura de datos o clase. Como las funciones, los objetos tienen 2 aspectos: cómo se crean y cómo se usan.

Al usar *JavaScript*, tenemos predefinidos una serie de objetos. Un objeto de *JavaScript* es un conjunto de componentes, llamados propiedades o miembros. Si se supone que se tiene un objeto llamado *cita* para organizar citas, éste tendrá las propiedades día, hora, con quién y para qué.

A cada una de estas propiedades del objeto se hace referencia con el operador punto (.).

Así, para referirse al mes de la cita se usa *una_cita.mes* mientras que *una_cita.con_quien* contendrá el nombre de con quién nos hemos citado.

Cada objeto puede contener todas las variables que nos interesen. Puede contener funciones que realicen algún trabajo. Puede incluso contener otros objetos, de tal manera que se pueden organizar los datos de modo jerárquico.

Más adelante se verán ejemplos al entrar en detalle con los objetos.

11.1.7. La TAG «Script».

En el sentido más general, cada página Web está hecha con sentencias de HTML que dividen la página en dos partes: el HEAD y el BODY.

La sección HEAD de un documento HTML es la que debe contener el código de *JavaScript* para los gestores de eventos. Aunque no es necesario que todo el código de *JavaScript* vaya en el HEAD, es importante que vaya en él para asegurar que todo el código de *JavaScript* haya sido definido antes del BODY del documento. En particular, si el documento tiene código para ejecutar un evento, y este evento se acciona antes de que el código se lea, podría ocurrir un error por que la función está sin definir.

En un documento HTML, el código de *JavaScript* se introduce mediante la TAG SCRIPT. Todo lo que haya entre <SCRIPT> y </SCRIPT> se considera como un tipo de código script, como *JavaScript*. La sintaxis para la TAG SCRIPT es:

```
<SCRIPT TYPE="Nombre del lenguaje" SRC="URL">  
El elemento Nombre del lenguaje da el lenguaje que se usa en el subsiguiente script.
```

El atributo SRC es necesario cuando se quiere hacer referencia a un fichero que contiene el código del script. Para *JavaScript*, el fichero suele tener extensión *.js*. Si se usa el atributo SRC la TAG <SCRIPT> es

inmediatamente seguida por </SCRIPT>. Por ejemplo un bloque <SCRIPT> que carga un código *JavaScript* del fichero click.js en el directorio relativo al documento, se haría del siguiente modo:

Ejemplo 5

```
<SCRIPT TYPE="text/javascript" SRC="click.js"></SCRIPT>
```

Si no se pone el atributo SRC, entonces entre las TAGS <SCRIPT> y </SCRIPT> se escribe el código en el lenguaje indicado en la primera TAG. La estructura general del código es:

```
<SCRIPT TYPE="text/javascript">
    Sentencias
</SCRIPT>
```

El siguiente programa usa la función del ejemplo 4. Con el formulario, <FORM>, nos pide un número que evaluamos con la función y devolvemos su valor absoluto. Puede que haya cosas que no queden claras, pero todo se explicará con más detalle a posteriori. La función de estos ejemplos previos no es sino la de dar una visión general del lenguaje.

Ejemplo 6:

```
<HTML>
<BODY>
<SCRIPT TYPE="text/javascript">
function valor_abs(form){
    var num = eval(form.expr.value)
    if (num >= 0)
        form.result.value = num
    else
        num = -num
        form.result.value = num
}
</SCRIPT>
<FORM>
    <SCRIPT>
        document.write("Introduce un número:") // Salida por pantalla
    </SCRIPT>
    <INPUT TYPE="text" NAME="expr" SIZE=15 ><BR>
    <INPUT TYPE="button" VALUE="Calcular" onClick="valor_abs(this.form)"><BR>
    <P ALIGN=LEFT>
        <SCRIPT>document.write("Valor Absoluto:")</SCRIPT>
        <INPUT TYPE="text" NAME="result" SIZE=15 >
    </P>
</FORM>
</BODY>
</HTML>
```

11.2. Activación de JavaScript: Eventos (Events).

El número de cosas que se pueden hacer en una página HTML es bastante limitado. La mayoría de los usuarios simplemente leen un texto, miran gráficos y como mucho escuchan sonidos. Para muchos, la experiencia del Web consiste en visitar una serie de páginas sin interaccionar prácticamente con ellas. La única interacción ocurre cuando el usuario selecciona un link o clica un mapa de imagen.

Los formularios de HTML han cambiado gradualmente este modelo para incrementar el nivel de interacción. Un formulario tiene varios modos de aceptar entradas. El usuario rellena el formulario y lo envía. Es difícil saber si el formulario ha sido rellenado correctamente y el tiempo de proceso del formulario es normalmente bastante largo. En el caso del HTML, este proceso ocurre porque el contenido del formulario tiene que ser enviado a través de la red a algún fichero en el servidor, donde se procesa y entonces se da una respuesta al usuario. Incluso el más simple error causa el rechazo del formulario, y por lo tanto que deba repetirse el proceso.

Uno de los objetivos de *JavaScript* es localizar la mayoría de estos procesos y mejorarlos dentro del browser del usuario. *JavaScript* es capaz de asegurarse que un formulario se rellene y envíe correctamente; evitando que el usuario tenga que repetir el formulario a causa de algún error.

JavaScript realiza esto mediante los gestores de eventos. Estos son sentencias de *JavaScript*, normalmente funciones, que se llaman cada vez que algo ocurre. Las funciones de *JavaScript* pueden ser llamadas cuando se envía un formulario o cuando el usuario usa campos del formulario.

11.2.1. Eventos y acciones

Para entender el modelo de gestores de eventos de *JavaScript*, hay que pensar primero sobre las cosas que pueden ocurrir actualmente en una página Web. Aunque algunas cosas se pueden hacer con el browser, la mayoría de estas no tienen que ver con la navegación en el Web (salvar una página como texto, imprimirla, editar un bookmark, etc.).

Para entender qué acciones del browser corresponden a los eventos de *JavaScript* y cuales no, es importante distinguir aquellas acciones que causan algún cambio en la página Web cuando se muestra. De hecho, realmente hay sólo dos tipos de acciones: las que permiten que el usuario pueda navegar o las que hacen posible que el usuario pueda interactuar con un elemento de un formulario HTML.

11.2.1.1. Acciones de Navegación y Eventos

En la categoría de navegación se pueden distinguir las siguientes acciones:

- Seleccionar un link de hipertexto
- Mover hacia adelante o hacia atrás en la lista de Webs visitados.
- Abrir otro fichero.
- Cerrar el browser

En la mayoría de estos casos la página activa se descarga, y otra nueva se carga y se muestra en la ventana del browser. Pero cualquier persona que ha usado la WWW sabe que al seleccionar un link de hipertexto no siempre se tiene éxito, puesto que la máquina puede estar desconectada o inaccesible. El link puede haber muerto. Seleccionando un link muerto se descarga la página activa, y no se carga una nueva.

Dependiendo del tipo de error y del browser usado puede perderse la página activa. Estos eventos, cargar y descargar una página, son los únicos eventos que pueden ser manejados por *JavaScript* a nivel de los documentos. Esto significa que es posible escribir código *JavaScript* contenido dentro de la definición de HTML de una página, que se ejecutará cada vez que la página sea cargada o descargada.

11.2.2. Gestores de Eventos (*Event Handlers*)

11.2.2.1. Declaración

En la introducción se ha dicho que las funciones de *JavaScript* sólo se ejecutan en respuesta a eventos. Se sabe que los eventos ocurren cuando se produce alguna interacción o cambio en la página Web activa.

Las declaraciones de los gestores de eventos es muy similar a los atributos de HTML. Cada nombre del atributo empieza con la palabra *on* y sigue con el nombre del evento, así por ejemplo *onClick* es el atributo que se usaría para declarar un gestor de eventos para el evento Click (clicar un objeto).

La declaración de un gestor de eventos es: *onEvent="Código_JS"*.

Normalmente, por convenio, se escribe *on* en minúscula y el nombre del evento con la letra inicial en mayúscula. Esto ayuda a distinguir éste de los demás atributos.

Los tipos de eventos y gestores de eventos son los siguientes:

Evento	Ocurre Cuando	Gestor
blur	El usuario quita el cursor de un elemento de formulario	onBlur
click	El usuario clica un link o un elemento de formulario	onClick
change	El usuario cambia el valor de un texto, un área de texto o selecciona un elemento.	onChange
focus	El usuario coloca el cursor en un elemento de formulario.	onFocus
load	El usuario carga una página en el Navegador	onLoad

Mouseover	El usuario mueve el ratón sobre un link	onMouseOver
Select	El usuario selecciona un campo del elemento de un formulario	onSelect
Submit	Se envía un formulario	onSubmit
Unload	Se descarga la página	onUnload

El valor del atributo es un conjunto de código *JavaScript* o una referencia a una función de *JavaScript*. El código o la función se ejecuta al activar el evento.

Ejemplo 7:

```
<INPUT TYPE="button" NAME="mycheck" VALUE="HA!"  
onClick="alert('Te he dicho que no me aprietas')">
```

Esta sentencia crea un botón (*INPUT TYPE="button"*). Al clicar el botón, el gestor de eventos *onClick* despliega una ventana con el mensaje que se pasa como argumento.

Normalmente una página HTML con programación en *JavaScript* tiene los siguientes componentes:

- Funciones *JavaScript* dentro de un bloque Script dentro del *<HEAD>* del documento.
- HTML no interactivo dentro del *<BODY>* del documento
- HTML interactivo con atributos gestores de eventos cuyos valores son funciones de *JavaScript*.

En general, ya sabemos declarar gestores de eventos. Ahora se verá qué gestores de eventos pueden asociarse con los TAGs específicos de HTML.

Los eventos de *JavaScript* suceden en tres niveles: a nivel del documento Web, a nivel de un formulario individual dentro del documento y a nivel de un campo de formulario.

11.2.2.2. Uso

11.2.2.2.1. Gestores a nivel de documento

La TAG HTML BODY contiene la descripción del contenido de la página HTML. La TAG BODY puede contener dos declaraciones de gestores de eventos usando los atributos *onLoad* y *onUnload*. Una declaración podría ser:

Ejemplo 8:

```
<BODY onLoad="cargarfuncion()" onUnload="descargarfuncion()">
```

El atributo *onLoad="cargarfuncion()"* declara un gestor de *JavaScript* que manejará la carga. El evento *load* se genera después de que el contenido de la página entre *<BODY>* y *</BODY>* se haya leído pero antes de que se haya mostrado. El gestor de evento *onLoad* es un buen lugar para mostrar el nombre de la compañía o la información de copyright, una ventana de seguridad preguntando el password de autorización, etc.

El atributo *onUnload="descargarfuncion()"* declara un gestor de eventos que se llama cada vez que la página se descarga. Esto ocurre cuando se carga una página nueva en la misma ventana, si una página no se carga con éxito y la página activa está aun descargada. El gestor de eventos *onUnload* puede servir para asegurarse de que no se ha perdido contacto con la página, por ejemplo si un usuario ha llenado un formulario pero se ha olvidado de mandarlo.

Eventos aplicados a las TAGs de HTML:

- FOCUS, BLUR, CHANGE: campos de texto, áreas de texto y selecciones.
- CLICK: botones, botones de tipo radio, cajas de chequeo, botón de envío, botones de reset y links.
- SELECT: campos de texto, áreas de texto, cuadros de selección.
- MOUSEOVER: links.

El evento **focus** se genera cuando el ítem de texto de un elemento de la lista obtiene el foco, normalmente como resultado de clicar con el ratón. El evento **blur** se genera cuando un ítem pierde el

foco. El evento **change** se genera cada vez que un ítem sufre algún cambio. En un ítem de texto esto resulta cuando se introduce nuevo texto o el que existía se borra. En una lista de selección ocurre cada vez que una nueva selección se hace, incluso en una lista que permite múltiples selecciones. El evento **select** se genera cuando el usuario selecciona algún texto o hace una selección en el cuadro de selección.

Estos eventos pueden usarse para obtener un buen control sobre el contenido de un texto o una lista de selección de ítems. Las aplicaciones más comunes usan el evento **change** o **blur** para asegurarse de que el texto tiene el valor apropiado.

El argumento/comando especial **this**: Este comando se usa para referirse al objeto activo. Cuando la función a la que se le pasa el argumento **this** es llamada, el parámetro que usa la función en su definición se introduce con el objeto sobre el que se actúa.

Si nos fijamos en el ejemplo 9, la función **cambiar()** se activa cuando el formulario , en este caso un cuadro de selección, pierde el foco. A la función se le pasa como argumento el formulario mediante el comando **this**. **form.cap.selectedIndex** es el índice de la selección escogida del formulario (**form**) llamado **cap**.

11.2.2.2. Gestores a nivel de formulario

La TAG FORM se usa para comenzar la definición de un formulario HTML. Incluye atributos como el METHOD usado para elegir el modo de envío del formulario, la acción que se debe cumplir (ACTION) y el atributo onSubmit. La sintaxis es como la que sigue:

```
<FORM NAME="nombre_del_formulario" ... onSubmit="función_o_sentencia">
```

El gestor onSubmit es llamado cuando el contenido del formulario se envía. También es posible especificar una acción onClick en un botón de envío. El uso común del gestor onSubmit es verificar el contenido del formulario: el envío continúa si el contenido es válido y se cancela si no lo es.

11.2.2.3. Gestores a nivel de elementos de formulario

Casi todos los elementos de un formulario tienen uno o más gestores de eventos. Los botones pueden generar eventos click, el texto y la selección de elementos pueden generar los eventos **focus**, **blur**, **select** y **change**.

Hay dos excepciones a la regla que todos los elementos de un formulario pueden tener gestores de eventos. La primera excepción se aplica a los ítems ocultos, <INPUT TYPE= "hidden">. No se ven, no se pueden cambiar y no pueden generar eventos. La segunda se aplica a los elementos individuales OPTION dentro de una lista de selección (que se crean con la opción SELECT). La TAG SELECT puede tener atributos declarando gestores de eventos (**focus**, **blur** y **change**), pero las OPTION no pueden generar eventos.

Los campos de texto (text fields) de HTML, <INPUT> con el atributo TYPE de texto "text" pueden declarar gestores de eventos como combinación de los 4 elementos de texto: **focus**, **blur**, **change** y **select**. Con la TAG TEXTAREA se crea la entrada de texto en varias líneas y pueden generarse estos gestores de eventos. En cambio la selección de listas creadas con <SELECT> pueden generar todos los eventos menos el **select**.

Estos eventos pueden usarse para obtener un buen control sobre el contenido de un texto o una lista de selección de ítems. Las aplicaciones más comunes usan el evento **change** o **blur** para asegurarse de que el campo tiene el valor apropiado.

Ejemplo 9:

```
<HTML><HEAD>
<TITLE>EJEMPLO DEL COMANDO this</TITLE>
<SCRIPT type="text/javascript">
function cambiar(form){
    var indice = form.cap.selectedIndex
    if(indice==0){var archivo="cap1.htm"}
    if(indice==1){var archivo="cap2.htm"}
    if(indice==2){var archivo="cap3.htm"}
    window.open(archivo,'capitulos')
    window.open('marcador.htm','resultados')
```

```

    }
</SCRIPT>
</HEAD>
<BODY>
<CENTER><FORM>
<SELECT NAME="cap" SIZE=1 onBlur="cambiar(this.form)">
<OPTION VALUE=1>1. HISTORIA Y CONCEPTOS DE LA AP</OPTION>
<OPTION VALUE=2>2. MÉTODOS EN PATOLOGÍA</OPTION>
<OPTION VALUE=3>3. PATOLOGÍA MOLECULAR</OPTION>
<SELECT>
</FORM></CENTER>
</BODY></HTML>

```

11.3. Clases en JavaScript

Las clases en *JavaScript* se pueden agrupar en tres categorías:

- Clases Predefinidas, incluyen las clases Math, String y Date.
- Clases del Browser, tienen que ver con la navegación.
- Clases del HTML, están asociadas con cualquier elemento de una página Web (link, formulario, etc.).

11.3.1. Clases Predefinidas (*Built-In Objects*).

11.3.1.1. Clase String

Cada vez que se asigna un valor string (cadena de caracteres a una variable o propiedad, se crea un objeto de la clase string. Al asignar un string a una variable no se usa el operador new.

Los objetos string tienen una propiedad, length (número de caracteres de la cadena), y varios métodos que manipulan la apariencia de la cadena (color, tamaño, etc.).

Métodos sobre el contenido: (recordar que las string tienen como base de índices el cero.)

- *charAt* (indice), muestra el carácter que ocupa la posición indice en la cadena.
- *indexOf* (carácter), muestra el primer índice del carácter.
- *lastIndexOf* (carácter), muestra el último carácter del índice.
- *subString* (primerindice, ultimoindice), muestra la cadena que hay que hay entre el primer índice (primerindice) y el último índice (ultimoindice) incluídos.
- *toLowerCase()*, muestra todos los caracteres de la cadena en minúsculas.
- *toUpperCase()*, muestra todos los caracteres de la cadena en mayúsculas.

Suponiendo que la variable *cadena* es un objeto de la clase string, el uso de los métodos se realiza de la siguiente manera: *cadena.método()*.

Métodos sobre la apariencia:

- *big* (), muestra las letras más grandes.
- *blink* (), muestra texto intermitente (parpadeando).
- *bold* (), muestra las letras en negrita.
- *fixed* (), muestra el texto en paso fijo (letra Courier New).
- *fontcolor* (color), cambia el color de las letras.
- *fontsize* (size), cambia el tamaño de las letras.
- *italics* (), muestra en letra itálica.

- *small ()*, muestra las letras más pequeñas.
- *strike ()*, muestra las letras tachadas por una ralla.
- *sub ()*, muestra la letra en subíndice.
- *sup ()*, muestra la letra en superíndice.

Ejemplo 10:

```
var cadena = "Mira hacia aquí".
cadena.charAt( 2 ) = "r"
cadena.indexOf( i ) = 1
cadena.lastIndexOf( a ) = 11
cadena.substring( 5, 9 )
cadena.toLowerCase() = "mira hacia aquí"
cadena.toUpperCase() = "MIRA HACIA AQUÍ"
```

Métodos sobre el HTML:

- *anchor (nombre_string)*, este método crea un ancla, llamada nombre_string como valor para el atributo NAME.
- *link (href_string)*, este método crea un link a un URL designado por el argumento href_string.

Ejemplo 11:

```
cadena.big() = "Mira hacia aquí".
cadena.blink() = "Mira hacia aquí". cadena.blink() = "
cadena.bold() = "Mira hacia aquí".
cadena.fixed() = "Mira hacia aquí". // Éste es el tipo de letra Courier New
cadena.fontcolor("red") = "Mira hacia aquí".
cadena.fontSize(3) = "Mira hacia aquí".
cadena.italics() = "Mira hacia aquí".
cadena.small() = "Mira hacia aquí".
cadena.strike() = "Mira hacia aquí".
cadena.sup() = "Mira hacia aquí".
cadena.sub() = "Mira hacia aquí".
```

11.3.1.2. Clase Math

La clase Math se usa para efectuar cálculos matemáticos. Contiene propiedades generales como *pi* = 3.14159..., y varios métodos que representan funciones trigonométricas y algebraicas. Todos los métodos de Math pueden trabajar con decimales. Los ángulos se dan en radianes, no en grados.

La clase Math es el primer ejemplo de clase estática (que no cambia). Todos sus argumentos son valores. Esta clase no permite crear objetos, por lo que hay que referirse directamente a la clase para usar los métodos.

Propiedades (se usan del modo *Math.propiedad*):

- *E*, número "e". Es un número tal que su logaritmo neperiano es 1, $\ln(e) = 1$
- *LN10*, logaritmo neperiano del número 10.
- *LN2*, logaritmo neperiano del número 2.
- *PI*, número pi = 3.14159...
- *SQRT1_2*, raíz cuadrada de $\frac{1}{2}$.
- *SQRT2*, raíz cuadrada de 2.

Métodos:

- *abs (numero)*, calcula el número absoluto de numero.

- *acos* (numero), calcula el ángulo cuyo coseno es numero.
- *asin* (numero), calcula el ángulo cuyo seno es numero.
- *atan* (numero), calcula el ángulo cuya tangente es numero.
- *ceil* (numero), calcula el entero mayor o igual que numero.
- *cos* (angulo), calcula el coseno de angulo.
- *exp* (numero), calcula el número e elevado a la potencia numero.
- *floor* (numero), calcula el entero menor o igual que numero.
- *log* (numero), calcula el logaritmo natural de numero.
- *max* (numero1, numero2), calcula el máximo entre numero1y numero2.
- *min* (numero1, numero2), calcula el mínimo entre numero1y numero2.
- *pow* (numero1, numero2), calcula numero1 exponentado a numero2.
- *random* (), calcula un número decimal aleatorio entre 0 y 1, SÓLO PARA UNIX.
- *round* (numero), devuelve el entero más cercano a numero.
- *sin* (angulo), calcula el seno de angulo.
- *sqrt* (numero), calcula la raíz cuadrada de numero.
- *tan* (angulo), calcula la tangente de angulo.

Ejemplo 12:

```
Math.abs(-4) = 4
Math.abs(5) = 5
Math.max(2,9).=.9
Math.pow(3,2) = 9
Math.sqrt(144) = 12
```

11.3.1.3. Clase Date

Una de las cosas más complicadas de cualquier lenguaje es trabajar con fechas. Esto es porque hay gente que para representar fechas y horas toma un sistema no decimal (los meses en unidades sobre 12, las horas sobre 24 y los minutos y segundos sobre 60). Para el ordenador es ilógico trabajar con números bonitos y redondeados.

La clase date simplifica y automatiza la conversión entre las representaciones horarias del ordenador y la humana.

La clase date de *JavaScript* sigue el estándar de UNIX para almacenar los datos horarios como el número de milisegundos desde el día 1 de enero de 1970 a las 0:00. Esta fecha se denomina "la época".

Aunque la clase date no tiene propiedades, tiene varios métodos. Para usar la clase date hay que entender cómo construir un objeto de esta clase. Para eso hay tres métodos:

- *new Date()*, inicializa un objeto con la hora y fecha actual.
- *new Date(string_dato)*, inicializa un objeto con el argumento string_dato. El argumento debe ser de la forma "Mes día, año" como "Noviembre 29, 1990".
- *new Date(año, mes, día)*, inicializa un objeto tomando 3 enteros que representan el año, mes y día. NOTA: los meses tienen como base el 0, lo que significa que 2 corresponde con el mes de marzo y 10 con el mes de noviembre.

Ejemplo 13:

```
var dato = new Date(90, 10, 23)
var dato = new Date(1990, 10, 23)
```

Estas dos declaraciones se refieren a la fecha del 23 de noviembre de 1990.

Hay un modo opcional para declarar la hora además de la fecha. Hay poner 3 argumentos más a la vez que se ponen los argumentos de la fecha.

Ejemplo 14:

```
var dato2 = new Date(90, 10, 23, 13, 5, 9)
```

Esta declaración se refiere a la 1:05:09 PM del día 23 de noviembre de 1990.

Métodos:

- *getDate()*, devuelve el número de día del mes (1-31).
- *getDay()*, devuelve el número de día de la semana (0-6).
- *getHours()*, devuelve el número de horas del día (0-23).
- *getMinutes()*, devuelve el número de minutos de la hora (0-59)
- *getMonth()*, devuelve el número de mes del año (0-11).
- *getSeconds()*, devuelve el número de segundos del minuto (0-59)
- *getTime()*, devuelve la hora.
- *getYear()*, devuelve el año.
- *setDate()*, fija la fecha.
- *setHours()*, fija el número de horas del día.
- *setMinutes()*, fija el número de segundos del minuto.
- *setMonth()*, fija el número de mes.
- *setSecond()*, fija el número de los segundos del minuto.
- *setTime()*, fija la hora.
- *setYear()*, fija el año.

Ejemplo 15:

```
var dato2 = new Date(90, 10, 23, 13, 5, 9)
dato2.getHours() = 13
dato2.getDay() = 0 (si es lunes), 1 (si es martes), etc.
dato2.getSeconds = 9
dato2.setMonth() = 2 // el mes cambia a marzo
dato2.setYear() = 96
```

11.3.2. Funciones Predefinidas (Built-in Functions): eval, parseFloat, parseInt.

Además de los clases **String**, **Math** y **Date** hay un pequeño conjunto de funciones predefinidas por **JavaScript**. No son métodos ni se aplican a los objetos. Tiene el mismo comportamiento que cuando se crean funciones con el comando **function**.

11.3.2.1. eval(string)

Esta función intenta evaluar su argumento de tipo string como una expresión y devolver su valor. Esta función es muy potente porque evalúa cualquier expresión de **JavaScript**.

Ejemplo 16:

```
w = (x * 14) - (x / z) + 11
z = eval ("(x * 14) - (x / z) + 11")
```

Si x es una variable con el valor 10 las siguientes expresiones asignan 146 a w y z.

11.3.2.2. parseFloat(string)

Esta función intenta convertir su argumento de tipo string como un número decimal (de coma flotante).

Ejemplo17:

```
parseFloat( "3.14pepecomeperas345" ) = 3.14
```

11.3.2.3. parseInt(string, base)

Esta función se comporta de forma muy similar a la anterior. Intenta convertir su argumento de tipo string como un entero en la base elegida.

Ejemplo 18:

```
ParseInt(10111, 2)= 23
```

Se convierte el número 10111 en base binaria.

11.3.3. Clases del browser

El modelo de clases de *JavaScript* y su conjunto de clases, métodos y funciones predefinidas dan un moderno lenguaje de programación. Puesto que *JavaScript* se diseñó para trabajar con y en el World Wide Web tuvo que haber un nexo entre *JavaScript* y el contenido de las páginas HTML. Este nexo viene dado por un conjunto de clases del browser y del HTML.

Las clases del browser son una extracción del entorno del browser e incluye clases para usar en la página actual, en la lista de documentos visitados (history list) y en el URL actual. Existen métodos para abrir nuevas ventanas, mostrar cajas de diálogos y escribir directamente HTML (en algunos ejemplos se ha usado el método **write** de la clase **document**).

Las clases del browser (o navegador) son el nivel más alto de la jerarquía de objetos de *JavaScript*. Representan información y acciones que no necesariamente hay que asociar con una página Web. Dentro de una página Web cada elemento HTML tiene su objeto correspondiente, un objeto HTML dentro de la jerarquía de objetos. Cada formulario HTML y cada elemento dentro de un formulario HTML tiene su correspondiente objeto.

11.3.3.1. Clase Window

Es el nivel más alto de la jerarquía de objetos de *JavaScript*. Cada ventana de un browser que está abierta tiene su correspondiente objeto window. Todo el resto de objetos desciende del objeto window. Normalmente, cada ventana se asocia a una página Web y la estructura HTML de esa página se refleja en el objeto *document* de la ventana. Cada ventana se corresponde con algún URL que se refleja en el objeto location. Cada ventana tiene una lista de documentos visitados que se han mostrado en esa ventana (history list), las cuales se representan por varias propiedades del objeto history.

Los métodos de un objeto window son:

- *alert(string_mensaje)*
- *confirm(string_mensaje)*
- *open(URL_string, nombre_ventana)*
- *close()*
- *prompt(string_mensaje)*

Ejemplo 19:

```
alert("No clicar el botón izquierdo")
confirm("¿Quieres continuar?")
window.open("fichero.htm", ventana_1)
window.close() // cierra una ventana
prompt("Rellena el cuestionario")
```

Todos estos métodos se usan para manipular el estado de la ventana del browser. Los métodos alert y confirm se usan para mostrar su argumento string_mensaje en una caja de diálogo. El método alert se usa para avisar al usuario sobre algo que no debe hacer. La caja de diálogo de alert contiene el botón OK,

mientras que la de confirm muestra el mensaje con un botón OK y otro Cancel. Devuelve, como valor de retorno, **true** si se clica OK o **false** si se clica Cancel.

El método **prompt** se usa para solicitar al usuario una entrada de datos, en forma de cadena de caracteres. Muestra una caja de diálogo con el string_mensaje y un campo de texto editable. Este método acepta un segundo argumento opcional que se usa para fijar un valor por defecto en el campo de texto. Devuelve lo que escribe el usuario en el campo de texto.

El método **open** se usa para abrir una ventana nueva en el browser. El argumento URL_string representa el URL que será cargado en la ventana donde el otro argumento nombre_ventana da nombre a la ventana. Este método devuelve una instancia del objeto window que representa la nueva ventana creada. Este método también acepta un tercer argumento opcional que se usa para especificar los modos de mostrar la nueva ventana. Cuando el método **close** se llama desde un ejemplar del objeto window, esa ventana se cierra y el URL se descarga.

11.3.3.2. Clase Document

Cada ventana se asocia con un objeto document. El objeto document contiene propiedades para cada ancla, link, y formulario en la página. También contiene propiedades para su título, color de fondo, colores de los links y otros atributos de la página. El objeto document tiene los siguientes métodos:

- **clear()**
- **close()**
- **open()**
- **write(string)**
- **writeln(string)**

El método **clear** se usa para borrar completamente un documento. Tiene mucho uso si se está construyendo una página Web sólo con *JavaScript*, y se quiere asegurar que está vacía antes de empezar. Los métodos **open** y **close** se usan para empezar y parar la salida de datos a memoria. Si se llama al método **open**, se ejecutan series de **write** y/o **writeln**, y se llama al método **close**, el resultado de las operaciones que se han escrito se muestran en la página.

El método **write** se usa para escribir cualquier cadena de caracteres, incluyendo programación HTML, al documento actual. Este método puede usar un número variable de argumentos. El método **writeln** es idéntico al método **write**, excepto que en la salida imprime un salto de línea al acabar de escribir sus argumentos. Hay que notar que el salto de línea será ignorado por el browser, el cual no incluye espacios en blanco, a menos que el **writeln** esté dentro de texto preformateado.

Ejemplo 20:

```
document.clear()
document.close()
document.open()
document.write("Juan come peras") -> Juan come peras
document.writeln("Juan come peras") -> Juan come peras
```

11.3.3.3. Clase Location

El objeto location describe el URL del documento. Este tiene propiedades representando varios componentes del URL, incluyendo su parte de protocolo, de hostname, de pathname, de número de puerto, entre otras propiedades. También tiene el método **toString** el cual se usa para convertir el URL a una cadena de caracteres. Para mostrar el URL actual podemos usar el siguiente código:

Ejemplo 21:

```
var lugar = document.location
document.write("<BR>El URL actual es " + lugar.toString())
document.write("<BR>")
```

11.3.3.4. Clase History

El objeto history se usa para referirse a la lista de URLs visitados (history list) anteriormente. Tiene una propiedad conocida como `length`, la cual indica cuántos URLs están presentes en la history list actualmente. Tiene los siguientes métodos:

- `back()`
- `forward()`
- `go(donde)`

El método `go` se usa para navegar en la history list. El argumento `donde` puede ser un número o un string. Si el argumento `donde` es un número indica el número de orden del lugar donde se desea ir en la history list. Un número positivo significa que avance tantos documentos como indique el número, y un número negativo significa que se atrase tantos documentos como indique el número. Si `donde` es una cadena de caracteres que representa un URL, entonces pasa a ser como el documento actual.

Ejemplo 22:

```
history.back()
history.forward()
history.go(-2)
history.go(+3)
```

11.3.4. Clases del documento HTML (*anchors, forms, links*)

Para entender cómo trabajan los objetos HTML en *JavaScript*, hay que considerar ciertas piezas de HTML que crean un ancla, un formulario y un link a este ancla. Para aclarar estos conceptos nos basamos en el ejemplo 23.

Este código crea una página HTML con un ancla al principio de la página y un link al ancla al final. Entre ambas hay un simple formulario que permite al usuario poner su nombre. Hay un submit button (botón de envío) por si se quiere enviar y un botón de reset por si no se quiere enviar. Si el usuario envía con éxito el contenido del formulario vía post al e-mail ficticio `nobody@dev.null`.

Ejemplo 23:

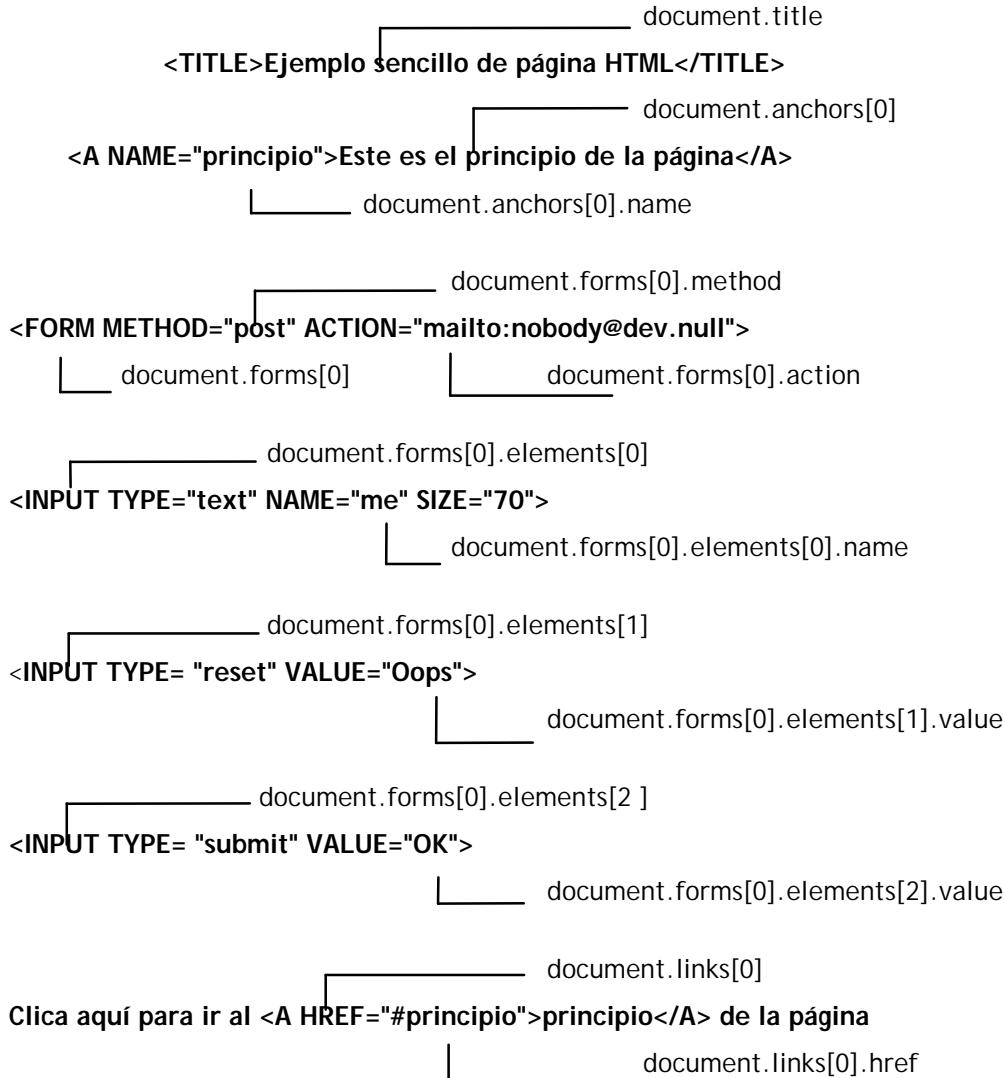
```
<HTML>
<HEAD><TITLE>Ejemplo sencillo de página HTML</TITLE></HEAD>
<BODY>
<A NAME="principio">Este es el principio de la página</A> <HR>
<FORM METHOD="post">
<P> Introduzca su nombre: <INPUT TYPE="text" NAME="me" SIZE="70"></P>
<INPUT TYPE= "reset" VALUE="Borrar Datos">
<INPUT TYPE= "submit" VALUE="OK">
</FORM><HR>
Clica aquí para ir al
<A HREF="#principio">principio</A> de la página // link
<BODY>
</HTML>
```

El aspecto más importante de este ejemplo es el hecho de que los elementos HTML se reflejan en la jerarquía de objetos de *JavaScript*. Se puede acceder al título del documento a través de la propiedad `title` del objeto documento. Se puede acceder a otros elementos HTML de este documento usando las siguientes propiedades:

- `anchors(anclas)`
- `forms(formularios)`
- `links`

Estas propiedades del objeto document son arrays que representan cada elemento HTML como un ancla, formulario o link de una página. En el ejemplo, el ancla en el principio de la página se referiría como `document.anchors[0]`, el link al final de la página como `document.links[0]`, y el formulario en medio de la página como `document.forms[0]`. Estos son el nivel más alto de los objetos representados por este documento. Cada uno de estos elementos tiene propiedades y métodos que se usan para describir y manipularlos.

El objeto form correspondiente a `forms[0]` tiene sub-objetos para cada uno de los tres elementos (el botón de reset, el botón de envío y el campo de texto) y propiedades para el método submit. `forms[0].elements[0]` corresponde a la entrada del campo de texto. `forms[0].elements[0].name` es el nombre de este campo, como el especificado por el atributo NAME, el cual en este caso es "me". La siguiente figura representa el código HTML del ejemplo y muestra cómo cada elemento en la página se asocia con el objeto HTML.



11.4. Clases y Funciones definidas por el usuario.

Para entender este tipo de programación vamos a ver un ejemplo donde creamos un objeto llamado casa que tiene las siguientes propiedades: nº de habitaciones, año de construcción, ¿tiene garaje?, estilo arquitectónico. Para definir un objeto para guardar esta información (las propiedades) hay que hacer una función que las muestre en un listado. Nota: Esta función usa el comando this, que hace referencia al objeto activo. En este caso hace referencia al objeto activo que estamos creando.

Esta función es una especie de constructor que se usa en *C++* para definir un objeto. Esto es porque en este sentido los objetos de *JavaScript* son similares a las estructuras de *C* y a las clases de *C++*. En los 3 casos a los miembros, funciones miembro o propiedades del objeto/clase/estructura se accede con el operador punto (.):

```
estructura.miembro
clase.función_miembro()
objeto.propiedad.
```

Hay varias cosas a ver en este ejemplo. El nombre de la función es el nombre del objeto: casa (en *C++* el constructor tiene el nombre de la clase). La función no tiene valor de retorno.

El ejemplo muestra cómo se define un objeto casa, pero no crea ningún objeto específico del tipo casa.

Ejemplo 24:

```
function casa( habs, estil, fecha, garage){
    this.habitaciones = habs
    this.estilo = estil
    this.fecha_construcción = fecha
    this.tiene_garage = garage
}
```

Un objeto específico de casa tendrá las 4 propiedades con sus valores. Las instancias se crean usando la sentencia **new** con una función de llamada. Se crearía un objeto de casa, llamado micasa del siguiente modo:

Ejemplo 25:

```
var micasa = new casa(10,"Colonial", 1989, verdadero)
```

Ahora el objeto micasa es otra variable. Tiene que declararse usando var. Ahora que micasa ha sido creada podemos referirnos a sus propiedades con el operador punto (.):

Ejemplo 26:

```
micasa.habitaciones = 10 (entero, int)
micasa.estilo = "colonial" (cadena de caracteres, String)
micasa.fecha_construcción = 1989 (entero ,int)
micasa.tiene_garage = true (booleano)
```

No hay nada que evite poner "yes" en la propiedad tiene_garage en vez de un valor booleano, por esto hay que tener cuidado con este tipo de confusión. Si se pone "yes", tiene_garage no será booleana sino una cadena de caracteres.

11.4.1. Funciones (métodos)

Uno de los aspectos más potentes de la programación orientada a objetos de *JavaScript* es la posibilidad de crear clases con funciones miembro, llamadas métodos. Esto tiene varias ventajas como la organización y la asociación de funciones con clases. Los métodos, aunque programando se trabaje como si fueran propiedades no se tienen en cuenta a la hora de contarlos como tales.

Por ejemplo, tenemos una función que muestra las propiedades de los objetos de la clase casa llamada muestra_props(). Para añadirla como propiedad (se llamará muestra) a un objeto o a su clase se debería escribir:

Ejemplo 27:

```
this.muestra = muestra_props dentro de la definición de la clase casa.
Micasa.muestra = muestra_props como una sentencia normal.
```

y para usarlas, con los objetos de la clase casa:

```
micasa.muestra( ) o bien muestra_props( micasa )
```

11.4.2. Objetos como Arrays (Vectores)

Algunos lenguajes de programación soportan datos de tipo array (*C*, *C++*, Visual Basic, Java, etc.). Un array es una colección de ítems con índices los cuales son del mismo tipo. En *C* por ejemplo para declarar un array de 10 datos de tipo entero, se hace de la forma: int nombre[10]; y estos enteros son definidos desde el nombre[0] al nombre[9]. Es más común que la base del primer índice sea un **0** (zero-based indexing) que un 1 (one-based indexing).

JavaScript usa 0 como base del primer índice. En *JavaScript*, quizá, arrays y objetos son 2 puntos de vista del mismo concepto. Cada objeto es un array de los valores de sus propiedades, y cada array es también un objeto. Volviendo al ejemplo anterior, el objeto micasa es un array con los siguientes cuatro elementos:

Ejemplo 28:

```
micasa[0]=10 (habitación)
micasa[1]="colonial" (estilo)
micasa[2]=1989 (fecha_construcción)
micasa[4]=True (tiene garaje)
```

No parece haber muchas ventajas al referirse a objetos de este modo más numérico y menos informático. Quizá, esta forma alternativa permite acceder a las propiedades secuencialmente (p. ej con un bucle) lo que es muy usado.

Es aconsejable definir todos las clases con una propiedad que dé el número de propiedades en el objeto y haciéndola la primera propiedad. Ahora el objeto micasa es un array de 5 elementos. La nueva propiedad se denomina length (longitud).

Ejemplo 29:

```
micasa.length = 5
micasa[0]=10 (habitación)
micasa[1]="colonial" (estilo)
micasa.habitaciones = micasa[2]=1989 (fecha_construcción)
micasa.estilo=micasa[3] = True (tiene garaje)
micasa.tiene_garage = micasa[4] = True (tiene garaje)
```

Aun hay otro modo de dar valor a las propiedades:

Ejemplo 30:

```
micasa["length"] = 5
micasa["habitaciones"] = 10
micasa["estilo"] = "colonial"
micasa["fecha_construcción"] = 1989
micasa["tiene_garaje"] = true
```

11.4.3. Extender Objetos

Qué pasa si queremos más propiedades en un objeto: nada. Es posible extender dinámicamente un objeto simplemente tomando una nueva propiedad. Con el Ejemplo 31 se verá mejor:

Ejemplo 31:

```
tucasa = new casa(26, "restaurante",1993, false)
tucasa.paredes = 6
tucasa.tiene_terrazas = true
```

Estas dos sentencias añaden dos propiedades al final del array tucasa. Las extensiones dinámicas se aplican sólo a objetos específicos. El objeto micasa no se ve afectado ni la clase casa se ve afectada ni el objeto casa cambia de ningún modo.

Esta característica que se puede aplicar a los objetos simplifica mucho la programación. Para asegurarse de que no se producen errores al intentar mostrar las propiedades de un objeto es importante cambiar la propiedad que almacena el número de propiedades.

Ejemplo 32:

```
tucasa.length += 2
```

Un caso común donde la extensión dinámica es muy usada es en arrays de número variable.

11.4.4. Funciones con un número variable de argumentos.

Todas las funciones de JavaScript tienen las siguientes 2 propiedades: **caller** y **arguments**.

La propiedad **caller** es el nombre de cada uno que llama a la función. La propiedad **arguments** es un array de todos los argumentos que no están en la lista de argumentos de la función. La propiedad **caller** permite a una función identificar y responder al entorno desde donde se llama. La propiedad **arguments** permite escribir funciones que toman un número de argumento variable. Los argumentos de la lista de argumentos de una función son obligatorios mientras que los que están en la propiedad **arguments** son opcionales.

El siguiente ejemplo muestra los argumentos obligatorios yopcionales de una función.

Ejemplo 33:

```
function anadir( string){
    var nargumentos = anadir.arguments.length
    var totalstring
    var parcialstring =
        for (var i = 1; i < nargumentos; i++ ){
            parcialstring += " " + anadir.arguments[i] ","
        }
    totalstring = "El argumento obligatorio es " + string + ". El número de
    argumentos opcionales es " + nargumentos + " y los argumentos opcionales
    son " + parcialstring
    return (totalstring)
}
```

11.5. Expresiones y operadores de JavaScript.

11.5.1. Expresiones

Una expresión es cualquier conjunto de letras, variables y operadores que evalúa un valor simple. El valor puede ser un número, una cadena, o un valor lógico. Hay dos tipos de expresiones:

- aquellas que asignan un valor a una variable, `x = 7`
- aquellas que simplemente tienen un valor, `3 + 4`
- *JavaScript* tiene los siguientes tipos de expresiones:
- Aritméticas: evalúan un número.
- De cadena: evalúan un string de carácter, por ejemplo "Fred" or "234".
- Lógicas: evalúan si son verdadero o falso.

11.5.1.1. Expresiones Condicionales

Una expresión condicional puede tomar uno de dos posibles valores según una condición. La sintaxis es `(condición) ? valor1 : valor2`.

Si la condición es verdadera (*true*) toma el valor `valor1`, y si es falsa (*false*) toma el `valor2`. Se puede usar una expresión condicional en cualquier parte.

Por ejemplo: `estado = (edad >= 18) ? "adulto" : "menor de edad".` Si edad es mayor o igual que 18 a la variable estado se le asigna la cadena "adulto", si no se le asigna el valor "menor de edad".

11.5.2. Operadores de asignación (`=, +=, -=, *=, /=`)

Un operador de asignación da un valor a la izquierda de su operando basado en la parte derecha de su operando. El operador básico de asignación es el igual (`=`), el cual asigna el valor de la derecha de su operando al de la izquierda de su operando. `x = y`, asigna el valor de `y` a `x`.

Los otros operadores de asignación para operaciones aritméticas son los siguientes:

- `X = y`
- `x += y` significa `x = x + y`
- `x -= y` significa `x = x - y`
- `x *= y` significa `x = x * y`
- `x /= y` significa `x = x / y`
- `x %= y` significa `x = x % y`

Estas sentencias primero operan a la derecha del operador y después devuelven el valor obtenido a la variable situada a la izquierda del operador.

11.5.3. Operadores aritméticos

Los operadores toman valores numéricos (sean letras o variables) y devuelven un único valor numérico.

- Los operadores estándar son la suma (`+`), la resta (`-`), la multiplicación (`*`), y la división (`/`). Estos operadores trabajan de la forma estándar *operando1 operador operando2*.

Además hay otros operadores no tan conocidos como:

- Resto (`%`) El operador resto devuelve el resto entero al dividir el primer operando entre el segundo operando. Sintaxis: `var1 % var2`.

Ejemplo 34:

`12 % 5` returns 2.

- Incremento (`++`) El operador incremento se usa de la siguiente manera: `var++` o `++var`. Este operador suma uno a su operando y devuelve el valor.

1. `var++`, primero devuelve el valor de la variable y después le suma uno.
2. `++var`, primero suma uno y después devuelve el valor de la variable.

Por ejemplo si `x` es 3, la sentencia `y = x++`, incrementa `x` a 4 y asigna 3 a `y`, pero si la sentencia es `y = ++x`, incrementa `x` a 4 y asigna 4 a `y`.

- Decremento (`--`) El operador decremento se usa de la siguiente manera: `var--` o `--var`. Este operador resta uno a su operando y devuelve el valor.

1. `var--` primero devuelve el valor de la variable y después le resta uno.
2. `--var`, primero suma uno y después devuelve el valor de la variable.

Por ejemplo si `x` es 3, la sentencia `y = x--`, decremente `x` a 2 y asigna 3 a `y`, pero si la sentencia es `y = --x`, decremente `x` a 2 y asigna 2 a `y`.

- Negación (`-`) El operador negación precede a su operando. Devuelve su operando negado. Por ejemplo, `x = -x`, hace negativo el valor de `x`, que si fuera 3, sería -3.

11.5.4. Operadores lógicos

Los operadores lógicos toman valores lógicos (booleanos) como operandos. Devuelven un valor lógico. Los valores lógicos son `true` (verdadero) y `false` (falso). Se suelen usar en las sentencias de control.

- And (`&&`) Uso: `expr1 && expr2`. Este operador devuelve `true` si ambas expresiones lógicas son verdaderas, o `false` si alguna no es `true`.

- Or (||) Uso: expr1 || expr2. Este operador devuelve *true* si una de las dos expresiones lógicas, o ambas, son verdaderas, o *false* si ambas son falsas.
- Not (!) Uso: !expr. Este operador niega su expresión. Devuelve *true* si es *false* y *false* si es *true*.

Ejemplo 35:

```
if ((edad_pepe>=18)&&(edad_juan==18)){
    document.write("Juan y pepe son adultos")
} else {
    document.write("Uno de los dos no es adulto")
}
```

11.5.5. Operadores de Comparación (=, >, >=, <, <=, !=)

Un operador de comparación compara sus operandos y devuelve un valor lógico según sea la comparación verdadera o no. Los operandos pueden ser números o cadenas de caracteres. Cuando se usan cadenas de caracteres, las comparaciones se basan en el orden alfabético. Al igual que los operadores lógicos, se suelen usar en sentencias de control.

Los operadores son:

- Igual (==), devuelve *true* si los operandos son iguales.
- Desigual (!=), devuelve *true* si los operandos son diferentes.
- Mayor que (>), devuelve *true* si su operando izquierdo es mayor que el derecho.
- Mayor o igual que (>=), devuelve *true* si su operando izquierdo es mayor o igual que el derecho.
- Menor que (<), devuelve *true* si su operando izquierdo es menor que el derecho.
- Menor o igual que (<=), devuelve *true* si su operando izquierdo es menor o igual que el derecho.

11.5.6. Operadores de String

Además de los operadores de comparación, que pueden usarse con cadenas de caracteres, existe el operador concatenación (+) que une dos cadenas, devolviendo otra cadena que es la unión de las dos anteriores.

Ejemplo 36:

```
"mi " + "casa" devuelve la cadena "mi casa".
```

El operador de asignación += se puede usar para concatenar cadenas. Por ejemplo, si la variable letra es una cadena con el valor "alfa", entonces la expresión letra += "beto" evalúa a "alfabeto" y asigna este valor a letra.

11.5.7. Prioridad de los operadores

La prioridad de los operadores determina el orden con el cual se aplican cuando se evalúan. Esta prioridad se rompe cuando se usan paréntesis.

La prioridad de operadores, de menor a mayor es la que sigue:

- coma ,
- asignación = += -= *= /= %=
- condicional ?:
- lógico-or ||
- logical-and &&
- igualdad == !=
- relación < <= > >=
- adición/sustracción + -
- multiplicación / división / resto * / %

- negación/incremento ! ~ - ++ --
- paréntesis, corchetes () [] .

11.6. Sentencias de control de JavaScript.

JavaScript soporta un conjunto de sentencias que se pueden usar para hacer interactivas las páginas Web.

11.6.1. La sentencia if

Una sentencia if es como un interruptor. Si la condición especificada es cierta, se ejecutan ciertas sentencias. Si la condición es falsa, se pueden ejecutar otras. Un sentencia if es :

```
if (condición) {  
    sentencias 1 }  
[else {  
    sentencias 2}]
```

La parte else es opcional.

Ejemplo 37:

```
if ((edad_pepe>=18)&&(edad_juan==18)){  
    document.write("Juan y pepe son adultos")  
} else {  
    docuemnt.write("Uno de los dos no es adulto")  
}
```

11.6.2. Bucles

Un bucle es un conjunto de comandos que se ejecutan repetidamente hasta que una condición especificada se encuentra. *JavaScript* proporciona dos tipos de bucles: for y while.

11.6.2.1. Bucle for

Una sentencia **for** repite un bucle hasta que una condición se evalúe como **false**. Este bucle es similar al tradicional bucle for en Java, C y C++.

Un bucle **for** es:

```
for ([expresión_inicial]; [condición] ;[expresión_incremento]) {  
    sentencias  
}
```

Ejemplo 38:

```
for (var contador = 0; contador <= 5; contador++) {  
    document.write("Número " + contador + "<br>")  
}
```

Y la salida por pantalla del ejemplo es:

Número 0
Número 1
Número 2
Número 3
Número 4
Número 5

Cuando se encuentra un bucle **for**, se ejecuta la expresión inicial. Las sentencias se ejecutan mientras la condición sea **true**. La expresión_incremto se ejecuta cada vez que vuelve a realizarse una vuelta o paso en el bucle.

11.6.2.2. Bucle while

Una sentencia **while** repite un bucle mientras la condición evaluada sea **true**. Un bucle **while** es:

```
while (condición)
{
    sentencias
}
```

Ejemplo 39:

```
var contador = 0
while (contador <= 5){
    document.write("Número " + contador + "<br>")
    contador++
}
```

Es el mismo ejemplo que el 38 pero con el bucle while.

Si la condición llega a ser false, las sentencias dentro del bucle dejan de ejecutarse y el control pasa a la siguiente sentencia después del bucle.

La condición se evalúa cuando las sentencias en el bucle han sido ejecutadas y el bucle está a punto de ser repetido. Dentro del bucle debe haber una sentencia que en algún momento haga parar la ejecución del bucle.

La comprobación de la condición tiene lugar únicamente cuando las sentencias del bucle se han ejecutado y el bucle está a punto de volverse a ejecutar. Esto es, la comprobación de la condición no es continuada, sino que tiene lugar por primera vez al principio del bucle y de nuevo a continuación de la última sentencia del bucle, cada vez que el bucle llega a este punto.

Ejemplo 40:

```
n = 0; x = 0
while( n < 3 ) {
    n++; x += n;
}
```

11.7. Comentarios

Los comentarios son anotaciones del autor para explicar qué hace cada sentencia. Son ignorados por el navegador. *JavaScript* soporta el estilo de comentarios de *C*, *C++* y *Java*.

- Los comentarios de una sola línea son precedidos por una doble barra normal (//).
- Los comentarios de más de una línea van escritos entre /* y */.

Ejemplo 41:

```
// Esto es un comentario de una sola línea.

/* Esto es un comentario de más de una línea. Puede
tener la extensión que se quiera. */
```

12. Graphical User Interface

12.1. Graphical User Interface (GUI)

Las interfaces gráficas de usuario (**GUI**) se componen de un conjunto de partes cada vez más usadas, como los botones para respuestas del usuario, regiones para despliegue y escritura de texto, menús descendentes y así sucesivamente. Estas partes se denominan **widgets**.

La simple adición de widgets a un programa permite tener escaso control del aspecto de la interfaz, lo que se ve bien en un ambiente podría ser del todo inaceptable en otro. Se hace necesario un análisis del diseño visual. Veremos dos características que hacen posible lograr el aspecto que se pretende de la **GUI** sin importar el sistema en que se ejecute el programa: los **Container** y los **LayoutManager**.

Además de un diseño visual correcto, las aplicaciones deben ser capaces de prestar atención a la actividad del teclado, los movimientos y clics del ratón, etc., en definitiva, ser capaces de interactuar con el usuario. Este tipo de aplicaciones se dice que están controladas por eventos. **Java** incluye un amplio conjunto de funciones para vigilar y notificar eventos.

12.1.1. Componentes gráficos: Abstract Window Toolkit (AWT)

El paquete `java.awt` constituye un conjunto de clases con independencia de plataforma que permite usar componentes gráficos muy diversos. Al utilizar las clases de **AWT**, es posible diseñar un programa útil y visualmente eficaz sin preocuparse por detalles de bajo nivel relacionados con los objetos gráficos.

12.1.1.1. Widgets o componentes elementales

En la jerga computacional se conoce como **widgets** a los componentes elementales que forman las interfaces gráficas de usuario (GUI). La clase `Component` es, sin duda, la más importante del paquete `java.awt` (Figura 12.1) y es la superclase de todas las clases de **widget**, salvo los menus, además de servir como depósito de todos los métodos comunes a **widget**.

Tenemos **componentes de texto** como la clase `Label`, y las clases `TextField` y `TextArea` y por otro lado **componentes activos** como las clases `Button`, `CheckBox`, `CheckboxGroup`, `Choice` y `List`.

12.1.1.2. Métodos de organización: Contenedores

Las subclases de `Container` posibilitan pensar en el diseño visual con base en una organización jerárquica, mientras que las clases de diseño permiten especificar el aspecto que la jerarquía de los `Container` y sus `Component` tienen para el usuario.

12.1.1.3. Diseño Visual: Layouts

Cada `Container` tiene su propio “diseño visual” o `LayoutManager`, interfaz que se encarga de colocar los componentes en un objeto `Container`. Para acceder al diseño y modificarlo se dispone de los siguientes métodos:

`LayoutManager getLayout()`

Devuelve el `LayoutManager` activo de este `Container`.

`void setLayout(LayoutManager layout)`

Este método permite cambiar el `LayoutManager` de este `Container`.

Se dispone de tres clases de diseño o `LayoutManager` que se analizan en este apartado y sólo difieren en la forma de posicionar los componentes. Estas son las clases `FlowLayout`, `BorderLayout` y `GridLayout`.

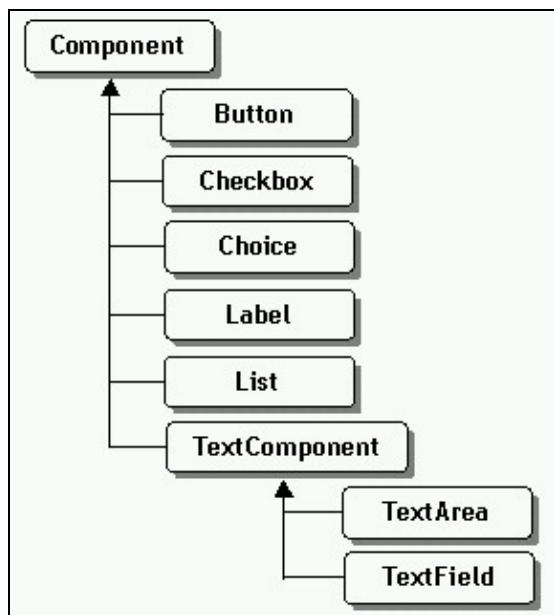


Figura 12.1. Una porción de la jerarquía de clases de Component

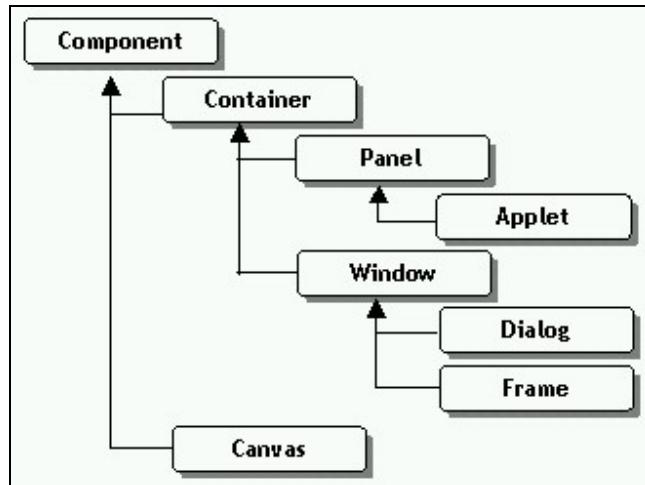


Figura 12.2. Una porción de la jerarquía de clases de Container

Label	TextField
aligned left aligned center aligned right	Enter your name: <input type="text" value="your name here"/> Enter your phone number: <input type="text"/>
TextArea	Button
Un objeto TextArea se puede usar cuando se requieren dos o más líneas para la entrada o salida de datos. 	Rewind <input type="button"/> Play <input type="button"/> Fast Forward <input type="button"/> Stop
CheckBox	CheckboxGroup
<input checked="" type="checkbox"/> Shoes <input type="checkbox"/> Socks <input type="checkbox"/> Pants <input checked="" type="checkbox"/> Underwear <input checked="" type="checkbox"/> Shirt	<input type="radio"/> Red <input type="radio"/> Blue <input type="radio"/> Yellow <input checked="" type="radio"/> Green <input type="radio"/> Orange <input type="radio"/> Purple
Choice	List
	Coca-Cola Cerveza Agua Café

Figura 12.3. Aspecto de los componentes elementales

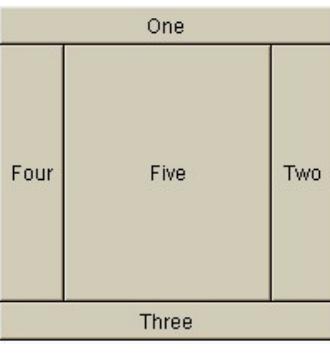
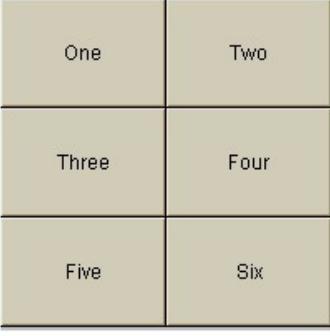
	<pre>import java.awt.*; import java.applet.*; public class BorderLayoutTest extends Applet { public void init() { setLayout(new BorderLayout()); add(BorderLayout.NORTH, new Button("One")); add(BorderLayout.EAST, new Button("Two")); add(BorderLayout.SOUTH, new Button("Three")); add(BorderLayout.WEST, new Button("Four")); add(BorderLayout.CENTER, new Button("Five")); } }</pre>
	<pre>import java.awt.*; import java.applet.*; public class GridLayoutTest extends Applet { public void init() { setLayout(new GridLayout(3,2)); add(new Button("One")); add(new Button("Two")); add(new Button("Three")); add(new Button("Four")); add(new Button("Five")); add(new Button("Six")); } }</pre>

Figura 12.4. Aspecto de los diferentes LayoutManager.

12.1.2. Eventos

Casi todos los lenguajes de programación evolucionan con el paso del tiempo, y Java no es la excepción. En la primera revisión importante de Java, la versión 1.1, se incluyó un grupo de nuevos métodos para las clases de la **AWT**. También se añadieron algunos nuevos paquetes y clases; pero gran parte de los cambios fue más o menos superficial, excepto lo relativo al manejo de eventos. Es en estos donde se advierten las diferencias más significativas entre las versiones 1.0 y subsiguientes de Java (Java 1.1 y sus modificaciones y Java 1.2, ahora llamada Java 2). Por fortuna (al menos, para los autores de libros), ya no se tiene soporte para el modelo de eventos de la versión 1.0, por lo que los autores pueden dedicar el tiempo a escribir acerca de una sola forma de manejar los eventos, con la confianza de que este modelo será válido al menos durante la vida útil de su obra, sin importar que se use el ambiente Java 2 ó cualquiera de las versiones 1.1.x.

Un programa escrito en Java maneja los eventos mediante lo que se conoce como **modelo de delegación**. En éste, existen dos actores, el objeto Component (un botón, por ejemplo) que genera los

eventos y otro objeto, que podrá ser un subprograma o una instancia de otra clase, el cual contiene el código para el manejo de los eventos. Las principales características de este modelo son:

- Todo componente puede ser la *fuente* de un evento.
- Toda clase puede ser un *escucha* (*listener*) de un evento, para lo cual basta la *instrumentación de la interface de escucha* apropiada.
- El evento que genera un Component se envía sólo a los escuchas *registrados* con el objeto fuente.

Para poder trabajar con eventos se hace necesario importar un nuevo paquete, `java.awt.event`. Los diversos tipos de eventos en Java están organizados en una jerarquía de clases que se ilustra en la Figura 12.7.

Vamos a desarrollar un ejemplo que muestra la sencillez de este modelo. La aplicación contiene dos botones, en los que se puede hacer clic para mover el punto rojo hacia la izquierda o la derecha. En la Figura 12.5 se ilustra el aspecto que tiene la aplicación en pantalla.



Figura 12.5. Aplicación Eventos.class

A continuación se muestra el código en Java de la aplicación. Se han resaltado en negrita las partes del código que se añaden para poder manejar los eventos. Estas sentencias y métodos realizan diferentes tareas: incluir el paquete de clases de eventos, establecer vínculos entre fuentes y escuchas e instrumentar el interfaz del escucha para manejar el evento.

```
/* Programa que muestra el funcionamiento de Eventos en Java */

import java.awt.*;
import java.awt.event.*;      //Paquete para las clases de eventos

public class Eventos extends Frame{

    private Button    left,right;
    private Display   myDisplay;

    //Constructor
    public Eventos(){
        super("Aplicación Ejemplo de Eventos");
        setSize(400,200);
        setLayout(new BorderLayout());
        myDisplay = new Display();
        add(BorderLayout.CENTER, myDisplay);

        Panel p = new Panel();
        left = new Button("Izquierda");
        p.add(left);
        right = new Button("Derecha");
        p.add(right);
        add(BorderLayout.SOUTH, p);
    }
}
```

```

//Registrar myDisplay con cada botón
left.addActionListener(myDisplay);
right.addActionListener(myDisplay);
}

public static void main(String args[]){
    Eventos a = new Eventos();
    a.show();
}
}

class Display extends Canvas implements ActionListener{
    private Point center;

    //Constructor
    public Display(){
        center = new Point(50,50);
        setBackground(Color.white);
    }

    //Método que se llama cuando se produce un evento
    public void actionPerformed(ActionEvent e){
        //Obtener el rótulo del botón que generó el evento
        String rotulo = e.getActionCommand();
        //Movemos el punto según qué botón haya generado el evento
        if(rotulo.equals("Izquierda")){
            center.x -= 12;
        }else if (rotulo.equals("Derecha")){
            center.x += 12;
        }
        //Se fuerza una llamada a paint()
        repaint();
    }

    //Dibujar el punto rojo
    public void paint(Graphics g){
        g.setColor(Color.red);
        g.fillOval(center.x - 5, center.y - 5, 10, 10);
    }
}
}

```

El manejo de eventos generados por un componente *fuente* en un programa requiere:

- Una declaración de la forma: `import java.awt.event.*;`
- Un objeto *escucha* que instrumente todos los métodos de una interface escucha apropiada.
- Un vínculo entre los objetos *fuente* y *escucha*, que se establece mediante una llamada a un método de Component, con la forma: `fuente.addListener(escucha);`, donde xxx indica el tipo de evento que se manejará.
- Código en el método *escucha* apropiado para manejar el evento.

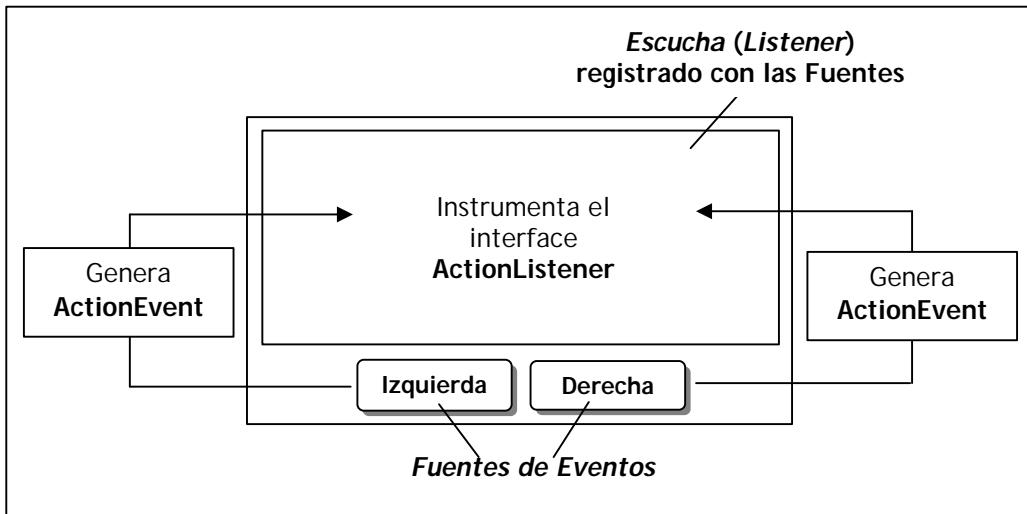


Figura 12.6. Fuentes y escuchas de Eventos en el ejemplo.

Los eventos más comunes que se manejan en aplicaciones Java son los siguientes:

- **Eventos de acción (clase ActionEvent):** Un ActionEvent se genera cuando el usuario hace clic en un botón, hace doble clic en un elemento de una lista, selecciona un MenuItem o presiona <Intro> en un TextField.
- **Eventos de elementos (clase ItemEvent):** Se genera una instancia de ItemEvent cuando el usuario hace clic en un CheckBox, CheckBoxMenuItem o Choice.
- **Eventos del teclado (clase KeyEvent):** Los eventos del teclado se generan cuando el usuario pulsa o suelta una tecla. En realidad existen tres tipos de KeyEvent: pulsar una tecla, soltar una tecla y escribir con una tecla, que es una secuencia de las dos primeras.
- **Eventos del ratón (clase MouseEvent):** Cuando se hace clic con el ratón, se le mueve o se realiza cualquier otra acción con él, el Component donde está el puntero en el momento de la acción genera un MouseEvent.
- **Eventos de texto (clase TextEvent):** Los TextEvent se generan cuando el usuario modifica el contenido de un objeto TextArea o TextField, o si cambia su contenido con un método como setText().

- **Eventos de ventana:** Son las diversas acciones que un usuario o programa pueden emprender con una ventana. Una ventana puede volverse activa o inactiva, al colocarse delante o detrás de otra. En muchos sistemas, es posible “iconificar” ventanas, al minimizarlas hasta una pequeña representación en un ícono, o “desiconificarlas”, mediante su expansión desde su estado de ícono. Se generan eventos a su vez cuando se abre o se cierra la ventana.

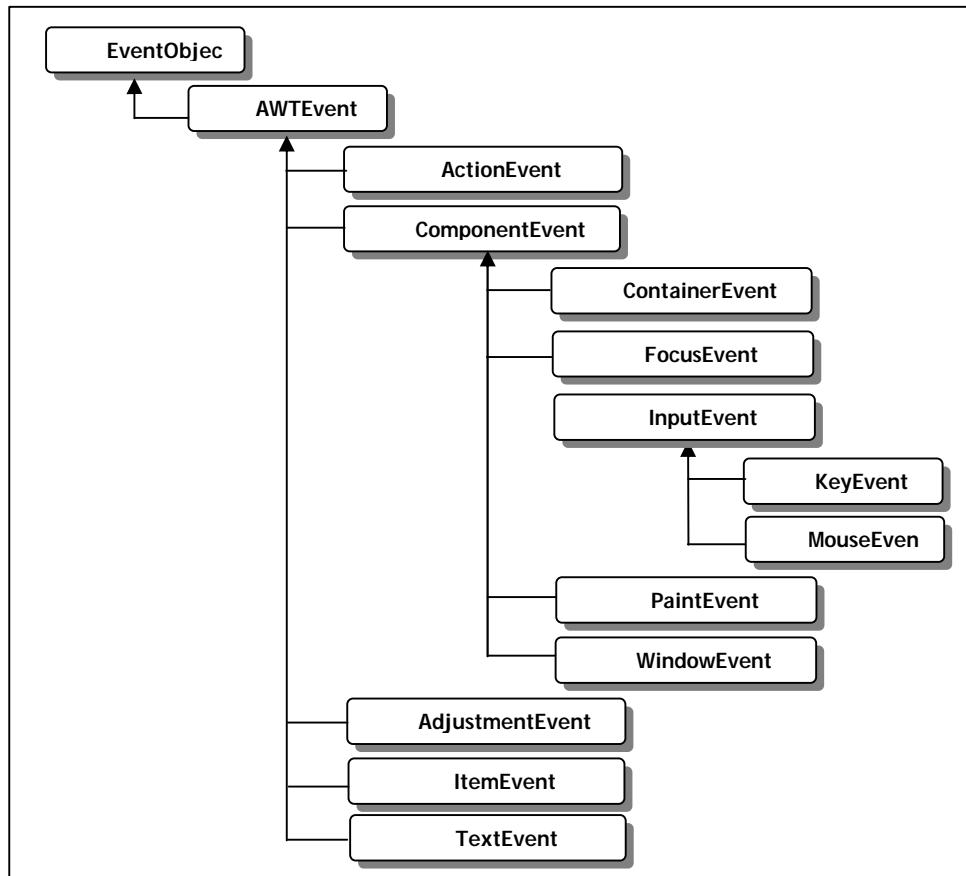


Figura 12.7. Jerarquía de clases de AWTEvent

En el modelo de eventos de Java, los Component generan eventos cuyo manejo corresponde a cualquier objeto escucha registrado con la fuente del evento. Un escucha es toda clase que instrumenta la interfaz apropiada para el evento. Luego, a fin de diseñar un manejador de eventos, se construye una clase (o se utiliza una ya definida que cuente con toda la información necesaria para responder al evento concreto) que instrumente el escucha apropiado del evento y se realiza una sobrecarga de **todos** los métodos de la interfaz. Por ello, es necesario conocer las interfaces escucha y sus métodos. Existen 11 interfaces escucha, que se listan a continuación:

ActionListener	// para ActionEvent
AdjustmentListener	// para Scrollbar
ComponentListener	// para ComponentEvent
ContainerListener	// para ContainerComponent
FocusListener	// para rastrear eventos de enfoque
ItemListener	// para ItemEvent
KeyListener	// para KeyEvent
MouseListener	// para todos los eventos del ratón excepto
MouseMotionListener	// MOUSE_DRAG y MOUSE_MOVE
TextListener	// para TextEvent
WindowListener	// para WindowEvent

12.1.3. Applets

Como se puede observar en la Figura 12.2 los **applets** son contenedores, herederos de la clase Panel. Permiten realizar aplicaciones a las que se accede mediante un visualizador de páginas web (o navegador), pues se transmiten a través de la red y los ejecuta la máquina virtual del visualizador. En castellano se denominan **subprogramas**.

Los applets no tienen método `main()` y su ciclo de vida se basa en la ejecución de los siguientes métodos:

- `void init()`: es ejecutado por el navegador la primera vez que se carga el applet. Se suele colocar en este método el código que se pondría en el constructor si se tratase de una aplicación independiente, es decir, todo lo necesario para su inicialización.
- `void start()`: se ejecuta cada vez que se carga el applet.
- `void stop()`: se ejecuta cuando se abandona el documento que contiene el applet.
- Otros métodos importantes son:
 - `void paint(Graphics g)`: dibuja el applet; `g` indica la zona que se va a dibujar.⁶
 - `void update(Graphics g)`: actualiza el applet, rellenándolo primero con el color de `g` e invocando después al método `paint`.
 - `void repaint()`: este método llama al método `update()`. Éste es el método que debe invocar el programador si quiere repintar el applet.
 - `void resize(int ancho, int alto)`: cambia el tamaño al indicado por ancho y alto.

Veamos a continuación un ejemplo sencillo de un applet que muestra un mensaje:

```
/* Applet que muestra un mensaje */
import java.applet.Applet;
import java.awt.Graphics;

public class MiApplet extends Applet{
    public void paint (Graphics g){
        g.drawString ("Mi primer Applet?", 10, 30);
    }
}
```

Guardamos el código en un fichero con nombre ***MiApplet.java*** y lo compilamos de la misma forma que vimos para aplicaciones independientes. Sin embargo, para visualizar el applet en un navegador (puesto que no es una aplicación independiente) es necesario disponer de una página **HTML** que lo despliegue. El contenido mínimo de esta página será:

```
<HTML>
<BODY>
    <APPLET code=?MiApplet.class? width=60 height=60></APPLET>
</BODY>
</HTML>
```

Guardamos el código **HTML** en un fichero con extensión ***.html***, por ejemplo ***MiApplet.html***. Ahora se utiliza cualquier navegador para abrir la página html. El fichero ***.class*** del applet debe estar en el mismo directorio que el ***.html***. La Figura 12.8 muestra el proceso completo de despliegue de un applet en una página Web.

⁶ La clase `Graphics` mantiene un contexto gráfico: indica la zona en que se va a dibujar, color de dibujo del background y del foreground, font, etc...

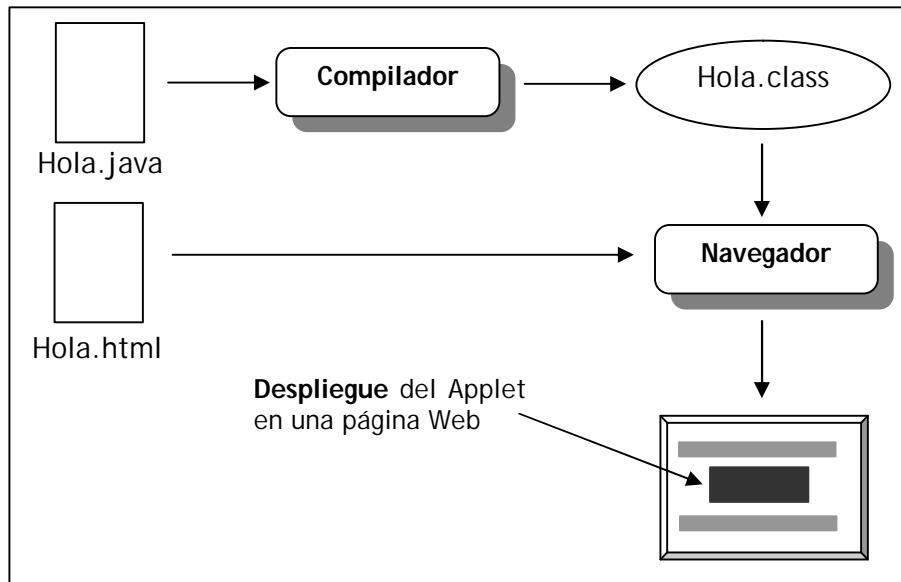


Figura 12.8. Despliegue de Applets.

El JDK pone a disposición una herramienta para visualizar applets sin necesidad de un navegador comercial. Esta herramienta es el **appletviewer.exe** y se utiliza con la siguiente sentencia desde una consola de MS-DOS y desde el directorio donde se encuentre el ***.html** y el ***.class**:

```
Appletviewer MiApplet.html
```

La salida gráfica será la siguiente:

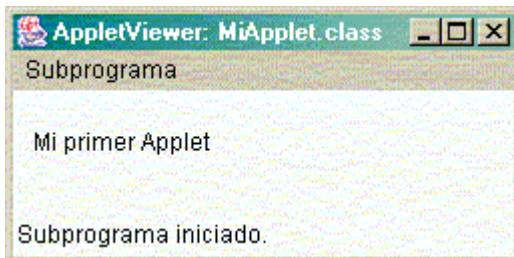


Figura 12.9. MiApplet.class en el AppletViewer.

12.2. Otros elementos de Java

12.2.1. Manejo de Excepciones y Errores

Java incorpora en el propio lenguaje la gestión de errores. El mejor momento para detectar los errores es durante la compilación. Sin embargo prácticamente sólo los errores de sintaxis son detectados durante este periodo. El resto de problemas surgen durante la ejecución de los programas.

En el lenguaje **Java**, una **Exception** es un cierto tipo de error o una condición anormal que se ha producido durante la ejecución de un programa. Algunas **excepciones** son fatales y provocan que se deba finalizar la ejecución del programa. En este caso conviene terminar ordenadamente y dar un mensaje explicando el tipo de error que se ha producido. Otras, como por ejemplo no encontrar un fichero en el que hay que leer o escribir algo, pueden ser recuperables. En este caso el programa debe dar al usuario la oportunidad de corregir el error (indicando una nueva localización del fichero no encontrado).

Un buen programa debe gestionar correctamente todas o la mayor parte de los errores que se pueden producir. Hay dos "estilos" de hacer esto:

1. **A la "antigua usanza"**: los métodos devuelven un código de error. Este código se chequea en el entorno que ha llamado al método con una serie de **if elseif ...**, gestionando de forma diferente el resultado correcto o cada uno de los posibles errores. Este sistema resulta muy complicado cuando hay varios niveles de llamadas a los métodos.

2. **Con soporte en el propio lenguaje:** En este caso el propio lenguaje proporciona construcciones especiales para gestionar los errores o **Exceptions**. Suele ser lo habitual en lenguajes modernos, como C++, Visual Basic y **Java**.

Los errores se representan mediante dos tipos de clases derivadas de la clase **Throwable**: **Error** y

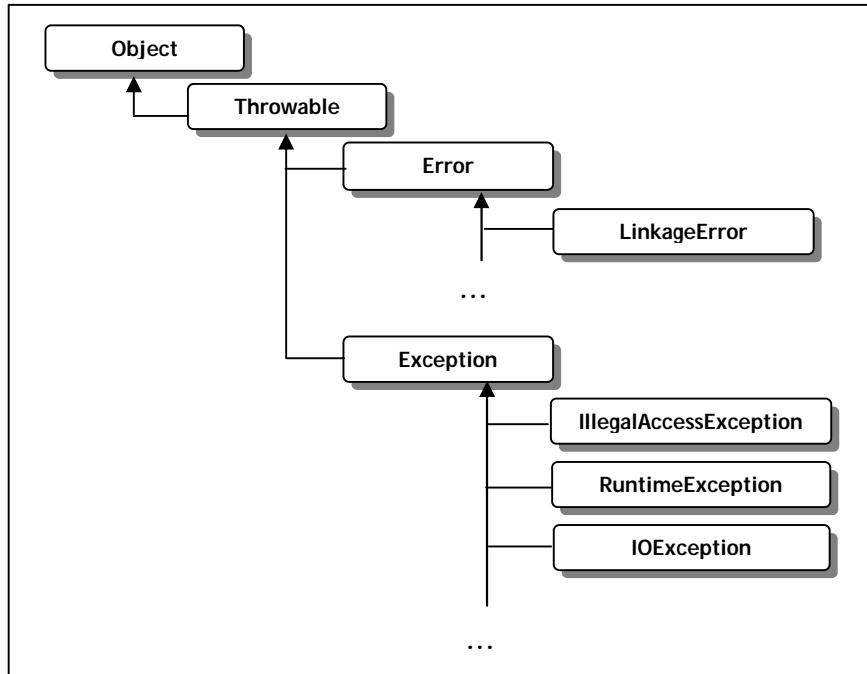


Figura 12.10. Una parte de la jerarquía de clases de **Throwable**.

La clase **Error** está relacionada con errores de compilación, del sistema o de la JVM. De ordinario estos errores son **irrecuperables** y no dependen del programador ni debe preocuparse de capturarlos y tratarlos. La clase **Exception** tiene más interés. Dentro de ella se puede distinguir:

1. **RuntimeException:** Son excepciones muy frecuentes, de ordinario relacionadas con errores de programación. Se pueden llamar **excepciones implícitas**.
2. Las demás clases derivadas de **Exception** son **excepciones explícitas**. **Java** obliga a tenerlas en cuenta y chequear si se producen.

Las clases derivadas de **Exception** pueden pertenecer a distintos packages de **Java**. Algunas pertenecen a **java.lang** (**Throwable**, **Exception**, **RuntimeException**, ...); otras a **java.io** (**EOFException**, **FileNotFoundException**, ...) o a otros packages.

12.2.2. Entrada/Salida de Datos

Todos los lenguajes de programación modernos poseen la característica de permitir que se guarde la salida de un programa en un archivo externo, que puede residir, por ejemplo, en un disco duro. Una vez que se guarda en el disco duro (o cualquier otro medio de almacenamiento), se puede leer posteriormente y usar como entrada para un programa, ya sea el mismo que generó el archivo, u otro programa totalmente distinto.

La entrada y salida de datos en Java se logran mediante las clases de **flujo de datos** **InputStream** y **OutputStream**, así como sus subclases, algunas de las cuales se ilustran en la Figura 12.11, junto con la clase **File**, que permite obtener información acerca de un archivo. Todas estas clases son parte del paquete **java.io**.

En **Java**, un flujo de datos es una secuencia ordenada de objetos y un conjunto de métodos que permiten extraer un objeto (leerlo del flujo) y añadir un nuevo objeto al flujo (escribirlo en el flujo).

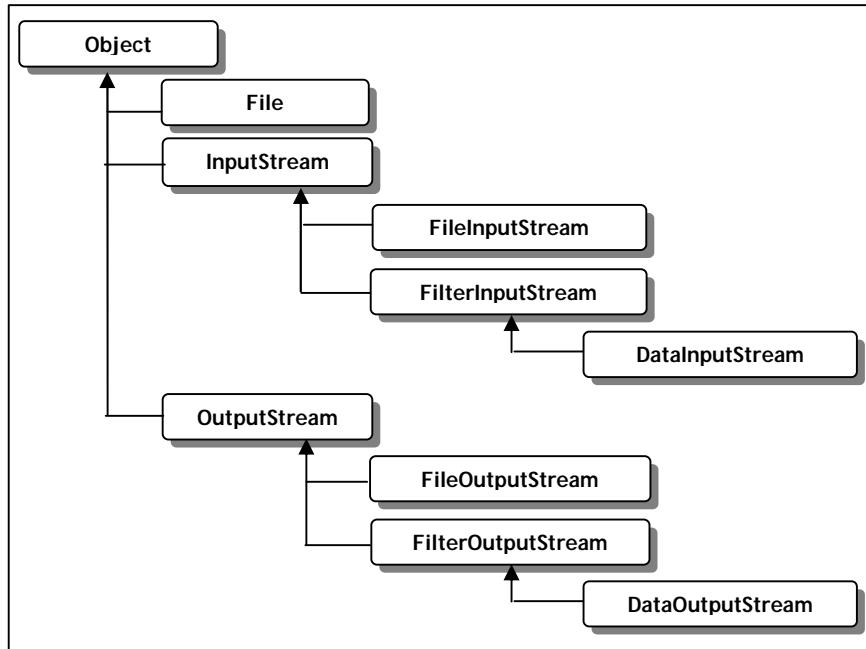


Figura 12.11. Una parte de la jerarquía de clases de flujo de datos.

12.2.3. Subprocesos

Generalmente se piensa en la ejecución de un programa como un proceso secuencial, de instrucción en instrucción, quizá realizando bucles o saltando a otra parte del código en respuesta a una llamada a un método. Sin embargo, es posible tener varios procesos secuenciales a la vez. En la práctica, los procesos secuenciales nunca se ejecutan simultáneamente, sino que uno ejecuta varias instrucciones y luego descansa, mientras otro ejecuta su código. Pero esta alternación ocurre tan rápidamente que puede pensarse que trabajan al mismo tiempo.

Así pues, un **subproceso** representa una secuencia de acción independiente en un programa. Java proporciona la clase `Thread`, parte del paquete `java.lang`, para el manejo de subprocesos.

Bibliografía

- Robert Sedgewick, Kevin Wayne. **Introduction to Programming in Java: An Interdisciplinary Approach.** Addison-Wesley (2007).
- Bruce Eckel. **Thinking in Java (3rdEdition)**. Prentice-Hall (2002)
- Camelia Muñoz Caro, Alfonso Niño Ramos, Aurora Vizcaíno. **Introducción a la Programación Con Orientación A Objetos.** Barceló. Prentice Hall (2002)
- Jasón Hunter, William Crawford. **Java Servlet Programming, Second Edition.** O'Reilly & Associates, Inc. (2001).
- Rick Decker, Stuart Hirshfield. **Programación con Java. Segunda edición.** Thomson. (2001).
- José M^a Pérez Menor, Jesús Carretero Pérez, Félix García Carballeira, José Manuel Pérez Lobato. **Problemas Resueltos de Programación en lenguaje Java.** THOMSON. (2002).
- Cay S. Horstmann & Gary Cornell. **Java 2. Volumen 1. Fundamentos.** Prentice Hall (2003).
- Cay S. Horstmann & Gary Cornell. **Java 2. Volumen 2. Características avanzadas.** Prentice Hall (2003).
- Neal Ford. **Art of Java Web Development.** Manning Publications Co. (2003)
- Steven Douglas Olson. **Ajax on Java.** O'Reilly (2007)