Resumen JAVA PSP

I. Compilación de java

Para poder usar los archivos `.java` de la carpeta “src”, hay que compilar y ejecutar los archivos en la terminal:

- **Compilar:** `javac -d bin directorio(src)/nombre_paquete/*.java`
- **Ejecutar:** `java -cp bin nombre_paquete.nombre_archivo` Con esto se pueden usar los procesos que se han creado en la carpeta “bin”.

1. Clases de Flujos y Ficheros (I/O)

Clase	Constructores
File	`File(String pathname)` / `File(String parent, String child)`
Métodos (Servicios)	**Funcionamiento**
`boolean mkdir()`	Crea el directorio.
`boolean mkdirs()`	Crea el directorio y los superiores si no existen.
`String[] list()`	Lista el contenido de un directorio (devuelve array de `String`).
`File[] listFiles()`	Lista el contenido (devuelve array de `File`).
`boolean delete()`	Borra el archivo o directorio (si está vacío).
`boolean createNewFile()`	Crea el fichero físicamente.

Clase	Constructores
FileReader (Lectura caracteres)	`FileReader(File fichero)` / `FileReader(String fileName)`
Métodos (Servicios)	**Funcionamiento**
`int read()`	Lee un carácter y lo devuelve (devuelve -1 al final).
`int read(char[] buf)`	Lee hasta `buf.length` caracteres.

Clase	Constructores

FileWriter (Escritura caracteres)	`FileWriter(File fichero)` / `FileWriter(String fileName)` `FileWriter(String fileName, boolean append)` (Si `append` es `true`, añade información al final)
Métodos (Servicios)	**Funcionamiento**
`void write(int c)`	Escribe un carácter.
`void write(char[] buf)` / `void write(String str)`	Escribe un array o una cadena.
`void append(char c)`	Añade un carácter.

Clase	**Constructores**
BufferedReader (Buffer lectura caracteres)	`BufferedReader(Reader input)`
Métodos (Servicios)	Funcionamiento
`String readLine()`	**Lee una línea completa o `null` si es el final**.

Clase	**Constructores**
BufferedWriter (Buffer escritura caracteres)	`BufferedWriter(Write r out)`
Métodos (Servicios)	Funcionamiento.
`void newLine()`	Inserta una línea nueva.

Clase	**Constructores**
DataInput/DataOutput	Se construyen envolviendo otros flujos (e.g., `DataOutputStream(new FileOutputStream(... ))`).
Métodos (Servicios)	**Funcionamiento**
`read/writeBoolean, read/writeChar, read/writeDouble, read/writeInt, read/writeUTF`	Lectura/Escritura de tipos primitivos (el último para cadenas).

Clase	**Constructores**

RandomAccessFile (Acceso Aleatorio)	`RandomAccessFile(String ruta, String modo)` / `RandomAccessFile(File fichero, String modo)`
Métodos (Servicios)	**Funcionamiento**
(Modos de uso)	`"r"` solo lectura, `"rw"` lectura/escritura.
`void seek(long pos)`	Posiciona el puntero en `pos` bytes desde el inicio.
`long getFilePointer()`	Devuelve la posición actual del puntero.
`long length()`	Devuelve la longitud del fichero.

2. Clases de Gestión de Procesos

Las clases `ProcessBuilder` y `Process` son las principales para la creación y control de procesos nativos del sistema.

```
// Creamos el proceso pasándole el nombre y la línea como argumentos
builder = new ProcessBuilder(
    ...command:"java",
    "practicaUT1.LeerFichero",
    file.getName()
);
builder.directory(new File(pathname:"practicaUT1/bin"));
```

```
String classPath = System.getProperty(key:"java.class.path");
builder = new ProcessBuilder(
    ...command:"java",
    "-cp",
    classPath,
    "practicaUT1.LeerFichero",
    file.getName()
);
```

Clase	**Constructores**
ProcessBuilder	`ProcessBuilder(List <String> command)` / `ProcessBuilder(String... command)`
Métodos (Servicios)	**Funcionamiento**
`List<String> command()`	Devuelve el programa y argumentos.
`ProcessBuilder directory(File directory)`	Establece el directorio de trabajo.
`Map<String,String> environment()`	Devuelve el mapa de variables de entorno.

`Process start()`	Inicia un nuevo proceso.
`ProcessBuilder redirectErrorStream(boolean redirectErrorStream)`	Combina `stderr` y `stdout` si es `true`.
`ProcessBuilder.Redirect redirectOutput()/redirectInput()/redirectError()`	Obtiene o establece la fuente/destino de E/S/Error.

Clase	**Constructores**
ProcessBuilder.Redirect (Clase Anidada Estática)	N/A (Se usan métodos estáticos para generar las instancias)
Métodos (Servicios)	**Funcionamiento**
`Redirect.INHERIT`	La E/S/Error del subproceso es la misma que la de la consola actual.
`Redirect.from(File)`	Redirige la entrada (`stdin`) para leer desde un fichero `File`.
`Redirect.to(File)`	Redirige la salida (`stdout` o `stderr`) para escribir en un fichero `File`.
`Redirect.appendTo(File)`	Redirige la salida (`stdout` o `stderr`) para añadir información al final de un fichero `File`.

Clase	**Constructores**
Process	`Process()` (Instanciado por `ProcessBuilder.start()`)
Métodos (Servicios)	**Funcionamiento**
`InputStream getInputStream()`	Lee la salida normal (`stdout`) del subproceso.
`InputStream getErrorStream()`	Captura la salida de error (`stderr`) del subproceso.
`OutputStream getOutputStream()`	Envía datos a la entrada estándar (`stdin`) del subproceso.
`int waitFor()`	Hace que el hilo actual espere hasta que el proceso termine. Devuelve 0 si termina normalmente.
`int exitValue()`	Devuelve el valor de salida (lanza `IllegalThreadStateException` si no ha terminado).
`void destroy()`	Termina forzosamente el subproceso.

3. Clases de Threads (Hilos)

Clase	Constructores
Thread	`public Thread ()` `public Thread (Runnable target)` `public Thread (Runnable target, String name)` `public Thread (String name)` `public Thread (ThreadGroup group, Runnable target)` `public Thread (ThreadGroup group, String name)`
Métodos (Servicios)	**Funcionamiento**
`String getName()`	Devuelve el nombre.
`void start()`	Inicia la ejecución del *thread*.
`void run()`	Contiene el código que ejecuta el *thread*.
`int getPriority()` / `setPriority(int newPriority)`	Consulta/establece prioridad.
`void join()`	Espera a que un *thread* termine su ejecución.
`static void sleep(long milis)`	Bloquea el *thread*.
`boolean isAlive()`	Mira si un *thread* está vivo.
`static Thread currentThread()`	Obtiene el hilo en ejecución actual.

4. Clases de Sockets

Clase	Constructores
Socket (Cliente TCP)	`public Socket ()` / `public Socket (String host, int port)` / `public Socket (InetAddress address, int port, ...)`
Métodos (Servicios)	**Funcionamiento**
`public void close()`	Cierra el canal.
`public InputStream getInputStream()`	Canal de lectura.
`public OutputStream getOutputStream()`	Canal de escritura.
`public InetAddress getInetAddress()`	Dirección IP remota.

Clase	Constructores
ServerSocket (Servidor TCP)	`public ServerSocket (int port)` / `public ServerSocket (int port, int backlog)` / `public ServerSocket (int port, int backlog, InetAddress bindAddr)`
Métodos (Servicios)	**Funcionamiento**
`public Socket accept()`	Devuelve el *socket* para la comunicación con el cliente.
`public void close()`	Cierra el canal.
`public int getLocalPort()`	Puerto local.

Clase	Constructores
DatagramSocket (UDP)	`public DatagramSocket ()` / `public DatagramSocket (int port)`
Métodos (Servicios)	**Funcionamiento**
`public void send(DatagramPacket p)`	Envía un datagrama.
`public void receive(DatagramPacket p)`	Recibe un datagrama.

Clase	Constructores
DatagramPacket	`public DatagramPacket (byte[] buff, int length)` (para recibir) `public DatagramPacket (byte[] buff, int length, InetAddress address, int port)` (para enviar)
Métodos (Servicios)	**Funcionamiento**
`set/getAddress, set/getData, set/getLength, set/getPort`	Métodos para consultar y actualizar campos.

Ejemplos con Solución Relacionada

Ejercicios UT9 (I/O y Ficheros)

1. Escritura simple en fichero: Crear un fichero llamado "ejercicio01.txt" que contenga la fecha y hora actual.

```
try {
    BufferedWriter writer = new BufferedWriter(new FileWriter("ejercicio01.txt")); //
    writer.write(LocalDateTime.now().toString()); //
```

```
    writer.close(); //
    System.out.println("\nFichero creado correctamente"); //
} catch (IOException ioe) {
    ioe.printStackTrace(); //
}
```

4. Creación de directorios anidados y escritura: Modificar el ejercicio anterior para un nuevo path `/nuevo_directorio/nuevodirectorio2...`.

```
// Código relevante de Ejercicio4
// Se asume lector está inicializado: BufferedReader lector = new BufferedReader(new
InputStreamReader(System.in));
// ...
File directorio = new File(lector.readLine()); //
File archivo = new File(lector.readLine()); //

try {
    if (directorio.mkdirs()) // Utiliza mkdirs() para crear directorios anidados
        System.out.println("Directorio creado.");
    else
        System.out.println("El directorio ya existe.");

    BufferedWriter writer = new BufferedWriter(new FileWriter(directorio.getPath() + "\\" +
archivo.getPath()));
    writer.write(LocalDateTime.now().toString());
    writer.close();
    System.out.println("Archivo escrito");
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

Ejercicios PSP (Procesos)

2.b) Llamada a otro programa y uso de `waitFor()`: Llamar a otro programa (`LeerNombre`), pasar argumentos, y usar `waitFor()` para comprobar el valor de salida del proceso.

```
// Clases Utilizadas: ProcessBuilder, Process.
ProcessBuilder pb = new ProcessBuilder("java", "ejerTanda2.LeerNombre", "Juan");
Process p = pb.start(); // Iniciar el proceso

try {
    int valorSalida = p.waitFor(); // Esperar a que el proceso termine
    System.out.println("El proceso terminó con valor de salida: " + valorSalida); //

    // Capturar la salida normal (stdout)
```

```
    InputStream is = p.getInputStream();
    int c;

    while ((c = is.read()) != -1) { //
        System.out.print((char) c);
    }
    is.close();
} catch (InterruptedException e) {
    // Manejo de InterruptedException
} catch (IOException e) {
    // Manejo de errores de E/S
}
```

II. Clases de Utilidad y Concurrencia Detalladas

A. Clase System (Utilidad y Control de Tiempo)

La clase `System` (`java.lang.System`) contiene varios campos y métodos útiles. Es una clase `final` y **no puede ser instanciada**.

Clase	Constructores
System (Utilidad y Control de Tiempo)	No tiene constructores; no puede ser instanciada.
Métodos/Campos (Servicios)	**Funcionamiento**
`static InputStream in`	El flujo de entrada estándar. Típicamente corresponde a la entrada del teclado.
`static PrintStream out`	El flujo de salida estándar. Se utiliza para escribir una línea de salida simple: `System.out.println(data)`.
`static PrintStream err`	El flujo de salida de error estándar. Por convención, se usa para mostrar mensajes de error que requieren atención inmediata.
`static long currentTimeMillis()`	Devuelve la diferencia, medida en milisegundos, entre el tiempo actual y la medianoche, 1 de enero de 1970 UTC. La unidad de tiempo es milisegundo, pero la granularidad puede ser mayor.
`static long nanoTime()`	Devuelve el valor actual de la fuente de tiempo de alta resolución, en nanosegundos. Solo debe usarse para medir tiempo transcurrido, no tiene relación con el tiempo de pared (wall-clock time). Ofrece mucha más precisión (Nanosegundo: $1 \text{sg}/1,000,000,000$).
`static void exit(int status)`	Termina la Máquina Virtual Java en ejecución. Un argumento distinto de cero indica terminación anormal.

`static String getenv(String name)`	Obtiene el valor de la variable de entorno especificada.
`static Map<String,String> getenv()`	Devuelve una vista de mapa de cadena no modificable del entorno actual del sistema.
`static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`	Copia una porción de un array de la fuente al destino.
`static String lineSeparator()`	Devuelve la cadena separadora de línea dependiente del sistema (e.g., `"\n"` en UNIX, `"\r\n"` en Windows).
`static Properties getProperties()`	Determina el conjunto actual de propiedades del sistema.
`static String getProperty(String key)`	Obtiene la propiedad del sistema indicada por la clave.
`static void setProperty(String key, String value)`	Establece la propiedad del sistema indicada por la clave.
`static int identityHashCode(Object x)`	Devuelve el mismo código *hash* que el método por defecto `hashCode()`.
`static void setIn(InputStream in)`	Reasigna el flujo de entrada estándar.
`static void setOut(PrintStream out)`	Reasigna el flujo de salida estándar.
`static void setErr(PrintStream err)`	Reasigna el flujo de salida de error estándar.
`static void gc()`	Ejecuta el recolector de basura (Garbage Collector).

B. Clase Thread (Hilos)

Un *thread* es un hilo de ejecución en un programa, y la Máquina Virtual Java (JVM) permite múltiples hilos corriendo concurrentemente.

Creación e Invocación

Hay dos maneras de crear un nuevo hilo:

1. **Subclase de `Thread`**: Se declara una subclase que anula el método `run()`. Se instancia la subclase y se llama a **`start()`**.
2. **Implementar `Runnable`**: Se implementa la interfaz `Runnable`, se define `run()`, y la instancia se pasa a un constructor de `Thread`. Luego se llama a **`start()`** sobre el objeto `Thread`.

El método `start()` causa que el hilo comience la ejecución, y la JVM llama al método `run()`. **Nunca es legal intentar iniciar un hilo más de una vez**.

Clase	Constructores	
Thread	`public Thread ()` `(Runnable target)` `(Runnable target, String name)` `(String name)` `(ThreadGroup group, Runnable target)` `(ThreadGroup group, String name)` `(ThreadGroup group, Runnable target, String name)` `(ThreadGroup group, Runnable target, String name, long stackSize)` (Permite especificar el tamaño de la pila, aunque el efecto depende de la plataforma)	
Categoría	**Método / Elemento**	**Funcionamiento**
Control de Ejecución	`void start()`	Inicia la ejecución del hilo. La JVM llama internamente al método `run()`. Nunca se debe llamar más de una vez al mismo hilo.
	`void run()`	Contiene el código que ejecutará el hilo. Si se implementa `Runnable`, es el método de esa interfaz.
	`static void sleep(long millis)`	Detiene temporalmente la ejecución del hilo actual por el tiempo especificado. **No libera** el bloqueo de los monitores que posea.
	`void join()` / `join(long millis)`	Hace que el hilo actual espere hasta que el hilo sobre el que se llama a `join` termine (muera) o pase el tiempo indicado.
	`static void yield()`	*Nota: A veces mencionado en contextos generales, sugiere al planificador ceder el turno, aunque no aparece explícitamente detallado en tus fuentes actuales.*
Estado e Información	`String getName()`	Devuelve el nombre del hilo.
	`static Thread currentThread()`	Devuelve una referencia al objeto hilo que se está ejecutando actualmente.
	`int getPriority()` / `setPriority(int)`	Consulta o cambia la prioridad del hilo (afecta a la planificación de la CPU).
	`boolean isAlive()`	Devuelve `true` si el hilo ha sido iniciado (`start()`) y todavía no ha muerto. Útil para verificar estado sin bloquear.
	`long getId()`	Devuelve el identificador único (positivo y `long`) del hilo.

	`boolean isDaemon()` / `setDaemon(boolean)`	Marca o verifica si el hilo es un "daemon" (servicio en segundo plano). La JVM se cierra si solo quedan hilos daemon vivos.
Interrupción (Parada Correcta)	`void interrupt()`	Establece la bandera de interrupción ("flag"). Si el hilo está bloqueado en `wait`, `sleep` o `join`, lanza una `InterruptedException` y limpia la bandera.
	`static boolean interrupted()`	Comprueba si el hilo actual fue interrumpido y **limpia/borra** el estado de interrupción (lo devuelve a false).
	`boolean isInterrupted()`	Comprueba si el hilo ha sido interrumpido, pero **mantiene** el estado de la bandera intacto.
	`void stop()` **@Deprecated**	Fuerza la parada inmediata liberando monitores abruptamente. Puede dejar objetos en estados inconsistentes o corruptos.
	`void destroy()` **@Deprecated**	Originalmente diseñado para destruir el hilo sin limpieza, pero nunca fue implementado.
Coordinación (Clase Object)	`void wait()` / `wait(long timeout)`	**Libera el monitor** y pone al hilo en espera hasta que otro hilo llame a `notify`/`notifyAll` o pase el tiempo. Debe capturar `InterruptedException`. *Nota: Se usan sobre el objeto monitor y dentro de bloques `synchronized`.*
	`void notify()`	Despierta a **un** solo hilo (elegido arbitrariamente) que esté esperando en el monitor de este objeto.
	`void notifyAll()`	Despierta a **todos** los hilos que esperan en el monitor. Todos compiten de nuevo por el bloqueo para continuar.
Visibilidad y Memoria	`volatile` (Palabra clave)	La lectura de la variable se hace directamente desde la memoria principal (RAM), ignorando la caché de la CPU. Asegura que los cambios de un hilo sean visibles inmediatamente para los demás.

Ejemplo:

```
package ejemplos;

public class HiloEjemplo1 extends Thread {

    // constructor
    public HiloEjemplo1(String nombre) {
```

```
        super(nombre); // Llama al constructor de Thread para ponerle nombre
        System.out.println("CREANDO HILO: " + super.getName());
    }

    // método run (si extiendes de Thread para sobreescribirlo)
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Hilo: " + getName() + " contador = " + i);
        }
    }

    public static void main(String[] args) {
        HiloEjemplo1 h1 = new HiloEjemplo1("Hilo A");
        HiloEjemplo1 h2 = new HiloEjemplo1("Hilo B");
        HiloEjemplo1 h3 = new HiloEjemplo1("Hilo C");

        h1.start();
        h2.start();
        h3.start();

        System.out.println("3 HILOS INICIADOS...");
    } // main
} // HiloEjemplo1
```

Ejemplo de Control de Tiempo (`System` y `Thread.sleep`)

Para medir el tiempo transcurrido o detener el hilo actual:

```
// Medir tiempo transcurrido
long startTime = System.nanoTime(); //
// ... el código siendo medido ...
long estimatedTime = System.nanoTime() - startTime; //
// Para comparar: t1 - t0 < 0 se usa para manejar el desbordamiento numérico.

// Detener el hilo actual
try {
    Thread.sleep(3000); // Bloquea el thread por 3000 milisegundos
} catch (InterruptedException e) {
    e.printStackTrace(); //
}
```

3. Coordinación de Hilos (Productor-Consumidor): Implementar un almacén (Mercado/Buffer) donde el productor espera si está lleno y el consumidor espera si está vacío, utilizando `wait()` y `notify()`.

```
class Market {
    private int item = 0; // Recursos en almacén
```

```
    // Método para consumir (Cliente)
    public synchronized void get() {
        // Mientras no haya productos (condición de espera), liberamos el monitor
        while (item < 1) {
            try {
                wait(); //,
            } catch (InterruptedException e) { }
        }
        item--;
        System.out.println("Cliente compró. Stock actual: " + item);
        notify(); // Notifica al productor que hay hueco libre,
    }

    // Método para producir (Fabricante)
    public synchronized void put() {
        // Mientras el almacén esté lleno (límite 6), esperamos
        while (item >= 6) {
            try {
                wait(); //,
            } catch (InterruptedException e) { }
        }
        item++;
        System.out.println("Fabricante produjo. Stock actual: " + item);
        notify(); // Notifica al cliente que hay stock disponible,
    }
}
// Nota: wait() libera el bloqueo; notify() despierta a un hilo en espera.
```

4. Parada correcta de un hilo mediante `interrupt()`: Detener un hilo que realiza una tarea repetitiva o que podría estar durmiendo (`sleep`), manejando la excepción `InterruptedException`.

```
public class HiloInterrumpido extends Thread {
    public void run() {
        try {
            // Comprobar periódicamente si se ha solicitado la interrupción
            while (!Thread.interrupted()) { //
                System.out.println("El hilo está trabajando...");
                Thread.sleep(1000); // Simula una tarea bloqueante
            }
        } catch (InterruptedException e) {
            // Si se interrumpe durante el sleep/wait, se captura aquí
            System.out.println("Hilo interrumpido mientras dormía. Finalizando..."); //
        } finally {
            System.out.println("Limpiando recursos antes de salir.");
        }
    }
```

```
    // Uso en main:
    // HiloInterrumpido h = new HiloInterrumpido();
    // h.start();
    // ...
    // h.interrupt(); // Solicita la parada de forma cooperativa
}
```

5. Parada mediante Bandera de Control y `volatile`: Uso de una variable booleana para detener un bucle. Se usa `volatile` para asegurar que el cambio de valor sea visible inmediatamente para el hilo, evitando lecturas de la caché de la CPU.

```
public class HiloConBandera implements Runnable {
    // volatile garantiza visibilidad inmediata entre hilos y evita caché de CPU
    private volatile boolean detener = false; //,

    public void run() {
        while (!detener) { // Verifica la bandera en cada iteración
            System.out.println("Hilo en ejecución...");
            // Lógica del hilo...
        }
        System.out.println("Hilo detenido por bandera.");
    }

    public void pararHilo() {
        this.detener = true; // Cambia la bandera desde otro hilo (ej. main)
    }
}
```