

## Repaso PAV

22-mayo-2006

Los siguientes ejercicios están pensados para repasar algunos de los conceptos vistos en el curso y para hablar Java con soltura. Los primeros son bastante elementales, pero pueden despejar alguna duda.

Se recomienda que escribas los programas en el compilador y los pruebes. De lo contrario, casi es preferible no hacerlos. Las explicaciones no desarrollan toda la teoría: también debes tener a la mano los resúmenes del curso, los apuntes del curso (que están en la Intranet) y tus apuntes de clase. Ayuda también un block para garabatear, y dedicar por lo menos dos horas seguidas sin interrupciones al asunto.

Una vez terminado, puedes volver a resolver la práctica 1, los ejercicios del laboratorio, la práctica 2 y el problema del parcial.

1. Define una clase `Persona` con dos atributos: `nombre` (de tipo `String`) y `edad` (de tipo entero).

```
class Persona {
    String nombre;
    int edad;
}
```

Nada especial: `String nombre` quiere decir que `nombre` es de tipo `String`. No decimos ni `public`, ni `private` ni nada, de modo que, por defecto, los atributos de esta clase serán visibles para las clases que estén dentro del mismo paquete (incluido las clases que no pertenecen a ningún paquete).

2. Crea otra clase, `Test`, que tenga un método `main` que cree dos objetos de la clase `persona`, les asigne un valor, y luego imprima el contenido de los objetos.

```
class Test {
    public static void main(String[] args) {
        Persona p1 = new Persona();
        Persona p2 = new Persona();
        p1.nombre = "Otto";
        p1.edad = 21;

        p2.nombre = "Fritz";
        p2.edad = 20;

        System.out.println(p1.nombre + ", " + p1.edad + " años.");
        System.out.println(p2.nombre + ", " + p2.edad + " años.");
    }
}
```

Podemos usar `p1.nombre`, `p1.edad` (y sus correspondientes en `p2`) directamente para conocer o asignar valores a `nombre` y `edad`, porque `nombre` y `edad` no son `private`. Por tanto, por defecto, todas las clases del mismo paquete (que es el caso que suponemos para `Test` y para `Persona`) pueden acceder estas variables directamente.

`Persona p1 = new Persona();` asigna a `p1` el resultado de `new Persona()`. Al usar `new Persona()` estamos usando el constructor por defecto de la clase `Persona` (al no definir nosotros un constructor, Java define uno automáticamente).

3. Declare `nombre` y `edad` como `private`, y reescriba la clase `Persona`.

```
class Persona {
    private String nombre;
```

```

        private int edad;

        /* constructor */
        public Persona(String nombre, int edad) {
            this.nombre = nombre;
            this.edad = edad;
        }
    }
}

```

Hemos definido un constructor `Persona(String nombre, int edad)`, con dos parámetros, porque al declarar `nombre` y `edad` como `private`, necesitamos un modo de inicializar estas dos variables cuando otras clases usan la clase `Persona`. El constructor puede tener también modificadores de acceso. En este caso lo hemos declarado `public` para que pueda usarse desde cualquier clase.

4. Vuelve a escribir la clase `Test` que se pide en el n.2, teniendo en cuenta la nueva definición de `Persona`. Puedes añadir los métodos que necesites a `Persona`.

```

class Test {
    public static void main(String[] args) {
        Persona p1 = new Persona("Otto", 21);
        Persona p2 = new Persona("Fritz", 20);

        System.out.println(p1.getNombre() + ", " + p1.getEdad() );
        System.out.println(p2.getNombre() + ", " + p2.getEdad() );
    }
}

class Persona {
    private String nombre;
    private int edad;

    /* constructor */
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    /* métodos para acceder a nombre y edad */
    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }
}

```

Necesitamos añadir `getNombre()` y `getEdad()` porque ahora `nombre` y `edad` son `private`, y por tanto `main` en la clase `Test` no puede acceder a ellos. Es más, ni siquiera sabe si se llaman `nombre` y `edad`. Sólo sabe que `getNombre()` y `getEdad()` le devuelven los datos que pasó en el constructor al crear el objeto de tipo `Persona`.

`getNombre()` y `getEdad()` podrían tener otros nombres, por ejemplo, `nombre( )` y `edad( )`; o `mataGato( )` y `alimentaPerro( )`. Es decir, la palabra “get” como prefijo es simplemente una convención, no es parte del lenguaje Java. Pero es una convención bastante usada, porque describe qué hace el método. Por eso, estos métodos se suelen llamar “getters”.

Por si quedan dudas, los métodos se definen así (hay un ppt bastante detallado sobre el tema

en miIng):

```
public String getNombre( ) {  
    return nombre;  
}
```

El String subrayado de la primera línea es el tipo de datos que devuelve el método. Lo que devuelve un método no tienen nada que ver con lo que puede imprimir el método a la pantalla usando `System.out.println( )`. Si declaramos que `getNombre` devuelve `String` (en la primera línea), entonces el `return` tiene que devolver un `String` (como es el caso, pues `nombre` es `String`).

Cuando un método no devuelve ningún valor, se escribe **void**. Si nuestro inglés `== null`, don't worry, simplemente hay que acordarse que si el método no va a devolver ningún valor, entonces hay que declarar el método como `void Método( ) { }`.

`public` indica que el método no tiene restricciones de acceso. `private` indicaría que el método sólo puede ser usado por otros métodos de la propia clase (`Persona`, en el ejemplo).

5. Modifica `Personas` de modo que lleve la cuenta de cuántos objetos de tipo `Persona` se han creado.

```
class Persona {  
    private String nombre;  
    private int edad;  
  
    private static int counter;  
  
    /* constructor */  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
        counter++;  
    }  
  
    /* métodos para acceder a nombre y edad */  
    public String getNombre() {  
        return nombre;  
    }  
  
    public int getEdad() {  
        return edad;  
    }  
  
    public int getCounter() {  
        return counter;  
    }  
}
```

Para llevar la cuenta de cuántos objetos se crean, usamos una variable `static`. Las variables estáticas son comunes a todos los objetos de la clase. Es decir, cuando una variable es `static` (como `counter`), aunque creemos dos millones de objetos `Persona`, sólo habrá una variable `counter` que todos comparten. Si modifico `counter` en uno de los objetos, se refleja el cambio en todos.

Para llevar la cuenta, aumentamos `counter` en el constructor, porque el constructor es el método que se llama cada vez que se crea un nuevo objeto: no hay escapatoria, Java tiene que ejecutar el código del constructor que se llama con `new`.

Como hemos declarado constructor `private`, también hemos añadido un método `getCounter( )` para que `Test` pueda acceder a `Counter`.

```

class Test {
    public static void main(String[] args) {
        Persona p1 = new Persona("Otto", 21);
        Persona p2 = new Persona("Fritz", 20);
        Persona p3 = new Persona("Hans", 19);
        Persona p4 = new Persona("Peter", 7);

        System.out.println(p1.getNombre() + ", " + p1.getEdad() );
        System.out.println(p2.getNombre() + ", " + p2.getEdad() );
        System.out.println(p3.getNombre() + ", " + p3.getEdad() );
        System.out.println(p4.getNombre() + ", " + p4.getEdad() );

        /* estas cuatro líneas imprimen exactamente lo mismo */
        System.out.println("No. de objetos:" + p1.getCounter());
        System.out.println("No. de objetos:" + p2.getCounter());
        System.out.println("No. de objetos:" + p3.getCounter());
        System.out.println("No. de objetos:" + p4.getCounter());
    }
}

```

6. ¿Podemos definir el método `getCounter` como `static`? ¿Para qué querríamos hacerlo?

La clase `Persona` cambiaría así:

```

    public static int getCounter() {
        return counter;
    }

```

y `Test` quedaría así:

```

class Test {
    public static void main(String[] args) {

        System.out.println("No. de objetos:" + Persona.getCounter());

        Persona p1 = new Persona("Otto", 21);
        Persona p2 = new Persona("Fritz", 20);
        Persona p3 = new Persona("Hans", 19);
        Persona p4 = new Persona("Peter", 7);

        System.out.println(p1.getNombre() + ", " + p1.getEdad() );
        System.out.println(p2.getNombre() + ", " + p2.getEdad() );
        System.out.println(p3.getNombre() + ", " + p3.getEdad() );
        System.out.println(p4.getNombre() + ", " + p4.getEdad() );

        System.out.println("No. de objetos:" + Persona.getCounter());
    }
}

```

Para que un método pueda ser declarado estático, a) no debe acceder a variables de la clase que no sean estáticas (pero sí puede crear variables dentro de su bloque de código); y, b) sólo puede llamar a métodos estáticos. En ese sentido, `getCounter()` cumple los requisitos, pues sólo accede a `counter`, que es una variable estática.

Nos interesa declarar `getCounter` como `static` porque refleja mejor lo que hace: es un método que es común a todos los objetos de la clase. Además eso nos permite utilizar la sintaxis `Persona.getCounter()` en vez de `p1.getCounter()` (o `p2`, `p3`, etc.), cosa que, nuevamente, refleja mejor que es un método de toda la clase.

La primera línea de main imprime el contenido de `getCounter()` aún antes de que se haya creado ningún objeto de tipo `Persona`. Esto es para demostrar que los métodos y variables estáticos existen antes de que se cree ningún objeto (por eso, salvo que creen nuevos objetos, sólo pueden usar otros métodos y variables estáticos, que son los únicos que con certeza existen).

Este es el motivo por el que `main` es un método estático: tiene que haber un método al que pueda llamar antes del big bang, cuando ningún objeto ha sido aún creado.

9. Añade un constructor adicional a la clase `Persona`, que sirva en los casos que no conocemos la edad de la persona. Reescriba `Test` de modo que pruebe cada uno de los constructores.

```
class Persona {
    private String nombre;
    private int edad;
    private static int counter;

    /* constructor */
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        counter++;
    }

    /* otro constructor */
    public Persona(String nombre) {
        this.nombre = nombre;
        this.edad = -1;
        counter++;
    }

    /* métodos para acceder a nombre y edad */
    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }

    public static int getCounter() {
        return counter;
    }
}

class Test {
    public static void main(String[] args) {
        /* usamos un constructor distinto para cada objeto */
        Persona p1 = new Persona("Otto", 21);
        Persona p2 = new Persona("Adolf");

        /* imprimimos el contenido de los objetos */

        if( p1.getEdad() != -1) {
            System.out.println(p1.getNombre() +
                               ", " + p1.getEdad());
        } else {
            System.out.println(p1.getNombre() +
                               ", edad desconocida.");
        }
    }
}
```

```

        if( p1.getEdad() != -1) {
            System.out.println(p1.getNombre() +
                               ", " + p1.getEdad());
        } else {
            System.out.println(p1.getNombre() +
                               ", edad desconocida.");
        }
    }
}

```

Ahora hay definidos dos constructores, que tienen el mismo nombre, pero que son métodos distintos porque tienen parámetros distintos `Persona(String, int)` y `Persona(String)`. Basta que el tipo de los parámetros, su número o su orden sean distintos para que Java los considere métodos distintos. Intencionalmente se han omitido en la línea anterior los nombres de las variables usadas como parámetros para que se vea que el nombre de la variable no tiene nada que ver en el asunto: es el **tipo** de variable el que importa.

En el segundo constructor, `Persona(String nombre)`, no sabemos la edad de la persona, de modo que arbitrariamente establecemos que en ese caso se guardará -1 en edad (para indicar que es desconocida). Luego, en el momento de imprimir, comparamos edad con -1 para saber si está definida o no en uno de los objetos en particular, e imprimimos el mensaje adecuado.

10. Me parece muy interesante lo que dices pero, ¿cómo es que Test sabe que `Persona(String nombre)` guarda un -1 en edad? ¿Qué pasa si mañana Persona decide cambiar esta convención y prefiere guardar 0 en edad cuando la edad está indefinida?

De acuerdo, hice una pequeña trampa. Aunque el código del n. 9 compila y el programa funciona como se espera, Test no debe dar cosas por supuestas. Además, tener que comparar en Test cada vez antes que imprimir es por lo menos tedioso.

La solución correcta es que Persona se encargue de manejar sus propios problemas. Escribe un método `public String toString()` para Persona que devuelva siempre la representación correcta del contenido de Persona, y reescribe Test para que use este método.

```

class Persona {
    private String nombre;
    private int edad;
    private static int counter;

    /* constructor */
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
        counter++;
    }

    /* otro constructor */
    public Persona(String nombre) {
        this.nombre = nombre;
        this.edad = -1;
        counter++;
    }

    /*
     * no se ha vuelto a copiar getNombre, getEdad ni
     * getCounter, para no ocupar tanto espacio.
     */

    public String toString() {
        String msg = nombre;
        if (edad != -1) {

```

```

        msg += ", " + edad;
    } else {
        msg += ", edad desconocida.";
    }
    return msg;
}
}

class Test {
    public static void main(String[] args) {
        /* usamos un constructor distinto para cada objeto */
        Persona p1 = new Persona("Otto", 21);
        Persona p2 = new Persona("Adolf");

        /* imprimimos el contenido de los objetos */
        System.out.println(p1);
        System.out.println(p2);
    }
}

```

El método `toString()` está definido en `Object`, la clase madre de la que descienden todas las demás clases. Muchos métodos de la librería de Java, cuando se les pasa como parámetro un objeto, llaman automáticamente al método `toString()` de ese objeto. Es lo que hemos aprovechado al escribir `System.out.println(p1)` en vez de `System.out.println(p1.toString())`.

Hemos reemplazado (*override*) `toString()` en `Persona` de modo que se encargue de ver si la edad está definida o no. `Test` no tiene que preocuparse de eso.

**Es importante notar que `toString()` no necesita imprimir nada a la pantalla usando `System.out.println()`.** `toString()` devuelve un `String`, usando `return`.

11. La verdad que me aburre tener que escribir `p1`, `p2`, `p3`, etc., y luego usar uno a uno `println` para imprimir el valor de cada objeto. ¿No hay un modo automático de hacerlo?

Sí, a mí también me aburre. Arreglemos el problema, al fin y al cabo es un curso de programación, ¿no? Sólo modificaremos la clase `Test`, `Persona` sigue siendo la misma del ejemplo anterior.

```

class Test {
    public static void main(String[] args) {
        String[] nombres = { "Otto", "Fritz", "Hans", "Peter" };
        int[] edades = { 21, 20, 0, 7 };

        Persona[] personas = new Persona[nombres.length];

        for (int i = 0; i < nombres.length; i++) {
            if( edades[i] != 0)
                personas[i] = new Persona(nombres[i], edades[i]);
            else
                personas[i] = new Persona(nombres[i]);
        }

        System.out.println("Hemos creado " + Persona.getCounter()
            + " objetos.");

        /* imprimimos */
        for (i=0; i<personas.length; i++) {
            System.out.println(personas[i]);
        }
    }
}

```

}

Hemos definido dos arreglos: `personas` y `edades`. La notación `String[] nombres`, con los corchetes, quiere decir que `nombres` no es un `String`, sino un arreglo de `Strings`. (y `edades` un arreglo de `int`).

Tanto para `nombres` como `edades` estamos especificando los elementos del arreglo directamente: `{ "Otto", "Fritz", "Hans", "Peter" }`, de modo que a partir de esa línea empiezan a existir en el programa `nombres[0]` (de valor "Otto"), `nombres[1]` ("Fritz"), etc.

Definimos un arreglo `personas` que almacenará objetos de tipo `Persona`. Nuevamente hemos usado la notación de corchetes: `Persona[] personas`. Sin embargo, esta vez no especificamos los elementos del arreglo, sino que sólo reservamos espacio. Eso es lo que quiere decir `new Persona[ nombres.length ]`. No estamos llamando al constructor de `Persona`, sino creando un arreglo de elementos de tipo `Persona`, por ahora vacíos. `nombres.length` es el número de elementos del arreglo `nombres`, y es el espacio que reservaremos para las `personas`.

En el arreglo `edades` hemos introducido otra convención, totalmente arbitraria: indicaremos con 0 (cero) que la edad es desconocida, de modo que en loop podamos comparar la edad y saber qué constructor de `Persona` llamar. ¿Por qué no hemos usado -1, como en `Persona`? Sólo para que quede claro que se supone que no sé que `Persona` ha usado -1, y que mi código debe funcionar independientemente de cómo esté implementada `Persona`.

12. ¿No hay una versión más moderna para el segundo `for` del ejemplo anterior?

Sí. Se puede escribir:

```
for( Persona p : personas ) {  
    System.out.println(p);  
}
```

s Se lee: este loop itera el arreglo `personas`, y en cada iteración almacena el siguiente elemento del arreglo en la variable `p`, que es de tipo `Persona`. Esta notación se puede usar cuando dentro del loop no necesitamos el índice `i` del `for`.

13. Te vas a dormir después del parcial de PAV. En la madrugada te levantas sobresaltado, con escalofríos. Has tenido una pesadilla. La única cosa que te viene a la mente es la siguiente línea de Java:

```
public void consumoConsolidado( InputStream[] maquinas, String filename)
```

Simplemente para ver si estamos entendiéndonos en esta entretenida conversación, ¿serías capaz de llamar el método `toString()` para los elementos de `maquinas`?

```
public void consumoConsolidado( InputStream[] maquinas, String filename) {  
    for (int i=0; i < maquinas.length; i++) {  
        System.out.println(maquinas[i]);  
    }  
}
```

Es importante darse cuenta que no me interesa qué es exactamente un `InputStream`. No necesito saber nada en absoluto de *streams* para hacer un loop que itere los elementos de un arreglo. Me basta saber que `maquinas` es un arreglo, y eso lo sé porque `InputStream[]` tiene corchetes. Tampoco necesito que me digan cuántos elementos tiene `maquinas`. Eso lo puedo obtener del atributo `length` del arreglo.

14. Pasemos a otro ejemplo más entretenido. Estás desarrollando un juego para celulares llamado Zork. En el juego, el jugador puede ingresar en una línea comandos de una, dos o tres palabras. Te piden que diseñes una clase `Parser` que separe las palabras en comandos, y otra clase `Command` que



almacene los comandos.

¿Cómo enfocamos el problema? Primero, veamos cómo **no** enfocamos el problema.

**No** empezamos escribiendo código que lea la línea (`while ((linea = in.readLine()) != null)`... ¿alguien me lo ha pedido? **Tampoco** tratamos de acordarnos, por lo menos en el primer intento, de cómo estaba escrito la clase `Parser` de la tarea pasada. **Tampoco** trato de hacer un array que tenga las palabras que son válidas o no: no me piden eso.

Lo primero, definamos las clases que nos piden:

`Command`: lo que sé es que un comando puede tener hasta tres palabras. Empecemos con eso, y un constructor que inicialice las tres palabras. Ya después veremos si necesitamos *getters* o *setters*.

```
public class Command {
    private String firstWord;
    private String secondWord;
    private String thirdWord;

    /* constructor */
    public Command(String firstWord, String secondWord,
                   String thirdWord ) {

        this.firstWord = firstWord;
        this.secondWord = secondWord;
        this.thirdWord = thirdWord;
    }
}
```

¿Cómo defino `Parser`, y cómo se relaciona con `Command`? Por lo que me dicen en el enunciado del problema, `Parser` se encarga de sacar los comandos de la línea, y `Command` de almacenarlos, podemos pensar en un método de `Parser` que devuelva un objeto `Command` con las palabras.

Esto puede parecer arbitrario. ¿Por qué así? Bueno, quizá hay otra manera de hacerlo. Si es coherente y funciona, adelante. A veces los problemas tienen más de una solución.

Un primer bosquejo de `Parser` podría ser así:

```
class Parser {

    public Command parse(String line) {
        String primera, segunda, tercera;

        /* por completar */

        return new Command( primera, segunda, tercera);
    }
}
```

`parse` será el método principal de `Parser`, recibe la línea como parámetro, y la devuelve partida en un objeto `Command`.

Si escribes estas dos clases en el compilador, el programa compila correctamente, aunque todavía no hace nada. Te sugiero que lo intentes.

Añadamos funcionalidad a `parse`. Vamos a usar el método `String[] split(String`

expr) de la clase `String`, que parte un `String` usando `expr` como criterio de división, y devuelve un arreglo de `String` con los elementos.

Es decir, si `String s = "Otto Fritz";` y `String[] elm = s.split(" ");` (un espacio en blanco entre las comillas), entonces a partir de ese punto del programa empiezan a existir `elm[0]` (que contiene "Otto") y `elm[1]` (que contiene "Fritz").

Algunos detalles que debo tener en cuenta:

- puede ser que me pasen una línea en blanco como parámetro. Para saber si la línea está en blanco, puedo usar `line.trim()` para quitarle los espacios del comienzo y final, y luego `.length()` para saber la longitud del `String`. Si es cero, entonces simplemente devuelvo `null`. Ojo que los arrays tienen una propiedad `length` (`elm.length`), en cambio los `String` tienen un método `length()` (`line.length()`).

```
if (line.trim().length() == 0 )
    return null;
```

- no sé cuántas palabras tiene la línea de comando que me pasan como parámetro del método `parse`. Por tanto, tengo que ver si se trata de una, dos o tres palabras. Eso lo puedo saber fijándome en cuántos elementos tiene el arreglo que devuelve `split()`:

```
String[] elm = line.split(" ");
if (elm.length == 1) {
    ...
} else if (elm.length == 2) {
    ...
} else if (elm.length == 3) {
    ...
} else {
    /* no sé qué hacer con más de
       3 palabras */
}
```

- en cada uno de los `if/else if` del punto anterior, tengo que crear un objeto `Command` con las palabras en cuestión como parámetros. Si no existe una de las palabras, paso `null` como parámetro para esa palabra (eso es totalmente arbitrario: es una decisión que estamos tomando ahora). Pasar `null` es decirle al compilador que esa variable no almacena ningún objeto.

Completamos el método `parse`, y ya tenemos la clases `Parser` y `Command` completas:

```
public class Parser {
    public Command parse(String line) {
        String primera, segunda, tercera;

        if (line.trim().length() == 0)
            return null;

        String[] elm = line.split(" ");
        if (elm.length == 1) {
            primera = elm[0];
            segunda = null; tercera = null;
        } else if (elm.length == 2) {
            primera = elm[0];
            segunda = elm[1];
            tercera = null;
        } else if (elm.length == 3) {
            primera = elm[0];
```

```

        segunda = elm[1];
        tercera = elm[2];
    } else {
        /* no sé qué hacer con la cuarta palabra */
        return null;
    }

    /* devuelvo el nuevo objeto */
    return new Command( primera, segunda, tercera);
}

}

public class Command {
    private String firstWord;
    private String secondWord;
    private String thirdWord;

    /* constructor */
    public Command(String firstWord, String secondWord,
                   String thirdWord ) {

        this.firstWord = firstWord;
        this.secondWord = secondWord;
        this.thirdWord = thirdWord;
    }
}

```

15. Bien, pero Command, tal como está, me sirve de poco. Me gustaría que tuviera unos métodos `hasFirstWord( )`, `hasSecondWord( )`, `hasThirdWord( )` que devuelvan true si Command tiene una primera, segunda y tercera palabra, respectivamente, y false si no; y también métodos `getFirstWord( )`, `getSecondWord( )` y `getThirdWord( )` que me permitan obtener la palabra correspondiente.

Primero, veamos cómo definiríamos cada método:

- parece lógico que `hasFirstWord`, `hasSecondWord` y `hasThirdWord` devuelvan un dato de tipo boolean (true/false). Para saber si Command tiene una palabra, comparamos con null, pues hemos visto que Parse usa null el al llamar al constructor de Command para indicar que no hay palabra.
- `getFirstWord`, `getSecordWord` y `getThirdWord` devuelven la palabra correspondiente, que es de tipo String.

La clase Command quedaría así:

```

public class Command {
    private String firstWord;
    private String secondWord;
    private String thirdWord;

    /* constructor */
    public Command(String firstWord, String secondWord,
                   String thirdWord ) {

        this.firstWord = firstWord;
        this.secondWord = secondWord;
        this.thirdWord = thirdWord;
    }

    public boolean hasFirstWord( ) {
        if(firstWord != null)
            return true;
    }
}

```

```

        else
            return false;
    }

    public boolean hasSecondWord( ) {
        if(secondWord != null)
            return true;
        else
            return false;
    }

    public boolean hasThirdWord( ) {
        if(thirdWord != null)
            return true;
        else
            return false;
    }

    public String getFirstWord( )      { return firstWord; }
    public String getSecondWord( )    { return secondWord; }
    public String getThirdWord( )     { return thirdWord; }
}

```

16. Hay algo que no me gusta de cómo ha quedado Parser. ¿No podríamos ponerle al método la misma objeción que pusimos en el n.10? Es decir, ¿por qué es Parser el que decide pasar null a Command para indicarle que una de las palabras no existe? ¿Qué pasaría, por ejemplo, si otra clase distinta de Parser usara Command, y tuviera otra convención distinta?

De acuerdo, mentí otra vez. En realidad, es a Command a quien le toca decidir cómo representa internamente las palabras que no existen. Para solucionar este problema, podríamos definir dos constructores más para Command: uno de una sola palabra, y otro de dos palabras. En Parse usaríamos estos constructores, y nos olvidaríamos de pasar null si la palabra no existe.

El código quedaría así:

```

public class Command {
    private String firstWord;
    private String secondWord;
    private String thirdWord;

    /* constructor de 3 palabras */
    public Command(String firstWord, String secondWord,
                   String thirdWord ) {
        this.firstWord = firstWord;
        this.secondWord = secondWord;
        this.thirdWord = thirdWord;
    }

    /* constructor de 2 palabras */
    public Command(String firstWord, String secondWord) {
        Command(firstWord, secondWord, null);
    }

    /* constructor de 1 palabra */
    public Command(String firstWord) {
        Command(firstWord, null, null);
    }
}

```

```

    public boolean hasFirstWord( ) {
        if(firstWord != null)
            return true;
        else
            return false;
    }

    public boolean hasSecordWord( ) {
        if(secondWord != null)
            return true;
        else
            return false;
    }

    public boolean hasThirddWord( ) {
        if(thirdWord != null)
            return true;
        else
            return false;
    }

    public String getFirstWord( )      { return firstWord; }
    public String getSecondWord( )    { return secondWord; }
    public String getThirdWord( )     { return thirdWord; }
}

```

Como se ve, hemos optado por escribir los dos constructores nuevos de Command usando el constructor más general, el de tres parámetros. Así el código es más claro y tengo que escribir menos.

Parse quedaría así:

```

public class Parser {
    public Command parse(String line) {

        if (line.trim().length() == 0)
            return null;

        String[] elm = line.split(" ");
        if (elm.length == 1)
            return Command(elm[0]);
        else if (elm.length == 2)
            return Command(elm[0], elm[1]);
        else if (elm.length == 3)
            return Command(elm[0], elm[1], elm[2]);
        else {
            /* no sé qué hacer con la cuarta palabra */
            return null;
        }

    }
}

```

Después de estos arreglos, las dos clases han quedado limpias y compactas, y el código es mucho más claro.

17. Escribe un método `copyStream` que lea un stream de bytes, y lo copie a otro stream de bytes.

Ya podría habernos dicho el enunciado del problema qué parámetros espera copiaStream. Pero no lo hace. Veo dos caminos en el horizonte... uno complicado, que define un copiaStream que recibe como parámetros dos nombres de archivos; otro, el camino sencillo, que recibe como parámetros dos streams: uno de salida y otro de entrada.

Como suele ser más fácil pensar en términos de archivos en vez de en términos de streams, suele pasar que la gente escoge, sin saberlo, el camino difícil. Craso error. Veamos cómo sería con streams.

Las clases base que trabajan con bytes son InputStream y OutputStream, para entrada y salida. Como no nos dicen qué tipo de streams son exactamente, podemos usar las clases base como tipo del parámetro (porque así como todos los perros son animales, y a un método que espera por parámetro un animal le puedo pasar un objeto perro, un FileInputStream es de tipo InputStream, de modo que si mi parámetro es de tipo InputStream, el método funcionará con cualquier descendiente de InputStream).

```
public void copiaStream( InputStream origen, OutputStream destino) {  
  
}
```

Como se ve, ya que no me dicen nada, he decidido arbitrariamente que el método no devolverá ningún valor. Por tanto lo defino como void.

Todos los InputStream tienen un método int read( ), que devuelve -1 si se acabó el stream y genera una IOException si hay un error de otro tipo; y todos los OutputStream un método void write(int b), que genera una IOException si hay algún error (por ejemplo, tratar de escribir a un stream que no está abierto). Además hay que importar java.io.\* en algún sitio, que es donde están definidos InputStream y OutputStream.

```
public void copiaStream( InputStream origen, OutputStream destino) {  
    int b;  
  
    /* leo un byte de la entrada, lo escribo a la salida */  
    while ( ( b = origen.read()) != -1 ) {  
        destino.write( b );  
    }  
}
```

Como read( ) y write( ) pueden generar excepciones, hay que manejarlas, de lo contrario el código no compilará. Podemos definir un bloque try/catch, o declarar que copiaStream arroja una excepción (pasando así la excepción al método que nos llamó.

Con bloque try/catch:

```
public void copiaStream( InputStream origen, OutputStream destino) {  
    int b;  
  
    /* leo un byte de la entrada, lo escribo a la salida */  
    try {  
        while ( ( b = origen.read()) != -1 ) {  
            destino.write( b );  
        }  
    } catch (IOException e) {  
        System.err.println("Ha ocurrido un error: " + e.getMessage());  
    }  
}
```

Declarando que el método arroja una excepción:

```

public void copiaStream( InputStream origen, OutputStream destino)
throws IOException {
    int b;

    /* leo un byte de la entrada, lo escribo a la salida */
    while ( ( b = origen.read()) != -1 ) {
        destino.write( b );
    }
}

```

18. Envuelve el método del n. 17 en una clase Copier, fíjate si se puede declarar copiaStream como método estático; escribe otra clase Test que pida en la consola el nombre de dos archivos, uno para la entrada y otro para la salida, e intenta copiarlos (ojo con el nombre del archivo de salida, no vayas a borrar tu programa por error).

```

import java.io.*;
public class Copier {

    public static void copiaStream(
        InputStream origen, OutputStream destino)
    throws IOException {
        int b;

        /* leo un byte de la entrada, lo escribo a la salida */
        while ( ( b = origen.read()) != -1 ) {
            destino.write( b );
        }
    }
}
-----
import java.io.*;
public class Test {

    /* lee una línea de la consola
     * Devuelve null si hay un error o la línea está vacía
     */
    public static String leeLinea() {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));

        String s;
        try {
            s = in.readLine();
        }
        catch (IOException e) {
            System.err.println("Ha ocurrido un error "
                + e.getMessage());
            return null;
        }
        if (s.trim().length() == 0)
            return null;
        else
            return s;
    }

    public static void main(String[] args) {
        FileInputStream in;
        FileOutputStream out;

        System.out.println("Archivo origen: ");
    }
}

```

```

String origen = leeLinea();

System.out.println("Archivo destino: ");
String destino = leeLinea();

// verificamos que haya ingresado algo realmente
if( origen != null && destino != null) {

    try {
        in = new FileInputStream(origen);
        out = new FileOutputStream(destino);
        Copier.copiaStream(in, out);
        in.close();
        out.close();

    } catch (FileNotFoundException e) {
        System.out.println("No encuentro uno de los
archivos: “
                                + e.getMessage());
    } catch (IOException e) {
        System.out.println("Hubo un error al copiar: “
                                + e.getMessage());
    } // end catch/try

} // end if
} // end main
} // end class

```

Fin Repaso 1