0. Conceptos previos.

1. La arquitectura Java.

Java no es sólo un lenguaje de programación, sino toda una arquitectura de desarrollo y ejecución de software que se estructura en varias capas:

a. La máquina virtual Java (JVM). Es la encargada de ejecutar las clases Java.
b. La API (Application Program Interface), que contiene las bibliotecas standard de programación.
c. Herramientas de programación, tales como el compilador Java, el depurador, etc.
d. El lenguaje Java propiamente dicho.

Las distribuciones de Java se pueden encontrar en dos modalidades:

e. JRE (Java Runtime Environment) o entorno de ejecución, compuesto por la máquina virtual y la API standard.
f. JDK (Java Development Kit) o kit de desarrollo Java. Se compone del JRE además de las herramientas de desarrollo.

El JRE contiene lo necesario para ejecutar aplicaciones Java y será lo que se instale en las máquinas cliente, mientras el JDK, además incluye las herramientas y utilidades necesarias para el desarrollo de aplicaciones. Además existen diversos entornos integrados de desarrollo (IDE, Integrated Development Environment), que se apoyan en el JDK instalado, o bien incluyen uno propio.

Es importante tener en cuenta que la arquitectura Java es una especificación de Sun Microsystems (actualmente Oracle Corporation), y aunque ésta misma empresa dispone de su propia implementación, también existen otras implementaciones de Java, como son la de GNU, Apache y otras.

2. Tecnologías Java.

Sun Microsystems (actualmente Oracle Corporation) ofrece especificaciones para varias tecnologías Java, todas ellas basadas en el mismo lenguaje de programación - el propio lenguaje Java - pero destinadas a distintos tipos de aplicaciones y plataformas hardware. A continuación hacemos un pequeño resumen de cada una de ellas:

- JSE (Java Standard Edition). Edición standard. Es la de uso más común, destinada principalmente a las aplicaciones de escritorio por medio de un amplio conjunto de bibliotecas standard, entre las que se incluyen utilidades, entornos gráficos de usuarios, etc.

- JEE (Java Enterprise Edition). Edición empresarial. Es un superconjunto de la anterior, y está orientada a las aplicaciones de servidor y distribuidas. Incluye bibliotecas especializadas en servicios web, modelos de componentes, comunicaciones, etc.

- JME (Java Micro Edition). Edición micro. Destinada a la programación de dispositivos con recursos limitados tales como PDAs, teléfonos móviles y otros dispositivos de consumo. Sus APIs incluyen bibliotecas para el desarrollo de interfaces de usuario muy flexibles, un modelo de seguridad muy robusto, y un amplio rango de protocolos de comunicaciones incorporados.

- Java Card. Edición de tarjeta. Proporciona un entorno seguro para ejecutar aplicaciones en smart-cards y otros dispositivos con capacidades muy limitadas de memoria y ejecución.

3. El JDK. Herramientas.

El JDK (Java Development Kit, o kit de desarrollo Java), proporciona un conjunto de herramientas standard para la compilación, depuración, documentación y ejecución de proyectos así como un conjunto de utilidades auxiliares. Éstas herramientas y utilidades se ofrecen como comandos de consola. A continuación comentamos brevemente algunas de las más importantes:

javac: es el compilador de lenguaje java.

java: lanzador de aplicaciones java. Inicia la ejecución de la máquina virtual.

javadoc: generador de documentación a partir del código fuente java.

jar: herramienta para la creación y manejo de archivos java (.jar).

jdb: depurador en línea de comandos.

javap: desensamblador de ficheros de clase.

javah: generador de ficheros de cabecera C, para crear métodos nativos.

keytool: herramienta para manejar almacenes de claves y certificados.

jarsigner: genera y verifica firmas en archivos jar.

policytool: herramienta para manejar políticas de seguridad.

Para obtener más información acerca de los comandos anteriores y de sus opciones y argumentos se aconseja consultar la documentación oficial de Java proporcionada por Sun Microsystems.

4. La máquina virtual Java.

El JDK de Sun Microsystems proporciona varias implementaciones de la máquina virtual. En las plataformas usadas normalmente para ejecutar aplicaciones cliente la máquina virtual se llama Java *HotSpot Client VM*,, y está optimizada para reducir el tiempo de arranque y la carga de memoria. En las plataformas usadas como servidores se proporciona la Java *HotSpot Server VM*, diseñada para obtener la máxima velocidad de ejecución. Los modificadores –client y –server del comando java sirven para forzar la ejecución de una u otra.

Cuando invocamos el comando java, lo primero que se hace es crear el entorno de ejecución e iniciar el hilo de ejecución principal - *thread main* -, dentro del cual se invoca al cargador de clases o *class-loader*. El cargador de clases es una parte fundamental de la máquina virtual, que se encarga de buscar los *bytecodes* que contienen la definición de las clases, y cargar éstas en memoria, comprobando su integridad y realizando las verificaciones de seguridad necesarias - firmas electrónicas, etc -. La primera clase que intentará cargar el cargador de clases será la especificada como parámetro del comando `java`. Es por ésta razón por la que a dicho comando se le debe especificar el nombre de la clase, y no el nombre del fichero donde se encuentran sus *bytecodes*. Posteriormente, el cargador de clases será invocado durante la ejecución del programa cada vez que se haga referencia a una nueva clase, y justo en el momento en que aparece dicha clase por primera vez. En cualquier caso, si no se consigue cargar una clase por alguna razón, el cargador de clases generará una excepción. Es bien conocida la excepción *NoClassDefFoundError*, producida cuando no se encuentra la definición de una clase.

Otra parte importante de la máquina virtual es la gestión de memoria, y dentro de ésta, el *garbage collector* o recolector de basura. La gestión de memoria en Java es dinámica. Cada vez que se instancia un objeto, se le reserva la memoria necesaria. La máquina virtual dispone de un espacio de memoria seguro que sirve como memoria dinámica. A diferencia de otros lenguajes, que obligan al programador a liberar explícitamente la memoria ocupada por los objetos que han dejado de usarse, Java libera al programador de esta tarea - y potencial fuente de errores - por medio del *garbage collector*. Éste se encarga de averiguar qué objetos han quedado sin referencias, y que por lo tanto, no se volverán a usar. Entonces, destruye tales objetos y libera su memoria. Dicho proceso de búsqueda de objetos sin uso es relativamente costoso, por lo que el recolector de basura no está funcionando constantemente, sino que se ejecuta periódicamente o cuando hay necesidad de liberar memoria. No obstante, el programador puede invocarlo explícitamente para aprovechar tiempos muertos, mediante una llamada al método `gc()` de la clases `System`.

5. Los paquetes Java.

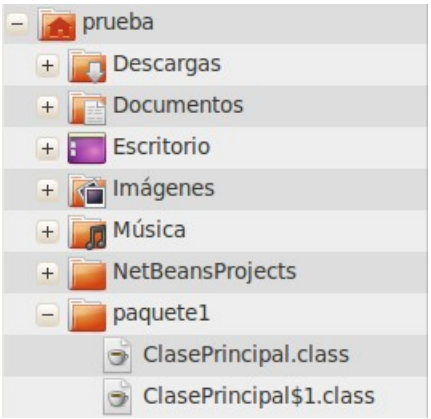
Las clases Java - y los ficheros que contienen sus definiciones - se agrupan en otras unidades superiores llamadas paquetes. Un paquete es una estructura que puede contener definiciones de clases, recursos de éstas y otros paquetes de nivel inferior. Los paquetes Java se ajustan por tanto a una estructura jerárquica. De hecho, normalmente los paquetes Java se soportan físicamente en directorios, de tal forma que cada paquete se corresponde con un directorio, y sus clases y recursos se corresponden con ficheros de dicho directorio. Además, los paquetes de nivel inferior se corresponderían con subdirectorios.

De esta forma, los paquetes forman un árbol de directorios o sistema de ficheros, que colgarán de un directorio al que llamaremos punto de montaje. Para que la máquina virtual sea capaz de encontrar las definiciones de clases dentro de los paquetes a los que pertenecen, tenemos que indicarle uno o varios puntos de montaje. Para ello se usa la variable del sistema `CLASSPATH`. Dicha variable se define en el sistema operativo y constará de una lista con las rutas absolutas de los puntos de montaje de los paquetes Java, usando como separador de la lista el carácter '**;**' en sistemas de Microsoft y el carácter '**:**' en sistemas Unix.

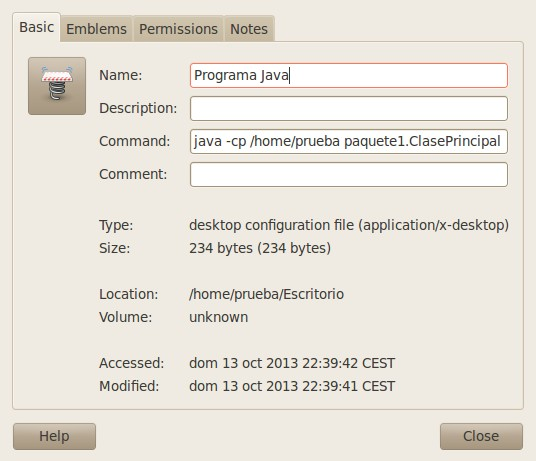
La máquina virtual le pedirá el valor de la variable `CLASSPATH` al sistema operativo y a partir de ese momento el cargador de clases será capaz de buscar las definiciones de clases a través de los paquetes, comenzando la búsqueda a partir de los puntos de montaje que aparecen primero en la lista proporcionada por la variable `CLASSPATH`.

Otra forma de especificar los directorios donde se encuentran las clases Java es mediante el modificador `-classpath` o `-cp` en el comando `java`, seguido de la lista de puntos de montaje tal y como se escribirían en la variable `CLASSPATH`.

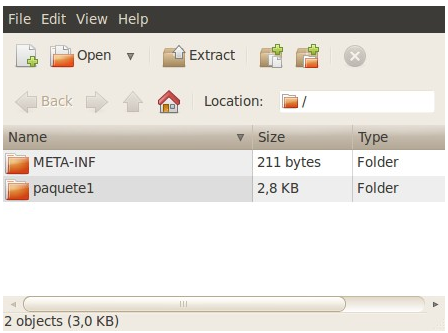
Supongamos, por ejemplo, que tenemos la siguiente estructura, en la que el directorio `paquete1` representa un paquete Java que contiene la clase llamada `ClasePrincipal`. Si el directorio paquete1 se encuentra dentro de /`home`/`prueba`/ el siguiente comando serviría para ejecutar el programa `ClasePrincipal`:



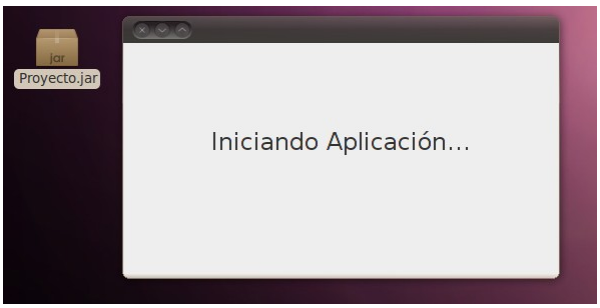

```
java -cp /home/prueba paquete1.ClasePrincipal
```



Si además, queremos instalar un icono para que el usuario ejecute la aplicación Java fácilmente – haciendo doble click, generalmente – sólo tenemos que crear un acceso directo – Windows – o un lanzador – sistemas Unix – con el comando anterior. En la figura se observan las propiedades de dicho lanzador en un sistema Ubuntu. Tanto los accesos directos de Windows como los lanzadores de Unix pueden ser instalados directamente en el escritorio o pueden copiarse en las carpetas que contienen los menús del sistema: "`Carpeta de Usuario\Menu Inicio`" en Windows XP y "`/usr/share/applications/`" en un sistema Ubuntu 10.04. Estos directorios pueden depender de los Sistemas Operativos y sus versiones. Conviene recordar también, que tanto los accesos directos como los lanzadores son ficheros normales, en el primer caso con extensión .lnk y en el segundo con extensión .desktop, lo que implica que podemos copiar dichos ficheros o crearlos de forma automatizada mediante un programa de instalación.



Además tenemos la posibilidad de almacenar toda la estructura de directorios y ficheros que contienen los paquetes java en un sólo fichero JAR - Java Archive o archivo java - mediante la utilidad del mismo nombre. De esta forma, a la hora de distribuir una aplicación o biblioteca Java sólo tendremos que copiar un fichero. Los archivos java son ficheros con extensión .jar, y generalmente están comprimidos mediante el algoritmo de compresión ZIP, aunque esto es opcional. La herramienta de línea de comando jar tiene una sintaxis y opciones similares a la herramienta tar de los sistemas UNIX, y nos permite crear archivos java, añadir o quitar ficheros y directorios, consultar el contenido de un archivo java y extraer el contenido. Los ficheros Jar contienen además un directorio llamado META-INF que contiene el fichero MANIFEST.MF, con información autocontenida acerca del archivo java.



Otro aspecto importante de los archivos java, es que el cargador de clases de la Máquina Virtual Java, es capaz de buscar las definiciones de tipos Java directamente dentro de los ficheros .jar, de forma que si tenemos una aplicación Java en dicho formato, no hay que extraer el contenido del archivo para poder ejecutarla. De hecho, podemos especificar la ruta del fichero Jar como punto de montaje en el

CLASSPATH, y el cargador de clases buscará los paquetes y los ficheros adecuados dentro de dicho fichero. Si además el fichero MANIFEST.MF de un archivo java contiene la sección Main-Class indicando una clase ejecutable dentro del archivo – una clase con el método main() -, podemos usar la opción -jar del comando java para indicarle el fichero .jar que contiene dicho archivo evitando tener que especificar la clase java que queremos ejecutar, ya que la máquina virtual la leerá directamente del fichero MANIFEST.MF.

6. Multithreading.

El término multithreading – multihilo – hace referencia a la capacidad de un programa para ejecutar varias tareas de forma simultánea. A éste mecanismo se le conoce también como programación concurrente, y en contraposición al modelo de programación secuencial, permite iniciar partes del programa como hilos simultáneos mientras continúan ejecutándose las siguientes instrucciones del hilo original.

La creación de hilos en Java, como era de esperar, se basa en una estructura de clases e interfaces que debemos extender o implementar en nuestros programas. Para esto disponemos de dos opciones: podemos crear una clase que herede de la clase Thread o que implemente la interfaz Runnable - ambas pertenecen al paquete java.lang de la API -. Nuestra clase deberá implementar en cualquier caso el método public void run(), que contendrá el código ejecutable del nuevo hilo.

Para poner en marcha el nuevo hilo deberemos crear un objeto de la clase que hemos definido, y llamar a su método start(). A partir de ese momento empezará a ejecutarse de forma paralela nuestro nuevo hilo, ejecutando el método run(), y devolviendo el control inmediatamente a la instrucción siguiente a la llamada a start() en el hilo original.

Es en la creación del nuevo objeto donde difieren significativamente las dos estrategias anteriormente expuestas. En el primer caso, cuando nuestra clase hereda directamente de la clase Thread, sólo tenemos que crear un objeto de nuestra clase de forma normal, llamando a su constructor. En el segundo caso, cuando nuestra clase implementa la interfaz Runnable, tendremos que crear igualmente un objeto de nuestra clase y posteriormente crear un objeto de la clase Thread, llamando a un constructor en el que le pasaremos como parámetro nuestro objeto. La segunda solución, aunque menos directa, nos permite que nuestra clase pueda heredar de otra distinta a Thread - no olvidemos que en Java sólo se puede heredar de una clase -. Por lo demás, ambas aproximaciones son equivalentes.

A continuación se muestra un esquema de la definición y creación de un nuevo hilo heredando de la clase Thread:

```
// Definición de la clase que implementa nuestro hilo
class Hilo extends Thread {
    // Declaramos un parámetro inicial para nuestro hilo
    long parametro;
```

```
        Hilo(long parametro) {
            this.parametro = parametro;
        }

        public void run() {
            // comenzamos la ejecución del nuevo hilo
            ...
        }
    }
    ...
    ...
    // Creamos un objeto de la clase Hilo con un valor inicial
    // y comenzamos su ejecución en paralelo

    Hilo h = new Hilo(7);
    h.start();
    ...
```

El siguiente fragmento de código hace lo mismo, pero esta vez, implementando el interfaz Runnable:

```
    // Definición de la clase que implementa nuestro hilo
    class Hilo implements Runnable {
        // Declaramos un parámetro inicial para nuestro hilo
        long parametro;
        Hilo(long parametro) {
            this.parametro = parametro;
        }

        public void run() {
            // comenzamos la ejecución del nuevo hilo
            ...
        }
    }
    ...
    ...
    // Creamos un objeto de la clase Hilo con un valor inicial
    // y comenzamos su ejecución en paralelo

    Hilo h = new Hilo(7);
    new Thread(h).start();
    ...
```

La clase Thread nos ofrece algunos métodos estáticos que sirven para controlar el hilo actual, entre los que destacamos los siguientes:

- currentThread(): devuelve un objeto de tipo Thread que hace referencia al hilo actual y que nos servirá para acceder a él con los métodos de instancia.
- sleep(): realiza una pausa del hilo actual durante el tiempo indicado en milisegundos por el parámetro de tipo long. Existe una versión que toma además un segundo parámetro de tipo int con un número de nanosegundos adicionales para realizar esperas de mayor precisión. Durante la pausa de un hilo, los demás podrán seguir ejecutándose.
- yield(): indica al planificador de la máquina virtual que el hilo actual quiere ceder su uso actual de procesador a otros hilos.

También disponemos de métodos de instancia que sirven para controlar el comportamiento del hilo sobre el que los ejecutamos. Algunos de los más interesantes son:

- getName()/setName(): Consulta/asigna el nombre del hilo.
- getPriority()/setPriority(): Consulta/asigna la prioridad del hilo.
- isDaemon()/setDaemon(); Consulta/establece si el hilo es un demonio, en cuyo caso continuaría ejecutándose aún cuando la aplicación hubiese terminado. Esto es útil para crear servicios.
- join(): El hilo actual suspende su ejecución hasta que el hilo con el que se ejecuta el método termine. Sirve para coordinar la ejecución de varios hilos. Existen versiones sobrecargadas en las que se especifica un tiempo máximo de espera.
- start(): Comienza a ejecutar en paralelo el hilo objetivo. Éste método devolverá el control inmediatamente al hilo original, que seguirá ejecutándose simultáneamente con el nuevo.

Para obtener una información más detallada sobre la clase Thread y sus métodos se recomienda consultar la referencia de la API en http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html.

Sincronización de hilos. Cuando dos o más hilos acceden a un objeto compartido para actualizarlo, aparece un problema bastante grave, tanto por su impredictibilidad – y la consecuente dificultad para reproducirlo - como por sus posibles resultados, que pueden llevar a datos incoherentes. El problema viene provocado porque todas las instrucciones de un programa, incluso las más sencillas, generan varias instrucciones de código máquina, y además, no podemos hacer ninguna suposición acerca del orden en que se ejecutarán las instrucciones de distintos hilos. Todo esto provoca, que algunas instrucciones trabajen con datos desactualizados – valores previos a una actualización realizada por otro hilo -.

Veámoslo con el siguiente ejemplo. Supongamos que tenemos la clase Entero que se muestra a continuación:

```
public class Entero {
    private int i;
    public Entero(int n) {
        i = n;
    }

    public void incrementar() {
        i++;
    }

    public void decrementar() {
        i--;
    }

    public int valor() {
        return i;
    }
}
```

Ahora supongamos que creamos una instancia de la clase Entero con un valor inicial 0, y que ponemos en ejecución dos hilos paralelos de forma que uno de ellos ejecutará el método incrementar() sobre dicho objeto un número N de veces, y el otro ejecuta el método decrementar() sobre el mismo objeto, el mismo número de veces. Resulta evidente que independientemente del orden en que se ejecuten las instrucciones de los dos hilos, y del valor que le demos a N, el resultado final debería ser 0. Pues bien, si lo probamos con N bastante grande – para valores pequeños hay menos probabilidades de que ocurra – observaremos que obtenemos resultados distintos de 0, y además distintos entre sí en cada ejecución. Esto se debe a que en algunas ocasiones ocurre algo como esto: el hilo A ejecuta el método incrementar cuando el objeto tiene el valor 7, por ejemplo. La instrucción i++ se compone de tres instrucciones de código máquina: leer i, sumar 1, y escribir i. Supongamos que cuando se ha ejecutado el primero de esos tres pasos, entra en ejecución el hilo B, que ejecuta el método decrementar. Este método también se compone de tres pasos, pero supongamos que se ejecutan de forma completa. Como el valor contenido en el objeto era 7, ahora pasará a valer 6. Ahora supongamos que se termina de ejecutar la llamada a incrementar() en el hilo A. Como ya se había realizado la lectura de memoria – con un valor de 7 – ahora se incrementa ese valor y se almacena en memoria. El resultado es que el objeto

pasa a tener un valor de 8, cuando debería permanecer con el valor original, ya que se ha ejecutado una operación de incremento y otra de decremento.

Éste problema es general y se da siempre que varios hilos acceden a un mismo objeto para actualizarlo en paralelo.

La solución al problema es la sincronización, que consiste en sincronizar las instrucciones de los distintos hilos que actualizan el objeto compartido, para que esas instrucciones nunca se ejecuten en paralelo. Para ello podemos usar la palabra reservada synchronized en dos modalidades: métodos sincronizados o bloques sincronizados. La primera solución consiste en declarar como sincronizados los métodos que realizan actualizaciones del objeto compartido. De esta forma el monitor de procesos de la máquina virtual establecerá un bloqueo sobre el objeto cunado el primer hilo empiece a ejecutar un método sincronizado. Dicho bloqueo no se liberará hasta que no termine la ejecución del método. Si en ese transcurso, otro hilo intenta ejecutar un método sincronizado sobre el mismo objeto, al existir un bloqueo, se quedará en espera hasta que el objeto sea desbloqueado, momento en el cuál, éste método podrá ejecutarse obteniendo a su vez un nuevo bloqueo sobre el objeto. De esta forma se asegura que nunca se podrán ejecutar simultáneamente dos métodos sincronizados sobre el mismo objeto.

Aplicando dicha solución a nuestro problema sólo tendríamos que realizar la siguiente modificación en los métodos afectados:

```
public synchronized void incrementar() {
    i++;
}

public synchronized void decrementar() {
    i--;
}
```

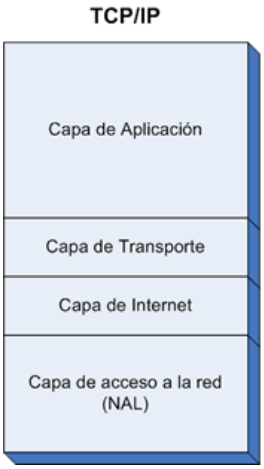
En el caso de que tengamos métodos que realicen varias operaciones y sólo una sección limitada sea la que puede generar problemas de sincronización – la que actualiza un objeto compartido – podemos optar por declarar un bloque sincronizado, que contendrá dicha sección. La sintaxis será :

```

```

```
...
synchronized(objetoCompartido) {
    // actualización del objeto compartido
    ...
}
...
```

3. TCP/IP. Modelo Cliente-Servidor.



La familia de protocolos ***TCP/IP*** se estructura en cuatro niveles o capas, de tal forma que los niveles más bajos se encargan de los asuntos relacionados con el hardware de red y los más altos están más cercanos a las aplicaciones de usuario. La familia toma el nombre de los dos protocolos más importantes de los niveles intermedios: *TCP* e *IP*. Dichos protocolos son los que se usan de forma estándar en Internet, y por extensión, en la gran mayoría de las redes actuales.

La capa de acceso a la red está relacionada con la conexión física a la red – interfaz de red, conexiones, tipos de señal, etc – y con la transferencia de secuencias de bits entre equipos conectados a un mismo medio. El protocolo de ésta capa depende, por tanto, de la infraestructura de red a la que está conectado el equipo, y será independiente de los protocolos de los niveles superiores. El protocolo más habitual en éste nivel es Ethernet en sus distintas modalidades – par trenzado, wireless, fibra óptica -.

En la capa de ***InterRed*** – InterNet, en inglés – se encuentra el protocolo *IP* – Internet Protocol -, que se encarga de conectar equipos de distintas redes, siempre que éstas estén interconectadas. Para ello *IP* soluciona el direccionamiento de red y el encaminamiento o *routing*. Una dirección IP identifica de forma única a un equipo o *host* en Internet. Cada paquete de datos que viaja por la red lleva una *cabecera IP* que, entre otras cosas, contiene la *dirección IP* de destino y la de origen. De esta forma los *routers* pueden utilizar esa información para encaminar cada paquete hacia su destino.

El protocolo más importante del ***nivel de transporte*** es *TCP* – Transmission Control Protocol – que se encarga de varias funciones como garantizar una comunicación fiable, realizar direccionamiento de nivel de transporte y la segmentación / reensamblaje de datos. Hay que tener en cuenta que el protocolo de nivel inferior – *IP* – no asegura una comunicación fiable, ya que algunos paquetes pueden perderse, repetirse o llegar desordenados, ya que cada paquete IP viaja por la red de forma independiente. Para garantizar una comunicación fiable, el protocolo TCP añade en las cabeceras de sus paquetes un *número de secuencia* para cada paquete de una conexión, y mediante un mecanismo de respuestas de confirmación y reenvios soluciona el problema de las pérdidas de paquetes. Si llegan paquetes desordenados, éstos se detectan en el *host* de destino y se reordenan. En el

caso de que un paquete llegue repetido, simplemente se ignora y se envía una respuesta de confirmación.

El direccionamiento a nivel de transporte se lleva a cabo mediante los *puertos TCP*, que identifican a las aplicaciones de red dentro de cada *host*. Así, aunque tengamos más de una aplicación realizando conexiones de red en un equipo, cada una de ellas tendrá asignado un número de puerto distinto, de tal forma que el protocolo *TCP* podrá identificar a qué aplicación pertenece cada paquete de datos. En las *cabeceras TCP* se añaden el puerto de destino y el de origen, que identifican respectivamente la aplicación a la que va dirigido el paquete en el *host* de destino, y la que lo generó en el *host* de origen.

El protocolo *TCP* ofrece a la aplicación – que se encuentra en el nivel inmediatamente superior – un servicio orientado a conexión. Esto significa que *TCP* crea una secuencia de bytes para el envío y otra para la recepción, de tal forma que la aplicación podrá escribir en la primera y leer de la segunda mientras permanezca abierta la conexión. Teniendo en cuenta que el protocolo *TCP* envía y recibe paquetes, es evidente que tiene que realizar la operación de *segmentación* – descomponer la secuencia de datos en paquetes – en el envío, y *reensamblado* en la recepción – unir los paquetes recibidos para formar una secuencia - .

En el nivel de ***Aplicación***, se encontrará el protocolo de comunicaciones de alto nivel, directamente relacionado con la aplicación de usuario. Algunos de los protocolos estándar de Aplicación son *HTTP* – HyperText Transfer Protocol -, *FTP* – File Transfer Protocol -, *SMTP* – Simple Mail Transfer Protocol - , etc.

Así, por ejemplo, si usamos un navegador para conectarnos a un servidor *HTTP*, tanto el navegador en el *host* local como el servicio *HTTP* en el servidor implementan el protocolo *HTTP* en el nivel de *aplicación*.

Es en el nivel de *aplicación*, precisamente en el que nosotros desarrollaremos las comunicaciones de red para nuestras aplicaciones, usando los servicios proporcionados por los niveles inferiores, principalmente *TCP*.

En las redes *TCP/IP* se usa el modelo ***Cliente – Servidor*** que distingue los dos extremos de una comunicación. De esta forma el servidor es un programa o servicio de red que se encuentra permanentemente en espera de conexiones desde los clientes. El cliente es siempre el que inicia una conversación, realizando una conexión con el servidor. Cuando esto ocurre, el servidor inicia la conversación con el cliente y a partir de ese momento ambos mantienen la conexión como iguales hasta que uno de ellos la cierra. Normalmente el servidor, cuando inicia una conexión con un cliente, se sigue manteniendo a la escucha para atender conexiones de otros clientes.

El proceso servidor se enlaza a un puerto específico, que será conocido por los clientes con el fin de poder establecer la conexión con dicho puerto. Los puertos más bajos – desde el 0 hasta el 1023 – están reservados para los servicios estándar. Así, por ejemplo, cuando un cliente web se conecta a un servidor, lo hace poniendo como dirección IP de destino la del *host servidor*, y como puerto de destino, el puerto 80 – que es el puerto predeterminado para el protocolo *HTTP* -. Los procesos clientes, por el contrario, se enlazan

a un puerto asignado automáticamente por el sistema.

4. Manejo de URL's.

Una ***URL*** – Uniform Resource Locator – es una cadena de texto con un formato determinado que sirve para identificar un recurso – fichero, imagen, página web, etc – que se encuentra en un servidor.

Las partes más importantes de una *URL* son:

- Protocolo: identifica el protocolo usado por el servicio que proporciona el recurso. Ejemplos: `http`, `https`, `ftp`, `smtp` ...
- Host: el nombre o dirección del host servidor. Ejemplo: `www.google.com`
- Puerto(opcional): indica el puerto en el que se encuentra el servicio. Si no se especifica se toma el puerto por defecto para el protocolo indicado. Se separa del host mediante el carácter ":".
- Recurso (opcional): Indica el recurso que queremos obtener del servidor. Se puede indicar una ruta dentro del servidor. En ese caso se usa "/" como separador de ruta. Ejemplos: `index.html`, `imagenes/logo.gif`, …
- Otras opciones, como parámetros, anclas, etc. Se especifican al final y dependen del protocolo usado. Ejemplos: `#ancla` , `?parametro=valor`, …

Un ejemplo de URL puede ser

```
http://docs.oracle.com/javase/7/docs/api/java/net/URL.html
```

donde el protocolo es **`http`**, el host servidor es **`docs.oracle.com`**, y el recurso es **`javase/7/docs/api/java/net/URL.html`**.

En la API java.net existe la clase ***URL***, que además de encapsular una dirección *URL* con todas sus partes, proporciona métodos para realizar la conexión con el servidor para obtener el recurso especificado.

Para crear un objeto *URL* podemos usar cualquiera de sus constructores, de los cuales, el más sencillo toma como parámetro una cadena con la dirección *URL*. Existen otros constructores para crear una *URL* a partir de sus partes, o una *URL* relativa a otra, etc.

Una vez creado el objeto *URL*, podemos conectarnos al servidor y obtener el recurso especificado mediante el método *openStream()*, que devuelve un objeto de tipo *InputStream*, del cual podemos leer secuencias de bytes mientras permanezca abierto.

El siguiente fragmento de código muestra un método que lee el texto – código fuente *HTML* – devuelto por el servidor cuando se le solicita la *URL* pasada como parámetro.

```
java -cp /home/prueba paquete1.ClasePrincipal
```

```
public void leerHTML(String cadenaURL) {
  URL url = null;
  try {
      // Creamos una URL a partir del texto introducido
      url = new URL(cadenaURL);
  } catch (MalformedURLException e) {
      salida.setText("URL incorrecta");
      return;
  }
  salida.setText("");
  BufferedReader entrada = null;
  try {
      // Obtenemos una secuencia de entrada de caracteres
      // para leer la respuesta del servidor
      entrada = new BufferedReader(new
                    InputStreamReader(url.openStream()));
  } catch (IOException e) {
      salida.setText("Error al abrir " + url.getHost());
  }
  String linea;
  try {
      // Mientras el servidor mantenga la conexion abierta..
      while ((linea = entrada.readLine()) !=null)
          salida.append(linea + "\n");
  } catch (IOException e) {
      System.err.println(e.getMessage());
  }
  try {
      // cerramos la secuencia de entrada
      entrada.close();
  } catch (IOException e) {
```

```
            System.err.println(e.getMessage());
        }
    }
```

Ésta es la forma más directa de obtener un recurso del servidor cuando no se necesita mayor interacción.

También disponemos de otra técnica más elaborada para conectarnos a un servidor mediante la clase *URL*. Usando el método *openConnection()* obtenemos un objeto de tipo *URLConnection*, que nos permite establecer la conexión con el servidor, obtener una secuencia de entrada para leer los datos recibidos y una secuencia de salida para enviar datos al servidor.

El método *getInputStream()* de la clase *URLConnection* nos devuelve un objeto de tipo *InputStream*, a través del cuál podemos leer los datos recibidos, de la misma forma que lo hacíamos para leer directamente del objeto *URL*.

Además, el método *getOutputStream()* devuelve un objeto de tipo *OutputStream* que permite escribir secuencias binarias – o de texto, si realizamos las conversiones adecuadas -, que serán enviadas al servidor. Para hacer esto, debemos habilitar previamente la escritura en el objeto *URLConnection*, mediante una llamada al método *setDoOutput()*.

El siguiente fragmento de código muestra cómo conectarse a un recurso web usando el método *POST* y enviando un parámetro en la petición.

```
        URL url = null;
        try {
            url = new URL("http://localhost:8080/Web/cabeceras.jsp");
        } catch (Exception e) {}
        URLConnection conexion;
        try {
            conexion = url.openConnection();
        } catch (IOException e) {
            System.err.println(e.getMessage());
            return;
        }
        // Habilitamos el envio de datos
        conexion.setDoOutput(true);
```

```
// Establecemos el metodo HTTP como POST
conexion.setRequestProperty("METHOD", "POST");
try {
    BufferedWriter salida = new BufferedWriter(new
    OutputStreamWriter(conexion.getOutputStream()));
    salida.newLine();
    salida.newLine();
    String parametros = "parametro=valor";
    salida.write(parametros);
    salida.close();
    BufferedReader entrada = new BufferedReader(new
    InputStreamReader(conexion.getInputStream()));
    String linea = null;
    while ((linea = entrada.readLine())!=null) {
        System.out.println(linea);
    }
    entrada.close();
} catch (Exception e) {
    System.err.println(e.getMessage());
```

4. Tratamiento de sockets en Java. Diseño de programas clientes y servidores.

Un ***socket*** representa un extremo de la conexión entre dos programas en la red. Ofrece un flujo de entrada (*InputStream*) y un flujo de salida (*OutputStream*) que se usan para la recepción y el envío de datos respectivamente. Dichos flujos son binarios (flujos de bytes), aunque se pueden encapsular en objetos de lectura y escritura de cadenas de caracteres (*BufferedReader* y *PrintWriter*, por ejemplo).

La escritura de un programa cliente consta de los siguientes pasos:

- Creación de un *socket* y establecimiento de la conexión mediante **`new Socket(servidor,puerto)`**. El primer parámetro es una cadena que contendrá la dirección del *host servidor* – ya sea mediante un nombre de dominio o una dirección IP -, y el segundo parámetro será un número entero que indicará el *puerto TCP* en el que se encuentra el servicio al que nos queremos conectar.

- Obtención de los flujos de entrada y salida mediante **`getInputStream()`** y **`getOutputStream()`** respectivamente.

- Comunicación con el servidor mediante lecturas y escrituras a través de los flujos obtenidos. La secuencia de las lecturas y escrituras y la semántica de los mensajes dependen del *protocolo de aplicación*.

- Cierre de los flujos de datos y del propio *socket*.

En el siguiente fragmento de código se muestra la secuencia anterior, omitiendo los bloques de control de excepciones, para simplificar:

```
        String servidor = “localhost”;
        int puerto =  7;
        …
        // Establecemos la conexión con el servidor en el puerto indicado
        Socket conexión = new Socket(servidor,puerto);
        …
        // Obtenemos las secuencias de entrada y de salida de la conexión
       BufferedReader entrada = new BufferedReader(new
InputStreamReader(conexion.getInputStream()));
        PrintWriter salida = new
PrintWriter(conexion.getOutputStream(),true);
        …
        // Mantenemos conversacion con el servidor
        salida.println(“mensaje”);
        …
        String mensajeRecibido = entrada.readLine();
       …
        // Cerramos secuencias de entrada y salida y conexión
        salida.close();
        entrada.close();
        conexión.close();
```

La escritura de un programa servidor difiere de la anterior, principalmente, en la forma de realizar la conexión. El servidor debe registrarse en el puerto reservado para el servicio, y a partir de ese momento aceptará conexiones de clientes. Cada vez que se acepta una conexión, se obtiene ésta en forma de un *socket*, que ya se puede utilizar para mantener la conversación con el cliente:

- Creación de un *socket servidor* y enlace a un puerto mediante **new**

`ServerSocket(puertoServidor)`.

- Aceptando conexiones de clientes y obteniendo *sockets cliente* para realizar la comunicación: **`socketCliente = socketServidor.accept()`**.
- Obtener las secuencias de entrada y salida de éste *socket* y mantener la comunicación con el cliente mediante lecturas y escrituras sobre dichas secuencias siguiendo las reglas definidas en el protocolo de aplicación.
- Cerrar secuencias y *socket* cliente.
- Cerrar el *socket* servidor.

Para aceptar múltiples conexiones simultáneas se crea un *hilo de ejecución* para manejar cada *socket cliente* obtenido mediante `socketServidor.accept()`.

El siguiente código muestra un servidor de *echo*:

```
package echo;

import java.net.*;
import java.io.*;

public class EchoServer {

    public static void main(String[] args) {
        // Conectamos el servicio al puerto 4000
        ServerSocket socketServidor = null;
        try {
            socketServidor = new ServerSocket(4000);
        } catch (IOException ex) {
            System.err.println(ex.getMessage());
            System.exit(-1);
        }
        // Aceptamos conexiones
        while (true) {
            try {
                Socket conexion = socketServidor.accept();
                // Creamos un nuevo subproceso para cada conexion de
cliente
```

```
                new ConexionCliente(conexion).start();
            } catch (IOException ex) {
                System.err.println(ex.getMessage());
            }
        }
    }

    private static class ConexionCliente extends Thread {
        private Socket conexion;

        public ConexionCliente(Socket s) {
            conexion = s;
        }

        @Override
        public void run() {
            try {
                // Obtenemos las secuencias de entrada y salida de la
conexion
                BufferedReader entrada = new BufferedReader(new
InputStreamReader(conexion.getInputStream()));
                PrintWriter salida = new
PrintWriter(conexion.getOutputStream(),true);
                // Mientras la entrada de la conexion permanezca abierta
                // Leemos mensaje y lo retransmitimos
                String linea;
                while ((linea = entrada.readLine()) != null) {
                    salida.println(linea);
                }
                // Cerramos la conexion
                conexion.close();
            } catch (IOException ex) {
                System.err.println(ex.getMessage());
            }
        }
    }
}
```