Hoja 3 de Ejercicios

Lenguaje Java: tipos primitivos y objetos, instrucciones básicas, asignación, condicionales, bucles, strings, arrays, entrada/salida simple, interfaz gráfica sencilla, clases y subclases, constructores, herencia, polimorfismo, visibilidad, clases abstractas y finales.

Inicio: Semana del 21 de Febrero.
Duración: 4 semanas.

RECOMENDACIONES: En general es buena idea *pensar, debatir y resolver cada ejercicio antes de compilarlo y ejecutarlo*, pero *después* de obtenido el resultado previsto, la explicación y, en su caso, la reparación del error, resulta muy conveniente ver el resulto real en la práctica.

1) ¿Por qué el siguiente fragmento de código Java no imprime `Dos y dos son 4`?

```
int x = 2;

System.out.println("Dos y dos son " + x + x );
```

Solución: Imprime `Dos y dos son 22` porque la asociatividad por la izquierda del operador `+` hace que primero se evalúe las subexpresión `"Dos y dos son " + x` que devuelve un `String` y que se concatena después con el valor de la segunda `x`. Para forzar a que se realice la suma de en primer lugar, se deben usar paréntesis de las siguiente forma:

```
System.out.println("Dos y dos son " + (x + x) );
```

2) En el siguiente fragmento de código Java se ha comentado la segunda línea porque daba un error de compilación ¿Qué error? Sin entender muy bien el error, se ha intentado arreglarlo con la segunda línea, pero seguía habiendo un error de compilación, y lo mismo con la tercera línea. Finalmente, se han conseguido dos asignaciones equivalentes ¡y correctas! Explica los errores, los intentos erróneos de corregirlos, las dos instrucciones correctas, y el resultado final, 9.

```
byte x = 3;
// error x =  2 + x;
// error x =  2 + (int) x;
// error x =  (byte) 2 + (int) x;
x =  (byte) (2 + (int) x);
x =  (byte) (2 + x);

System.out.println(x + 2);
```

Solución: La primera línea comentada daba un error porque el compilador sabe que el operador `+` no puede devolver un `byte`, y aunque puede convertir automáticamente su argumento de `byte` a `int` (sin pérdida de precisión) para sumarlo con el 2 (que es de tipo `int`), el resultado de la suma será otro `int`. El primer intento de arreglo no consigue nada porque, como acabamos de decir, el *casting* `(int) x` se hace automáticamente al compilar esa expresión `+`. Tampoco sirve de nada el *casting* de 2 a tipo `byte`, ya que el problema no está en los operandos sino en el resultado. Los operadores aritméticos solo pueden devuelven datos de los siguientes tipos: `int`, `long`, `float`, y `double`. La solución es hacer *casting* del resultado de la suma a `byte` antes asignárselo a la variable `x` de tipo `byte`. Las dos asignaciones con dicho *casting* son correctas, ya que el *casting* a es `int` innecesario en este caso. El resultado final es que se imprime `9` en la salida después de sumar 3 + 2 + 2 + 2.

3) ¿Qué errores de compilación puedes detectar en el siguiente fragmento de código Java (suponiendo que esas fuesen las únicas instrucciones y declaraciones en el programa principal)? Explícalos y corrígelos todos, e indica los valores que se imprimen finalmente en la salida. Ten en cuenta que el compilador no detectará todos los errores a la vez. Es muy probable que el compilador detecte primero dos o tres errores y, después de que corrijas esos errores, detecte otros dos o tres errores diferentes en líneas donde antes no detectaba ningún error. ¿A qué se debe este comportamiento?

```
int x = x + 1;
int y = 2 * x;
int y = y / 3;
int z;

System.out.println("x: " + x
                   + " y: " + y
                   + " z: " + z
                   + " w: " + w
                  );
```

Solución: Mencionamos los 4 errores en el orden en que los hemos detectado nosotros, aunque los compiladores pueden detectarlos en un orden diferente:

1. Al evaluar la expresión `x + 1`, la variable `x` aún no está inicializada, aún no tiene un valor que se pueda sumar con `1`. Quizá se quería hacer primero una inicialización a cero y luego la suma con `1`. En cambio, la inicialización de `y` con `2 * x`, sí que es correcta porque el compilador cree que, en ese punto, la variable `x` ya está inicializada previamente (o lo estará cuando se arregle el error en su inicialización).
2. La variable `y` está siendo declarada por duplicado, indicándose su tipo de datos `int` en las dos líneas. Quizá la segunda aparición de `y` pretendía ser una asignación, y para ello se debería eliminar la indicación del tipo de datos `int` de la variable, o quizá se pretendía declarar e inicializar otra variable, dándole un identificador distinto.
3. En la sentencia de impresión la variable `z` es conocida, de tipo `int`, pero sin valor asignado previamente.
4. En la sentencia de impresión la variable `w` es desconocida, ya que no ha sido declarada.

Los compiladores suelen realizar su trabajo en varias pasada sobre el programa, haciendo en cada una de ellas un tipo de comprobación progresivamente más avanzadas o complejas (desde un mero análisis lexicográfico hasta un nivel semántico más avanzado, pasando por varios niveles de verificación de la estructura sintáctica).

4) Corrige los errores de compilación que tenga este fragmento de programa Java y explica la diferencia entre los valores que aparecen en la salida al ejecutarlo.

```
float  a = 2;
float  b = 2.46;
float  c = 2.4618483617391846291736F;
double d = 2.461848361739284629174 6;
double e = 2.461848361739284629174 6F;

System.out.println("c: " + c + " d: " + d + " e: " + e);
```

Solución: Mencionamos el único error de compilación que hay en ese fragmento, y otra condiciones que podrías parecer erróneas pero no lo son:

1. En Java, `2.46` es de tipo `double`, y por tanto, la inicialización de `b` que es de tipo `float` (de menor precisión que `double`) se detecta como un error. En general, la conversión de una expresión de tipo `double` a tipo `float` se puede resolver con *casting* (dejando al programador la responsabilidad de si la pérdida de precisión es relevante o no). En cambio, como en este caso la expresión de tipo `double` a convertir es un literal, basta con añadirle la especificación de tipo, es decir, basta con poner `2.46F` en vez de `2.46` en la inicialización de `b`.

2. No hay ningún problema con inicializar la variable `a` con un `int` ni la variable `e` con un `float` ya que en ese caso el tipo de cada variable (`float` y `double`, respectivamente) tiene mayor precisión que el valor que se le asigna.
3. Tampoco hay problema en escribir un literal con demasiada precisión, tanta que no pueda ser almacenada ni siquiera en un `double`, como ocurre al inicializar `d`, y por eso el valor que se imprime en la salida para esa variable no tiene tantos decimales como pretendíamos darle en la inicialización.
4. Ni siquiera hay problema en poner la especificación de precisión `F` en un literal con demasiada precisión para almacenarse como `float`, tal y como ocurre en las inicializaciones de `c` y `e`. Ambas inicializadas con el mismo valor de tipo `float`, aunque como una es `float` y la otra `double` sus valores internos serán ligeramente diferentes.

Así pues, la salida real obtenida (y debe ser la misma en cualquier entorno Java) es:

```
c: 2.4618483 d: 2.4618483617392846 e: 2.461848258972168
```

donde el valor de `c` es diferente al de `e` porque es de tipo `float` en vez de `double`, y es diferente al de `d` porque además de ser `float` en vez de `double` fue inicializada con un valor diferente (con la `F`); y el valor de `d` es diferente al de `e` porque aun siendo ambas de tipo `double` fueron inicializadas con valores diferentes (sin y con la `F`, respectivamente).

5) Considerando las variables declaradas con los mismos tipos de datos que se declararon en el ejercicio anterior, ¿por qué, de las asignaciones siguientes, solo son correctas las que tienen una suma? Corrige las erróneas y explica las correctas.

```
a = c * d;
b = (float) b - e;
c = (float) d / e;
a = (float) d + a;
b = d * 2;
c = (float) 2.0 + b;
a = 2.0 - (float) b;
```

Solución: Lo de la suma es pura casualidad. Todos son problemas relativos a la necesidad de *casting* para convertir `double` en `float` y al hecho de que la prioridad del operador *casting* es superior a la de los operadores binarios. El código corregido y con los errores comentados es el siguiente:

```
a = (float) (c * d); // añadimos casting que es necesario
b = (float) (b - e); // con paréntesis para evitar prioridad casting
c = (float) (d / e); // con paréntesis para evitar prioridad casting
a = (float) d + a;
b = (float) d * 2;   // basta con el casting para d porque 2 es int
c = (float) 2.0 + b;
a = 2.0F - (float) b; // basta con especificación de tipo F para 2.0
```

6) Considera las variables declaradas a continuación, y supón que las seis primeras han sido asignadas ya con los valores correspondientes a los instantes de entrada y salida, respectivamente, de un vehículo un túnel de carretera. Escribe y comenta las instrucciones Java necesarias para calcular el número de segundos transcurridos entre ambos instantes y asignárselo a la variable declarada abajo con ese objetivo.

```
int horaEntrada, minutoEntrada, segundoEntrada;
int horaSalida, minutoSalida, segundoSalida;
long segundosTrascurridos = 0;
```

Solución: Aunque no se pedía el programa completo, mejor tenerlo así, por muchas razones, e incluso hacerlo con un método estático dentro de la clase principal.
Código en versión fácil: convertir a segundos y restar, teniendo en cuenta el paso de medianoche.

```
public class Ej6v2 {
    public static void main(String[] args) {
        int horaEntrada, minutoEntrada, segundoEntrada;
        int horaSalida, minutoSalida, segundoSalida;
        long segundosTrascurridos = 0;

        horaEntrada=23; minutoEntrada=58; segundoEntrada=50;
        horaSalida=0;  minutoSalida=3;  segundoSalida=5;
        System.out.printf("Entrada %02d:%02d:%02d\n",horaEntrada,minutoEntrada,segundoEntrada);
        System.out.printf("Salida  %02d:%02d:%02d\n", horaSalida,minutoSalida,segundoSalida);

        segundosTrascurridos = segDesdeHasta(horaEntrada, minutoEntrada, segundoEntrada,
                                             horaSalida,  minutoSalida,  segundoSalida);
        System.out.println("Segundos trascurridos = " + segundosTrascurridos);
    } // end main

    /**
     * Calcula el número de segundos trascurridos desde un instante de entrada dado
     * como hora, minuto y segundo hasta el instante de salida en el mismo formato.
     * Supone que el intervalo nunca será superior a 23h59m59s dado que no hay
     * indicadición de a que días corresponden la entrada y la salida, pero se trata
     * bien el caso de cambio de un día al siguiente.
     * @param horaEntrada La hora del instante de entrada como un int entre 0 y 23
     * @param minutoEntrada El minuto del instante de entrada como un int entre 0 y 59
     * @param segundoEntrada El segundo del instante de entrada como un int entre 0 y 59
     * @param horaSalida La hora del instante de entrada como un int entre 0 y 23
     * @param minutoSalida El minuto del instante de entrada como un int entre 0 y 59
     * @param segundoSalida El segundo del instante de entrada como un int entre 0 y 59
     * @return número de segundos trascurridos entre el instante de entrada y el de salida
     */
    static long segDesdeHasta(int horaEntrada, int minutoEntrada, int segundoEntrada,
                              int horaSalida, int minutoSalida, int segundoSalida) {
        // constantes básicas para conversion de horas a minutos y a segundos
        final int SEGS_MIN = 60;
        final int MINS_HORA = 60;
        final int HORAS_DIA = 24;
        // conversion de los instantes de entada y salida a instante segundos desde 00h00m00s
        long timeIn = ((horaEntrada * MINS_HORA) + minutoEntrada) * SEGS_MIN
                      + segundoEntrada;
        long timeOut = ((horaSalida * MINS_HORA) + minutoSalida) * SEGS_MIN
                       + segundoSalida;
        // OJO: ajuste de hora de salida (en segundos) al día siguiente, en caso de que
        // el periodo de entrada a salida incluya la media noche
        if (timeOut < timeIn) timeOut +=  SEGS_MIN * MINS_HORA * HORAS_DIA;

        return timeOut - timeIn;
    }
}
```

Otra solución: Código en versión manual, paso a paso, restando segundos trascurridos y restando minutos y horas, teniendo en cuenta el salto al minuto siguiente, y a la hora y al día siguientes.

```
public class Ej6v1 {
    public static void main(String[] args) {
        int horaEntrada, minutoEntrada, segundoEntrada;
        int horaSalida, minutoSalida, segundoSalida;
        long segundosTrascurridos = 0;

        horaEntrada=23; minutoEntrada=58; segundoEntrada=50;
        horaSalida=0;  minutoSalida=3;  segundoSalida=5;
        System.out.printf("Entrada %02d:%02d:%02d\n",horaEntrada,minutoEntrada,segundoEntrada);
        System.out.printf("Salida  %02d:%02d:%02d\n", horaSalida,minutoSalida,segundoSalida);

        segundosTrascurridos = segDesdeHasta(horaEntrada, minutoEntrada, segundoEntrada,
                                             horaSalida,  minutoSalida,  segundoSalida);
        System.out.println("Segundos trascurridos = " + segundosTrascurridos);
    } // end main

    /**
     * Calcula el número de segundos trascurridos desde un instante de entrada dado
     * como hora, minuto y segundo hasta el instante de salida en el mismo formato.
     * Supone que el intervalo nunca será superior a 23h59m59s dado que no hay
     * indicadición de a que días corresponden la entrada y la salida, pero se trata
     * bien el caso de cambio de un día al siguiente.
     * @param horaEntrada
     * @param minutoEntrada
     * @param segundoEntrada
     * @param horaSalida
     * @param minutoSalida
     * @param segundoSalida
     * @return número de segundos trascurridos entre el instante de entrada y el de salida
     */
    static long segDesdeHasta(int horaEntrada, int minutoEntrada, int segundoEntrada,
                              int horaSalida, int minutoSalida, int segundoSalida) {
        // constantes básicas para conversion de horas a minutos y a segundos
        final int SEGS_MIN = 60;
        final int MINS_HORA = 60;
        final int HORAS_DIA = 24;
        // Restar segundos trascurridos contando con el paso al minuto siguiente
        if (segundoSalida < segundoEntrada) {
            // hubo cambio de minuto, luego se cuenta los segundos de la
            // fracción del minuto siguiente,
            segundosTrascurridos = segundoSalida + (SEGS_MIN - segundoEntrada);
            // y a la vez se avanza ese minuto en la hora de entrada
            minutoEntrada = (minutoEntrada + 1) % MINS_HORA;
            // con cuidado de que avanzar un minuto puede conllevar avance de hora
            if (minutoEntrada == 0) { horaEntrada = (horaEntrada + 1) % HORAS_DIA; }
         } else {  // hubo cambio de minuto
              segundosTrascurridos = segundoSalida - segundoEntrada;
         }
        // Restar minutos trascurridos contando con el paso a la hora siguiente
        // y convirtiendo la difeerencia en minutos a segundos
        if (minutoSalida < minutoEntrada) {
              segundosTrascurridos += (minutoSalida + (MINS_HORA - minutoEntrada))
                                      * SEGS_MIN;
              horaEntrada = (horaEntrada + 1) % HORAS_DIA;
        } else {  // no hubo cambio de hora, basta con restar y convetir a segundos
              segundosTrascurridos += (minutoSalida - minutoEntrada) * SEGS_MIN;
        }
        // Restar horas trascurridas contando con el paso al día siguiente
        // y convirtiendo la diferencia en horas a minutos y de ahí a segundos
        if (horaSalida < horaEntrada) {
              segundosTrascurridos += (horaSalida + (HORAS_DIA - horaEntrada))
                                      * SEGS_MIN * MINS_HORA;
        } else { // no hubo cambio de día, basta con restar y convetir a segundos
              segundosTrascurridos += (horaSalida - horaEntrada)
                                      * SEGS_MIN * MINS_HORA;
       }
       return segundosTrascurridos;

    }
}
```

7) Completa el fragmento de programa del ejercicio anterior de forma que calcule la velocidad media en km/h del vehículo dentro del, conociendo la longitud de éste que se declara abajo, e imprima un mensaje en caso de que supere la velocidad máxima declarada a continuación.

```
final float LONG_TUNEL_EN_KM = 3.47F;
final int VELOCIDAD_MAX_EN_KMHORA = 100;
double velocidadVehiculo;
```

Solución: Aunque no se pedía el programa completo, mejor tenerlo así, por muchas razones, e incluso hacerlo con un método estático dentro de clase principal.

```
public class Ej7 {
  public static void main(String[] args) {
        int horaEntrada, minutoEntrada, segundoEntrada;
        int horaSalida, minutoSalida, segundoSalida;
        long segundosTrascurridos = 0;

        final float LONG_TUNEL_EN_KM = 3.47F;
        final int VELOCIDAD_MAX_EN_KMHORA = 100;
        double velocidadVehiculo;

        horaEntrada=23; minutoEntrada=58; segundoEntrada=50;
        horaSalida=0;  minutoSalida=3;  segundoSalida=5;
        System.out.printf("Entrada %02d:%02d:%02d\n",horaEntrada,minutoEntrada,segundoEntrada);
        System.out.printf("Salida  %02d:%02d:%02d\n", horaSalida,minutoSalida,segundoSalida);

        segundosTrascurridos = segDesdeHasta(horaEntrada, minutoEntrada, segundoEntrada,
                                             horaSalida,  minutoSalida,  segundoSalida);
        System.out.println("Segundos trascurridos = " + segundosTrascurridos);

       velocidadVehiculo = velocidadDeKmSegAKmH(LONG_TUNEL_EN_KM,segundosTrascurridos);
        System.out.print("velocidad: " + velocidadVehiculo);
        if (velocidadVehiculo > VELOCIDAD_MAX_EN_KMHORA) {
           System.out.println("   vehículo multado");
       }
    } // end main

    /**
     * Calcula la velocidad en km/h, a partir de kilometros recorridos en los segundos dados
     * @param km  distancia en kilometros
     * @param seg tiempo en segundos
     * @return velocidad calculada en km/h
     */
    static double velocidadDeKmSegAKmH(float km, long seg) {
       double segundos = seg; // para forzar division en coma flotante
       return km / ((segundos / SEGS_MIN) / MINS_HORA);
    }

    /**
     * Calcula el número de segundos trascurridos desde un instante de entrada dado
     * como hora, minuto y segundo hasta el instante de salida en el mismo formato.
     * Supone que el intervalo nunca será superior a 23h59m59s dado que no hay
     * indicadición de a que días corresponden la entrada y la salida, pero se trata
     * bien el caso de cambio de un día al siguiente.
     * ... Igual que antes
     */
    static long segDesdeHasta(int horaEntrada, int minutoEntrada, int segundoEntrada,
                              int horaSalida, int minutoSalida, int segundoSalida) {
          // igual que antes
     }
}
```

Debe tenerse en cuenta el detalle de forzar la aritmética en coma flotante ya que esta otra versión del cálculo de velocidad generaría demasiadas multas por velocidad `Infinity`:

```
static double velocidadInfinita(float km, long segundos) {
      return km / ((segundos / SEGS_MIN) / MINS_HORA);
}
```

8) ¿Has probado tu programa del ejercicio anterior con casos como los de abajo? Conviértelo en un programa completo en Java (si es que no lo has hecho ya), para ejecutarlos con estos casos y explica el resultado en cada uno de ellos.

```
// caso 1:
horaEntrada=11; minutoEntrada=59; segundoEntrada=42;
horaSalida=12;  minutoSalida=0;   segundoSalida=09;

// caso 2:
horaEntrada=08; minutoEntrada=58; segundoEntrada=07;
horaSalida=09;  minutoSalida=01;   segundoSalida=02;

// caso 3:
horaEntrada=23; minutoEntrada=59; segundoEntrada=42;
horaSalida=0;  minutoSalida=1;   segundoSalida=9;
```

Solución: Primero hay que corregir los errores de compilación, producidos por los números enteros fuera de rango, `08` y `09`, ya que el cero inicial indica que deberían estar en sistema octal (es decir, solo dígitos de 0 a 7); y después, la salida que se deber obtener es la siguiente:

```
Entrada: 11:59:42
Salida : 12:00:09
segundos: 27
velocidad: 462.6666704813639   vehículo multado
Entrada: 08:58:07
Salida : 09:01:02
segundos: 175
velocidad: 71.38285773141044
Entrada: 23:59:42
Salida : 00:01:09
segundos: 87
velocidad: 143.5862080804233   vehículo multado
```

9) Considera las variables declaradas a continuación, y supón que la primera han sido asignada ya con el valor correspondiente al número total de milisegundos de CPU que ha consumido la ejecución de todos los programas de un usuario. Escribe y comenta las instrucciones Java necesarias para: calcular el número de horas, minutos y segundos consumidos, asignárselos a las variables declaradas abajo con ese objetivo, e imprimr sus valores en la salida.

```
long milisegundosEjecutados = 0;
long horasEjecucion = 0, minutosEjecucion = 0, segundosEjecucion = 0;
```

Solución:

```
public class Ej9 {
   final static int MILISEGS_MIN = 60000;
   final static int MINS_HORA = 60;

   public static void main(String[] args) {
      long horasEjecucion = 0, minutosEjecucion = 0, segundosEjecucion = 0;
      long milisegundosEjecutados = 23641200;
      long milisegundosRestantes = milisegundosEjecutados;
      horasEjecucion = (long) (milisegundosRestantes / (MINS_HORA * MILISEGS_MIN));
      milisegundosRestantes -= horasEjecucion * MINS_HORA * MILISEGS_MIN;
      minutosEjecucion = (long) (milisegundosRestantes / MILISEGS_MIN);
      milisegundosRestantes -= minutosEjecucion * MILISEGS_MIN;
      segundosEjecucion = (long) (milisegundosRestantes / 1000);

      System.out.printf("Entrada: %02d:%02d:%02d\n",
                  horasEjecucion,minutosEjecucion,segundosEjecucion);
      }
}
```

10) Amplia el ejercicio anterior para calcular el importe total a facturar (con un 18% de IVA) por dicho consumo de CPU, teniendo en cuenta que se cobra a un coste fijo por la primera hora (o fracción), se cobra ½ del coste fijo por cada uno de los 2 periodos de ½ de hora siguientes (o fracción), se cobra un ¼ del coste fijo por cada uno de los 4 periodos de ¼ de hora siguientes (o fracción), se cobra ⅛ del coste por cada uno de los 8 periodos de ⅛ de hora siguientes (o fracción), y así sucesivamente.

```
final long COSTE_FIJO = 10;
double importeFactura;
```

Solución: Realmente los cálculos realizados en el ejercicio anterior no son de utilidad aquí ya que es preferible contar con una variable que almacene el número total de horas a cobrar, incluida la parte fraccionaria. La implementación con dos bucles sigue literalmente la descripción del enunciado (con la particularidad de que el bucle `for`, interno, tiene como condición adicional de terminación la misma condición de terminación del bucle `while`, externo).

```
public class Ej10 {
   final static int MILISEGS_MIN = 60000;
   final static int MINS_HORA = 60;
   final static int HORAS_DIA = 24;
   final static int COSTE_FIJO = 10;
   final static double IVA = 18.0;

   public static void main(String[] args) {
      long horasEjecucion = 0, minutosEjecucion = 0, segundosEjecucion = 0;
      long milisegundosEjecutados = 23641200;


      long milisegundosRestantes = milisegundosEjecutados;
      horasEjecucion = (long) (milisegundosRestantes / (MINS_HORA * MILISEGS_MIN));
      milisegundosRestantes -= horasEjecucion * MINS_HORA * MILISEGS_MIN;
      minutosEjecucion = (long) (milisegundosRestantes / MILISEGS_MIN);
      milisegundosRestantes -= minutosEjecucion * MILISEGS_MIN;
      segundosEjecucion = (long) (milisegundosRestantes / 1000);

      System.out.printf("Entrada: %02d:%02d:%02d\n",
                  horasEjecucion,minutosEjecucion,segundosEjecucion);

      // Convertimos el número de milisegundos a número de horas a cobrar
      double horas =  (double) milisegundosEjecutados
                    / ( (double) MINS_HORA * MILISEGS_MIN );
      System.out.printf("fraccion horas:%.4f\n", horas);

      double importeFactura;
      double horasSinCobrar = horas;
      double totalSinIVA =  0;

      int divisor = 1; // en cada paso del bucle se multipicara por 2
      while (horasSinCobrar > 0) {
            // se cobran de una en una, desde 1 hasta 'divisor' fracciones
            // de hora, cada una al precio de Coste Fijo / divisor,
            // a menos que se agote antes el valor de hora a cobrar
            for (int i=1; horasSinCobrar > 0 && i<=divisor; i++) {
                  totalSinIVA += (double) COSTE_FIJO / divisor;
                  horasSinCobrar -= 1.0 / divisor;
            }
            divisor *= 2;
      }

      importeFactura = totalSinIVA * (1 + (IVA / 100.0));
      System.out.printf("importe factura:%.2f\n", importeFactura);

      }
}
```

Debe tenerse en cuenta el detalle de forzar la aritmética en coma flotante al dividir el `COSTE_FIJO` por un `divisor` entero, o de lo contrario, acabaríamos con un total de facturación demasiado bajo (pruébalo).

11) ¿Por qué el siguiente fragmento de código Java no imprime ningún `true`? ¿Imprimiría algún `true` si inicializásemos `y` con 9 en vez de con 3? Tampoco. ¿Y que pasaría si intercambiásemos los nombres de la variables `x` e `y` en sus respectivas declaraciones en la versión inicial del fragmento de programa? ¡Tampoco!

```
short x = 9;
byte  y = 3;

System.out.println( x < y );
System.out.println( x = y );
System.out.println( x > y );
```

Solución: (a)A ver, por sorprendente que pudiera parecer, si vamos paso a paso podemos concluir que no se imprime ningún `true`. Primero, es obvio que `x`, que vale `9`, no es menor que `y`, que vale `3`; segundo, `x = y` no es una comparación sino una asignación, es decir que se cambia el valor de `x` a `3` y acto seguido se imprime dicho valor, que tampoco es `true` sino `3`; y tercero, aunque inicialmente `x` sí que era mayor que `y`, ahora es igual a `y` debido a la asignación anterior, así que por tanto se imprime `false` otra vez.
(b) Si inicializásemos `y` con 9 en vez de con 3, la primera comparación volvería a hacer que se imprimiera `false` ya que ambas variables valen lo mismo ahora, es decir, `x` no es menor que `y`. La segunda instrucción sigue siendo una asignación, y se imprime el valor asignado a `x`, es decir, en este caso `9`. Y la tercera instrucción vuelve a imprimir `false` porque ambas variable siguen teniendo idéntico valor. Es decir, que tampoco así se imprime ningún `true`.
(c) Y si intercambiásemos los nombres de la variables `x` e `y` en sus respectivas declaraciones en la versión inicial del fragmento de programa, podría pensarse que la primera instrucción obviamente debería imprimir `true`, dado que es obvio que ahora `x`, que vale `3`, claramente menor que `y`, que vale `9`, pero … no se imprime `true`, ni `true` ni nada porque ahora el programa daría un error de compilación y no podría llegar a ejecutarse. El error de compilación estaría en la sentencia de asignación `x = y` porque `x` sería de tipo `byte` (inicializada con `3`) e `y` sería de tipo `short` (inicializada con `9`), pero es un error intentar asignar directamente un `short` (2 bytes) a un `byte`. Así que, tampoco así se imprimiría ningún `true`. ¡Tampoco!

12) En el siguiente fragmento de código Java hemos corregido un error de compilación y ahora se ejecuta correctamente, pero se obtiene un resultado distinto al aparentemente esperable. ¿Por qué es falsa la segunda comparación y cierta la primera?

```
int n1 = 1234567890;
// float n2 = n1 + 0.3; Error de compilación ¿por qué?
float n2 = n1 + 0.3F;

System.out.println( n1 < n2 );
System.out.println( n1 < n1 + 0.3 );
```

Solución: Es obvio que `0.3` es de tipo `double` y, por tanto, su suma con `n1` no se puede asigngar directamente a un `float` como es `n2`; hacía falta un casting o convertir `0.3` en `float` añadiendo la `F` de especificación de precisión. La primera instrucción de salida imprime `false` porque aunque `n2` se inicializó a `n1` sumado con algo positivo (`0.3F`), la suma se hizo en aritmética de tipo float y el valor de `n1` ya estaba al límite de precisión en `float`; es decir, la fracción sumada a `n1` se perdió (por falta de precisión en `float`) y no quedó almacenada en `n2`; por lo tanto, `n1` no es menor que `n2` (sino igual) y se imprime `false`. En cambio, en la segunda instrucción de salida, lo que se suma a `n1` no es `0.3F` sino `0.3`, que sin la `F` es un `double`, y por tanto, la precisión de la aritmética permite, esta vez, sí, mantener esa pequeña fracción sumada con `n1` (que aunque siga estando cercano al límite de la precisión en aritmética `float`, se convierte en un `double` al sumarse con el `double 0.3`, y también se convierte en un `double` al compararse con la suma `n1 + 0.3`); así pues el resultado de la segunda comparación es `true`.

13) En el siguiente fragmento de código Java combinamos el tipo `char` (de 2 bytes) con dos tipos numéricos: `short` (de 2 bytes) e `int` (de 4 bytes), y hemos puesto todos los *castings* necesarios para que no haya errores de compilación, pero se obtiene un resultado distinto al aparentemente esperable. ¿Por qué la diferencia `s - i` no es cero, pero `c2 - c3` sí lo es?

```
char c1 = '\u8001';
short s = (short) c1;
char c2 = (char) s;
int i = c1;  // único casting no necesario ¿por qué?
char c3 = (char) i;

System.out.println( i - s );
System.out.println( c2 - c3 );
```

Solución: La primera instrucción de salida imprime 65536 como resultado de restar el `short s` del `int i`. Aunque ambas variables han sido inicializadas a partir del mismo valor de tipo `char` almacenado en `c1`, no reciben el mismo valor entero y por eso la diferencia no es cero. El `char c1` ocupa 2 bytes, lo mismo que el `short s`, pero es necesario utilizar *casting* para asignar `c1` a `s` debido que puede haber pérdida de precisión pues el char no usa bit de signo pero el `short` sí. Efectivamente, ocurre en este caso que el bit más significativo de `c1` vale 1, sin que ello signifique signo negativo en el `char`. En cambio, al convertirlo a `short` ese bit más significativo con valor 1 se interpreta como un signo negativo, y por tanto, `s` termina adquiriendo un valor negativo (concretamente, −32767 en este caso). Dado que el `int i` ocupa 4 bytes, no es necesario *casting* para asignarle el valor numérico de `c1` (que en este caso es 32769). Así pues, la resta de `i - s` resulta en 32769 − (−32767) = 65536 que es el valor que se imprime. Curiosamente, o no, aunque las variables `i` y `s` tienen valores distintos, dado que ambas proceden de asignarles el mismo `char c1`, resulta que al volverlas a convertir en tipo `char` asignándoselas (con su debido *casting*) a `c2` y `c3` respectivamente, se convierten de nuevo en el mismo valor de tipo `char`, es decir que `c2 == c3` es `true`, y por tanto, la segunda instrucción de impresión imprime `0`.

14) En el siguiente programa completo pretendíamos imprimir en una línea una frase cualquiera almacenada en el `String cadena` seguida de otra línea rellena de espacios justo hasta la posición de la primera aparición de `letraBuscada` donde aparecerá una `marcaDeLetraEncontrada`. Es decir, queríamos obtener la siguiente salida

```
Mi vaso de vino esta vacio.
   *
```

pero en realidad se obtiene esta otra

```
Mi vaso de vino esta vacio.
Mi *
```

Consulta en http://download.oracle.com/javase/1.4.2/docs/api/ las operaciones que se usan para manipular Strings en ese programa y corrígelo para que tenga el resultado que se pretende. Recuerda que una expresión regular como "." se empareja con cualquier carácter, o sea que los argumentos pasados a `replaceAll` no parecen ser incorrectos si se pretende obtener una copia de cadena con todos su caracteres convertidos en blancos.

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

        String cadena = "Mi vaso de vino esta vacio.";
        char letraBuscada = 'v';
        char marcaDeLetraEncontrada = '*';

        String copia = new String(cadena);
        copia.replaceAll("."," ");

        int pos = cadena.indexOf(letraBuscada);
        copia = copia.substring(0,pos) + marcaDeLetraEncontrada;

        System.out.println(cadena);
        System.out.println(copia);

    }
}
```

Solución: El error es fácil de corregir una vez detectado, aunque quizá no sea tan fácil detectarlo para un principiante en Java. La invocación del método `copia.replaceAll("."," ");` hace exactamente lo que queremos, pero no modifica el objeto sobre el que se realiza la invocación, sino que devuelve el resultado como un nuevo String. Así pues lo fundamental es cambiar el valor de `copia` con una sentencia de asignación, como por ejemplo:

```
copia = copia.replaceAll("."," ");
```

De hecho, de nada útil nos ha servido crear un nuevo `String` con una copia del valor de `cadena` para inicializar la variable `copia` en su declaración. Esa memoria, sobre la que trabajará el método `replaceAll`, sin modificarla, se quedará sin referencias al hacer la nueva sentencia de asignación sobre `copia`, y quedará a disposición del *garbage collector* de Java. Así pues, no estamos haciendo las cosas de manera muy eficiente. Habría sido mejor, sustituir estas dos líneas

```
String copia = new String(cadena);
copia.replaceAll("."," ");
```

por esta otra:

```
String copia = cadena.replaceAll("."," ");
```

15) Amplía el programa corregido en el ejercicio anterior para que aparezcan marcas debajo de todas las apariciones de la letra buscada, es decir, para que se obtenga la siguiente salida

```
Mi vaso de vino esta vacio.
   *       *         *
```

Pistas: Dado que no se sabe cuántas apariciones de la letra buscada habrá que marcar parece necesario algún tipo de bucle no indexado; además puede ser útil mirar si existen varias formas del método `indexOf` y alguna de ellas permite que no se busque siempre desde el principio de cadena.

Solución:

```
public class Ej15 {
        public static void main(String[] args) {
         String cadena = "vMi vaso de vino esta vacio.v";
         char letraBuscada = 'v';
         char marcaDeLetraEncontrada = '*';

         String copia = cadena.replaceAll("."," ");

         System.out.println(cadena);
         int pos = cadena.indexOf(letraBuscada);
         // repetir marcardo mientras se haya encontrado letraBuscada
         while (pos >= 0) {
                // descomponemos la copia de la cadena en tres partes
                // parte previa + marca de lettra + parte posterior (si procede)
                if (pos+1 < copia.length()) {
                   copia = copia.substring(0,pos)
                          + marcaDeLetraEncontrada
                          + copia.substring(pos+1);
                } else { // si la letra a sustituir esta en la ultima posición
                           // no hay que concatenar nada detras de la marca
                   copia = copia.substring(0,pos)
                          + marcaDeLetraEncontrada;
                }
            pos = cadena.indexOf(letraBuscada,pos+1);

         }
         System.out.println(copia);
    }
}
```

16) Modifica el programa obtenido en el ejercicio anterior para que no busque apariciones de un carácter sino de una subcadena de caracteres; es decir, si buscásemos `"va"` en vez `'v'`, se debería obtener la siguiente salida

```
Mi vaso de vino esta vacio.
   *                 *
```

Solución: Realmente solo tenemos que cambiar esta declaración

```
char letraBuscada = 'v';
```

por esta otra:

```
String cadenaBuscada = 'va';
```

Y por supuesto, cambiar las dos invocaciones de `indexOf` para que se hagan así:

```
pos = cadena.indexOf(cadenaBuscada,pos+1);
```

La razón de que el cambio sea tan simple es que el método `indexOf` que usamos antes ya esperaba recibir un `String` como primer argumento y aunque le pasábamos un `char`, Java nos hizo la conversión automáticamente de `char` a `String`, sin que nosotros nos diésemos cuenta.

17) Utiliza las operaciones básicas de lectura de ficheros de texto vistas en clase de teoría y amplia el programa del ejercicio anterior para que tome como primer argumento de la línea de comandos la subcadena a localizar y como segundo argumento el nombre de un fichero de texto cuyas líneas deberán leerse e imprimirse por pantalla seguidas cada una de ellas por las marcas que le correspondan según el programa del ejercicio anterior. Es decir, si el contenido del fichero base.txt es

```
Mi vaso de vino esta vacio.
Avanzo mas con un vaso de cerveza
¡Valiente vago! Se va de vacaciones.
```

la orden **`java ejercicio8 va base.txt`** debe producir la siguiente salida

```
Mi vaso de vino esta vacio.
   *                 *
Avanzo mas con un vaso de cerveza
 *                 *
¡Valiente vago! Se va de vacaciones.
          *        *     *
```

Solución: ***Solución***:

```
import java.io.*;
public class Ej17 {
 public static void main(String[] args) throws IOException {
  BufferedReader buffer = new BufferedReader(
                          new InputStreamReader(
                            new FileInputStream(args[1]) ) );
  String cadenaBuscada = args[0];
  char marcaDeLetraEncontrada = '*';
  String cadena;
  while ((cadena = buffer.readLine()) != null) {
      String copia = cadena.replaceAll("."," ");
      System.out.println(cadena);
      int pos = cadena.indexOf(cadenaBuscada);
      // repetir marcardo mientras se haya encontrado letraBuscada
      while (pos >= 0) {
        // descomponemos la copia de la cadena en tres partes
        // parte previa + marca de lettra + parte posterior (si procede)
        if (pos+1 < copia.length()) {
             copia = copia.substring(0,pos)
                    + marcaDeLetraEncontrada
                    + copia.substring(pos+1);
        } else { // si la letra a sustituir esta en la ultima posicion
                 // no hay que concatenar nada detras de la marca
             copia = copia.substring(0,pos)
                    + marcaDeLetraEncontrada;
        }
        pos = cadena.indexOf(cadenaBuscada,pos+1);
        }
        System.out.println(copia);
  }
  buffer.close();
 }
}
```

18) Para este ejercicio dispones de dos arrays que almacenan información correspondiente a diversos intervalos en un viaje. El elemento i-ésimo del array `minutos` indica cuántos minutos duró ese intervalo y el elemento i-ésimo del array `velocidad` indica la velocidad media registrada en ese intervalo. A partir de esa información, debes calcular la velocidad media global del viaje en km/hora, y lo has hecho mediante el siguiente programa, acumulando los kilómetros recorridos en cada intervalo y dividiendo ese resultado por el número total de minutos empleados en el viaje.

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

        int[] minutos =   { 4, 8,10, 6, 5,12, 2};  // minutos
        int[] velocidad = {30,50,90,80,70,50,20};  // km/hora

        double recorrido = 0.0;
        int    totalMinutos = 0;

        for (int i=0; i<minutos.length; i++) {
          totalMinutos += minutos[i];
          recorrido += minutos[i] * velocidad[i] / 60;  // Linea *1*
        }

        System.out.println(recorrido / totalMinutos / 60); // Linea *2*
    }
}
```

Sin embargo, el resultado que sale de ese programa es cercano 0.016312 km/hora, lo cual es obviamente incorrecto; por tanto, acudes a un amigo que te sugiere que en la expresión de la línea *2* podría ser mejor usar paréntesis para dividir primero `totalminutos` entre 60 y después hacer la otra división de la misma expresión. Hecho este cambio el resultado es `Infinity`. Sí, eso mismo, se imprime esa palabra. Sin pensártelo mucho vuelves a dejar la línea *2* como estaba y pruebas, *por si acaso*, poniendo los paréntesis en la expresión de la línea *1* para que se haga la división por 60 en primer lugar. El resultado ahora es cercano 0.0074468 km/hora, o sea menos aún que al principio. Así que vuelves a dejar el programa como estaba al principio, y después otro amigo te dice que lo de poner paréntesis en la línea *2* te lo habían dicho bien pero que además hay que dividir por 60.0 en vez de por 60 y con estos dos cambios en la línea *2* se obtiene una velocidad media global de aproximadamente 58.7234 km/hora que parece bastante más razonable que los otros valores.
Explica razonadamente los errores que has ido intentando corregir con todos esos cambios que has probado, y por favor, arregla de una vez el programa porque el profesor dice, a ojo, que no se cree que el resultado sea inferior a los 60 km/hora.

Solución: La línea *2* ya está correcta; ahora basta con hacer ese mismo cambio en la línea *1* ya que ahí también es importante que la división por 60 no se haga en aritmética entera sino en coma flotante, y para ello basta sustituir también ese `60` por `60.0` en la línea *1*. Los paréntesis en esta línea no tenían mucho sentido (a menos que pensásemos que la multiplicación de minutos por velocidad pudiese exceder el rango de `int`).

19) Escribe un programa Java que acepte un número impar como único argumento de la línea de comando y lo utilice como tamaño de diagonal para imprimir un rombo como el siguiente, que correspondería recibir 9 como argumento:

Solución:

```
public class Ej19 {
      public static void main(String[] args) {
              int x = Integer.parseInt(args[0]);

              int d = (x - 1) / 2; // media diagonal
              // vertice superior
              for (int i=1; i<=d; i++) {System.out.print(" ");}
              System.out.println("*");
              // mitad superior
              for (int i=d-1; i>=0; i--) {
                  for (int j=1; j<=i; j++) {System.out.print(" ");}
                  System.out.print("*");
                  for (int j=1; j<=(2*(d-i)-1); j++) {
                      System.out.print(" ");
                  }
                  System.out.println("*");
              }
              // mitad inferior, mismo cuerpo de for externo, pero
              // cambiado a orden ascendente y con una iteracion menos
              for (int i=1; i<=d-1; i++) {
                  for (int j=1; j<=i; j++) {System.out.print(" ");}
                  System.out.print("*");
                  for (int j=1; j<=(2*(d-i)-1); j++) {
                      System.out.print(" ");
                  }
                  System.out.println("*");
              }
              //vertice inferior
              for (int i=1; i<=d; i++) {System.out.print(" ");}
              System.out.println("*");

        } // end main
}
```

20) No es que el programa siguiente sea perfecto para detectar si el número entero que se pasa como primer argumento en la línea de comando es superior o no a cierto umbral, pero ¿de qué se puede quejar el compilador? Propón dos formas de hacer que compile, manteniendo su aparente lógica.

```
public class Umbrales {
    public static void main(String[] args) {
        final int UMBRAL = 9;
        int n = Integer.parseInt(args[0]);
        String resultado;

        if (n > UMBRAL) { resultado  = "mayor"; }
        if (n <= UMBRAL) { resultado  = "no mayor"; }
        System.out.println(resultado);

    }
}
```

Solución: El compilador se queja de que *posiblemente* la ejecución llegue a la instrucción de salida sin que se le haya asignado ningún valor a resultado. Nosotros sabemos que eso es imposible (¿verdad?) pero claro, el compilador no puede analizar la lógica de cualquier programa. Así que bastaría con inicializar la variable resultado, con un String vacío, o con algo así:

```
        String resultado = "resultado sin definir";
```

O bien, sin usar esa inicialización, el compilador también se quedaría contento juntamos las dos instrucciones condicionales en una sola, de la siguiente forma (lo cual sería hasta más lógico para nosotros, además de permitir al compilador saber que la ejecución siempre asignará un valor a `resultado` antes de llegar a la instrucción de salida):

```
        if (n > UMBRAL) { resultado  = "mayor"; }
        else            { resultado  = "no mayor"; }
        System.out.println(resultado);
```

21) Escribe un programa Java que adivine el número natural menor que 1000 que ha pensado el usuario, haciéndole una serie de preguntas sobre si es menor o mayor que un número candidato elegido provisionalmente por el programa. El programa puede usar búsqueda dicotómica en el intervalo de posibles valores, eligiendo provisionalmente el punto medio del intervalo como candidato para preguntarle al usuario si el número a adivinar es menor o mayor que ese candidato, y tras la respuesta del usuario ajustar consecuentemente los límites del intervalo de posibles valores. Utiliza `System.in.read()` para saber si el usuario pulsa la tecla `'<'` o `'>'` para indicar si su número es menor o mayor que el candidato presentado, o pulse `'='` si el candidato propuesto coincide con su número. Ten en cuenta que la entrada por consola (`in`) exige que el usuario pulse la tecla de retorno (*enter*) después de la tecla elegida, ya sea `<`, `>`, o `=`. Y eso genera caracteres adicionales que tu programa debe ignorar de la mejor forma posible, es decir, sin que el usuario lo note. Consulta en http://download.oracle.com/javase/1.4.2/docs/api/ para ver los detalles de `System.in`, a qué clase pertenece y qué otros métodos ofrece.

Solución:

```
public class Ej21 {
      public static void main(String[] args) {
            int min = 0;
            int max = 1000;
            int m = 0; // punto medio entre min y max
            char c = ' ';
            while (c != '=') {
                m = (min + max) / 2;
                System.out.println("Es tu numero < = > que " + m + "?");
                do {
                     c = (char) System.in.read();
                } while (c != '<' && c != '>' && c != '=');
                if (c == '<') { max = m; }
                else  if (c == '>') { min = m; }
            }
            System.out.println("found it!! " + m);

        } // end main
}
```

22) Cuando tengas el programa anterior funcionando correctamente escribe otra versión similar pero que utilice el método `showInputDialog` de `javax.swing.JOptionPane` para pedir al usuario que escriba `menor` o `mayor` en el campo de entrada, según su número sea menor o mayor que el candidato propuesto. Y utiliza el método `JOptionPane.showMessageDialog(null, mensaje)` para mostrar al usuario un `mensaje` informándole de que su número ha sido adivinado y en cuántos intentos se ha conseguido. Para facilitar la interacción con el usuario prueba a utilizar este método `int b = JOptionPane.showConfirmDialog(null, "¿Es su numero < " + n + "?")` que ofrece tres botones (Sí, No, y Cancelar) para que el usuario pulse uno de ellos y se puede saber cuál ha pulsado comparando `b` con `JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION` o bien `JOptionPane.YES_OPTION`. Intenta evitar que aparezca la opción Cancelar de esta forma `int b = JOptionPane.showConfirmDialog(null, "¿Es su < " + n + "?", "Elija su respuesta", JOptionPane.YES_NO_OPTION);`

Solución:

```
import javax.swing.JOptionPane;
public class Ej22  {
   public static void main(String[] args) {

          int min = 0;
          int max = 1000;
          int m = 0;  // punto medio entre min y max
          int intentos = 0;
          String n = "";
          while (! n.equals("igual") ) {
                m = (min + max) / 2;
                intentos++;
                do {
                       n = JOptionPane.showInputDialog(null,
                                          "Escriba menor mayor o igual según\n"
                                          + "sea su numero respecto de " + m + "?");
                } while ( ! (n.equals("menor") || n.equals("mayor")
                                                || n.equals("igual")) );
                if (n == "menor") { max = m; }
                else  if (n == "mayor") { min = m; }
           }
           System.out.println("found it!! " + m + " en " + intentos + " intentos.");
          } // end main
}
```

Otra solución:

```
import javax.swing.JOptionPane;
public class Ej22  {
   public static void main(String[] args) {

          int min = 0;
          int max = 1000;
          int m = 0;  // punto medio entre min y max
          int intentos = 0;
          int r = JOptionPane.NO_OPTION;
          while ( r != JOptionPane.CANCEL_OPTION ) {
                m = (min + max) / 2;
                intentos++;
                r = JOptionPane.showConfirmDialog(null,
                       "Si su número es " + m + " lo he adivinado\n"
                              + "en " + intentos + " intentos. Pulse cancelar.\n\n"
                              + "En caso contrario,\n"
                              + "dime si tu número es mayor que ese.");
                if (r == JOptionPane.YES_OPTION) { min = m; }
                else if (r == JOptionPane.NO_OPTION) { max = m; }
           }
     } // end main
}
```

23) Escribe un programa que tome dos nombres de ficheros como argumentos de la línea de comandos y genere un archivo con el segundo nombre y cuyo contenido sea el resultado de haber invertido línea a línea y palabra a palabra el contenido del archivo nombrado como primer argumento, donde entendemos por palabra cualquier cadena separada por uno o mas blancos (cuyo número no es necesario respetar en la inversión). Es decir, si el primer archivo contiene

```
Mi vaso    de vino       esta    vacio.
Avanzo mas  con  un vaso     de cerveza
¡Valiente   vago!   Se va de vacaciones
```

el archivo generado debe contener lo siguente

```
vacaciones de va Se vago! ¡Valiente
cerveza de vaso un con mas Avanzo
vacio. esta vino de vaso Mi
```

Sugerencias: Almacena todo el archivo de entrada en memoria en un array de líneas donde cada línea se almacene como un array de Strings (palabras), y después procesa los arrays en orden inverso. Por el momento, puedes leer el fichero de entrada dos veces: la primera para contar líneas y poder crear el array de tamaño exacto y la segunda para procesar sus líneas una a una. Lo mismo puedes hacer con cada línea para saber cuántas palabras se almacenarán en cada array de Strings antes de crearlo.

Solución:

```
import java.io.*;
public class Ej23 {
 public static void main(String[] args) throws IOException {
  BufferedReader buffer = new BufferedReader(
                            new InputStreamReader(
                              new FileInputStream(args[0]) ) );
  String linea;
  int nl = 0;
  while (( linea = buffer.readLine()) != null) {
        nl++;
  }
  buffer.close();
  buffer = new BufferedReader(
                            new InputStreamReader(
                              new FileInputStream(args[0]) ) );
  while (( linea = buffer.readLine()) != null) {
        String[] palabras = linea.split(" +");
        for (int np = palabras.length - 1 ; np >= 0; np--) {
          System.out.print(palabras[np]);
          if (np > 0) { System.out.print(" "); }
        }
        System.out.println("");
  }
  buffer.close();
 }
}
```

24) Sin utilizar el compilador, encuentra y explica los 6 cambios que se deben hacer en este programa para que no produzca errores de compilación (pero sin llegar a cambiar por completo su estructura). Después de eso, utiliza el compilador para comprobar que él tampoco es capaz de detectar todos los errores en el primer intento. Arregla los que detecte y repite la compilación ¿Cuántas veces has tenido que ejecutar el compilador para eliminar todos los errores?

```
public class Errores1 {

   String saludo = "Hola mundo!;"

   public static void main(String[] args) {

      saluda("");

    }// main

    public static saluda(String saludoExtra);
    {
       System.println(saludo, saludoExtra);
    }

}// class
```

Solución: Una vez más el orden en que damos los errores puede no coincidir con el orden en que los detectará el compilador:

1. En la cabecera del método `Saluda` sobra el punto y como final, y eso hace que el compilador tampoco reconozca `saludoExtra` como variable declarada, por ser un parámetro del método, y se queje de ello en la sentencia de impresión, pero es todo un solo error.
2. También en la cabecera del método `Saluda` falta el tipo de dato que devuelve el método, o bien `void` en caso de que no devuelva ningún valor.
3. `Println` es un método de `out` que es un objeto de la clase `System`, luego debemos poner `System.out.println`(…).
4. En la misma instrucción hay otro error, porque no existe método `println` que tome como parámetros dos Strings; se debe cambiar al coma por un signo + para concatenarlos.
5. En la misma instrucción tampoco se permite el acceso a la variable `saludo` ya que la instrucción está dentro del método `Saluda` que es `static` pero la variable `saludo` no se ha declarado como `static`.
6. Vaya, parece que al declarar `saludo`, el punto y coma de finalización de la declaración se nos coló dentro del literal con el que inicializamos la variable.

En cuanto al número de ejecuciones del compilador, pues depende mucho del entorno porque muchos editores modernos detectan ya muchos errores antes de que compilemos el programa. Pero utilizando un editor de texto simple, sin ayudas para Java, y llamando a `javac` para compilar: en la primera ejecución se detectaron los errores 2 y 6. Una vez corregidos, en la segunda compilación, se detectaron el error (doble) indicado arriba como 1. En la tercera ejecución, se detectaron los errores 3 y 5. Y finalmente, en la cuarta compilación se detectó el error 4.

25) Este programa tiene un error de compilación que se puede arreglar de manera inmediata, pero que realmente se relaciona con otro error más grave: el programa no produce el resultado deseado. ¿Cuál es el error de compilación y cuál el otro error más grave? Corrígelos y prueba tu programa.

```
public class Temperaturas {
  public static void main(String[] args) {
    double fahrenheitTemp = 103.5;
    double celsiusTemp;

    aCelsius(celsiusTemp, fahrenheitTemp);
    System.out.println(fahrenheitTemp + "°F son "
                       + celsiusTemp + "°C");
  }// main

  public static void aCelsius(double c, double f) {
     c =  5.0 / 9.0 * (f - 32.0);
  }
}// class
```

Solución: El compilador se queja de que la variable `celsiusTemp` puede no haber sido inicializada cuando la ejecución llegue a la instrucción de salida. Lo cual podría parecernos absurdo porque pretendemos precisamente darle valor en la llamada al método `aCelsius`, pero lo curioso es que el compilador se queja también de que esa variable también llega sin haber sido inicializada al punto en que se invoca dicho método. Podríamos estar tentados de arreglarlo inicializando la variable en su declaración, con valor 0, por ejemplo, y eso dejaría al compilador contento. Ahora bien, el error es que siempre se imprimirían todas las temperaturas Celsius como `0.0°C`, porque la asignación del valor convertido dentro del método `aCelsius` a su parámetro `c` no afecta al valor de la variable `celsiusTemp` se pasó como *parámetro por valor*. Dicho de otra forma, el método de conversión debería ser una función que devolviese como resultado el valor de la temperatura convertida a Celsius, de la siguiente forma:

```
public class Temperaturas {
  public static void main(String[] args) {
    double fahrenheitTemp = 103.5;
    double celsiusTemp;

    celsiusTemp = aCelsius(fahrenheitTemp);
    System.out.println(fahrenheitTemp + "°F son "
                       + celsiusTemp + "°C");
  }// main

  public static double aCelsius(double f) {
     return  5.0 / 9.0 * (f - 32.0);
  }
}// class
```

26) Amplia el programa corregido en el ejercicio anterior, añadiéndole otro método para convertir de Celsius a Fahrenheit y prueba que aplicando las dos conversiones se obtiene el mismo valor de temperatura original. Repite un número muy alto de veces la conversión de una misma temperatura en las dos direcciones comprobando en cada repetición que seguimos teniendo un valor idéntico al de la temperatura original, o imprime un mensaje con la diferencia encontrada y el número de iteración en la que se ha encontrado.

Solución:

```
public class Ej26  {
   public static void main(String[] args) {
      double fahrenheitTemp = 103.3;
      double celsiusTemp;
      long iteraciones = 0;
      double originalTemp = fahrenheitTemp;

      while ((iteraciones >= 0) && (fahrenheitTemp == originalTemp)) {
          iteraciones++;

         celsiusTemp = aCelsius(fahrenheitTemp);
         System.out.println(fahrenheitTemp + "°F son " + celsiusTemp + "°C");

         fahrenheitTemp = aFahrenheit(celsiusTemp);
         System.out.println(fahrenheitTemp + "°F son " + celsiusTemp + "°C");

         if (fahrenheitTemp != originalTemp) {
                System.out.println(originalTemp + "°F pasó a ser "
                                   + fahrenheitTemp + "°F despues de "
                                   + iteraciones + " iteraciones.");
         }
    }
       System.out.println(iteraciones + " iteraciones.");

    }// main

    public static double aCelsius(double f) {
       return  5.0 / 9.0 * (f - 32.0);
    }

    public static double aFahrenheit(double c) {
       return  (9.0 / 5.0 * c) + 32.0;
    }

}// class
```

El programa se pensó para hacer un número de iteraciones muy grande (desde 1 hasta el máximo `long` positivo, que al sumarle 1 se convertirá en un número negativo), pero en realidad bastaba con una iteración para ver que muchos de los valores que representamos como exactos con uno o dos decimales (en nuestro sistema decimal) no se pueden representar exactamente en coma flotante en binario, y de ahí que incluso en la primera conversión tengamos este resultado:

```
103.3°F son 39.611111111111114°C
103.30000000000001°F son 39.611111111111114°C
103.3°F pasó a ser 103.30000000000001°F despues de 1 iteraciones.
1 iteraciones.
```

27) El siguiente fragmento Java define una clase con todos los atributos de libros que se van a manejar. Completa la clase con tres constructores públicos: uno para libros de los que solo conocemos título, autor y editorial que lo va a publicar; otro para cuando además conocemos el precio (y seguimos suponiendo aún no se ha publicado); y el tercero para cuando además conocemos el número de copias vendidas (y por lo tanto, ahora tenemos constancia de ya se ha publicado). Procura reutilizar en todo lo posible el código de los constructores que escribas.

```
public class Libro {

  private String titulo;
  private String autor;
  private String editorial;
  private double precio;
  private boolean sinPublicar;
  private int copiasVendidas;

}// class
```

Solución:

```
public class Libro  {

        private String titulo;
        private String autor;
        private String editorial;
        private double precio;
        private int copiasVendidas;
        private boolean sinPublicar;

        public Libro (String titulo, String autor, String editorial) {
                this.titulo = titulo;
                this.autor = autor;
                this.editorial = editorial;
                this.precio = 0.0;
                this.copiasVendidas = 0;
                this.sinPublicar = true;
        }

        public Libro (String titulo, String autor, String editorial,
                        double precio) {
            this(titulo,autor,editorial);
            this.precio = precio;
            this.copiasVendidas = 0;
            this.sinPublicar = true;
      }

        public Libro (String titulo, String autor, String editorial,
                        double precio, int copiasVendidas) {
            this(titulo,autor,editorial,precio);
            this.copiasVendidas = copiasVendidas;
            this.sinPublicar = false;
      }

}// class
```

28) Encuentra los tres errores de compilación en esta clase que manipula dos factores de escala al calcular áreas de figuras geométricas sencillas.

```
public class Escala {
  private int escalaX;
  private int escalaY;

  private Escala() {
     escalaX = 1;
     escalaY = 1;
  }
  public Escala(int x, int y) {
     escalaX = x;
     escalaY = y;
  }
  public Escala(int x) {
     this();
     this.escalaX = x;
  }
  public Escala(int y) {
     this(1,y);
     this.escalaY = y;
  }

  public int areaRectangulo(int base, int altura) {
     return escalaX * base * escalaY * altura;
  }

  public double areaRectangulo(int base, int altura) {
     return (double) (escalaX * base * escalaY * altura);
  }

  public double areaCuadrado(double lado) {
     this.areaRectangulo((int) lado, (int) lado);
  }
}// class
```

Solución: Como siempre, el orden en que se listan los errores no es relevante:

1. El segundo y el tercer constructor son idénticos respecto a su firma (número y tipo de parámetros), luego no es posible distinguir cuál de los dos se desea usar en una declaración como `Escala e = new Escala(3)`
2. Los dos métodos de nombre `areaRectangulo` esperan el mismo tipo de argumentos aunque cada uno devuelve un resultado de tipo diferente (`int` y `double`). La sobrecarga de métodos permite que métodos del mismo nombre tengan distintos tipos de datos en sus argumentos, pero si tienen mismo nombre y mismos tipos de argumentos, entonces el resultado debería ser el mismo, o de lo contrario el compilador no podría decidir sobre qué tipo de datos tiene una expresión como `areaRectangulo(1,2)`.Y de hecho, si además tuvieran el mismo tipo de dato de retorno, entonces se trataría de métodos duplicados, y sobraría uno.
3. El método `areaCuadrado(double lado)` pretende calcular el área del cuadrado reutilizando el método `areaRectangulo` y pasándole el `lado` convertido a `int` como base y como altura del rectángulo. Todo ello es correcto, asumiendo que hayamos resuelto el problema descrito en el punto 2. Y la invocación `this.areaRectangulo` también es correcta, pero el valor devuelto por dicha invocación no se devuelve como valor resultante del método `areaCuadrado`, es decir, falta una instrucción `return`.

29) Revisa bien el siguiente conjunto de clases Java por si hubiese algún error y corrígelo antes de contestar las 2 preguntas que figuran que figuran debajo de él.

```
public final class SubclaseFinal extends SubclaseIntermedia {
  public SubclaseFinal() {
    System.out.println("finalmente");
  }
  public SubclaseFinal(String e) {
    System.out.println(e);
  }

  public int raro(int x, int y) {return super.raro(x,x) + y;}
} // class SubclaseFinal

public class SubclaseIntermedia extends Abstracta {
  SubclaseIntermedia() {
    this("pero OK");
    System.out.println("y feliz");
  }
  private SubclaseIntermedia(String e) {
    System.out.println(e);
  }

  public int raro(int x, int y) {return x + y;}
} // class SubclaseIntermedia

abstract public class Abstracta {
  protected Abstracta() {
    System.out.println("clase abstracta");
  }

  abstract public int raro(int x, int y);
}  // class Abstracta
```

Pregunta 1: ¿Qué se imprime al hacer cada una de estas declaraciones de objetos? Relaciona los casos que son similares entre sí y explica por qué lo son.

```
SubclaseFinal      a = new SubclaseFinal();
SubclaseFinal      b = new SubclaseFinal("===");
SubclaseIntermedia c = new SubclaseIntermedia();
Abstracta          d = new SubclaseIntermedia();
Abstracta          e = new SubclaseFinal("...");
```

Pregunta 2: Después de realizadas las declaraciones anteriores ¿Qué imprime cada una de estas instrucciones? Relaciona los casos que son similares entre sí y explica por qué lo son.

```
System.out.println(a.raro(2,10));
System.out.println(b.raro(2,10));
System.out.println(c.raro(2,10));
System.out.println(d.raro(2,10));
System.out.println(e.raro(2,10));
```

Solución: Independientemente del tipo de datos con que se declara cada variable (antes de su identificador) y dado que todas sus inicializaciones son correctas, lo que determina qué constructor se ejecutará en primer lugar no es el tipo de la variable sino la clase del objeto que se esta creando con **new**.

En la inicialización de `a`, `b` y `e` se ejecuta un constructor de los dos que tiene declarados la clase `SubclaseFinal`; en ambos constructores se imprime un texto, pero antes de ello se invoca automáticamente al constructor de la superclase inmediata que resulta ser `SubclaseIntermedia`, y cuyo funcionamiento describiremos más abajo; el texto que se imprime, después de ejecutado ese otro constructor, será el String que se haya pasado como

parámetro al constructor invocado con **new**, o bien la cadena "finalmente" si no se pasó ningún parámetro al invocar al constructor con **new**.
En las otras dos inicializaciones, las de c y d, al igual que de forma automática desde el constructor de SubclaseFinal, se ejecuta el único constructor visible de la clase SubclaseIntermedia; habría sido un error intentar invocar al segundo constructor de SubclaseIntermedia pasándole un String, dado que este constructor fue declarado como private, es decir, visible sólo desde dentro de la clase SubclaseIntermedia. Ahora bien, el primer constructor de SubclaseIntermedia, al que se ha invocado sin parámetros, invoca explícitamente (usando **this**) al constructor privado de esa misma clase SubclaseIntermedia pasándole el String "pero OK" como parámetro, y después imprime él mismo el String "y feliz". Esta invocación entre constructores de la misma clase no podría ser recursiva, y no lo es, afortunadamente. Se detectaría un error si este segundo constructor intentase invocar al primero (el que le invocó a él) con una instrucción como **this()**, y afortunadamente Java no incluye esa invocación automáticamente en este caso.
Como hemos visto, en toda las inicializaciones, la secuencia de invocación de constructores llega hasta el constructor privado de SubclaseIntermedia, que antes de imprimir el String que haya recibido como parámetro e, invoca implícitamente al constructor de su superclase inmediata, es decir, la clase al que Abstracta; nótese que esta clase abstracta no puede declarar su constructor como abstracto, aunque sí que podría no declararlo o declararlo con un cuerpo vació; además, no podría darle visibilidad private, porque va a ser invocado desde fuera de la clase Abstracta.
Aquí termina la cadena visible de invocaciones a constructores, ya que la invocación al constructor de la superclase inmediata de Abstracta, que es la clase Object, no genera ningún efecto visible para nosotros. Así pues, todas las inicializaciones imprimen en primer lugar, lo que imprime el constructor de la clase Abstracta, y después se irá devolviendo control sucesivamente a los constructores cuya invocación está pendiente de terminar de ejecutarse, para que cada uno imprima sus mensajes.

- La declaración
```
SubclaseFinal      a = new SubclaseFinal();
```
imprime lo siguiente:
```
clase abstracta
pero OK
y feliz
finalmente
```

- La declaración
```
SubclaseFinal      b = new SubclaseFinal("===");
```
imprime lo siguiente:
```
clase abstracta
pero OK
y feliz
===
```

- La declaración
```
SubclaseIntermedia c = new SubclaseIntermedia();
```
imprime lo siguiente:
```
clase abstracta
pero OK
y feliz
```

- La declaración
```
Abstracta          d = new SubclaseIntermedia();
```
imprime lo siguiente:
```
clase abstracta
pero OK
y feliz
```

- La declaración
```
Abstracta          e = new SubclaseFinal("...");
```
imprime lo siguiente:

```
clase abstracta
pero OK
y feliz
...
```

El método `raro` se define como abstracto y público en la clase `Abstracta`, y por tanto, sólo podrá tener visibilidad `public` en todas las clases donde se implemente. La primera subclase que lo implementa es `SubclaseIntermedia`, en la que el método simplemente suma sus dos parámetros y devuelve el resultado. Además, la clase `SubclaseFinal` hereda el método `raro` implementado `SubclaseFinal`, y lo sobrescribe con una nueva implementación que suma su segundo parámetro `y` con el resultado de invocar al método `raro` de su superclase inmediata de la siguiente forma: **`super.`**`raro(x,x)`, es decir, con ello se calcula `x + x` y al resultado se le suma `y`. Así pues, las invocaciones al método `raro(2,10)` sobre las variables `a`, `b` y `e`, que contiene un objeto de tipo `SubclaseFinal`, calculan 2 + 2 + 10, y las otras dos invocaciones calculan simplemente 2 + 10. Por lo tanto, en el fragmento de código anterior, las llamadas a `raro` imprimen lo siguiente:

```
14
14
12
12
14
```

30) Encuentra todos los errores que hay en este conjunto de subclases. Hay muchos, y recuerda que el compilador no los detecta todos en una primera pasada, pero tú sí que puedes: aunque todavía no sabes tanto Java como el compilador, tú eres inteligente y él es sólo un programa repite siempre el mismo algoritmo *sin entender* lo que hace.

```
abstract public class Abstracta {
  public int z = 0;

  abstract public void misterio(int x, int y);
  public int oscuro(int x, int y);
}

class Subclase extends Abstracta {
  final public int oscuro(int x, int y) {
    z = z + x * y;
  }
  abstract public void raro(int x, int y);
}

final class ClaseHoja extends Subclase {
  abstract public void misterio(int x, int y) {
    z = x + y;
  }
  public int oscuro(int x, int y) {
    z =  z * x * y;
  }
  protected void raro(int x, int y) {
    z = z + x + y;
  }
  protected void raro(int x) {
    z = z + x;
  }
}

class Clase extends ClaseHoja {
  public void raro(int x) { z = z + x * x; }
}
```

Solución: Hay 7 errores que se comentan a continuación sin dar relevancia al orden de presentación:

1. La historia del método `oscuro` es compleja y contiene al menos tres errores diferentes. El método se declara en la clase `Abstracta` sin darle implementación pero sin declarar el método como `abstract`.
2. Dando por supuesto que la intención era declarar el método `oscuro` como abstracto en la clase `Abstracta`, posteriormente se implementa en `Subclase` y se le da en ese momento la característica de método `final`, lo cual hace erróneo el intento de sobrescribir dicho método en `ClaseHoja`.
3. Por último, tanto la implementación válida del método `oscuro` en `Subclase` como el intento erróneo de implementación en `ClaseHoja` se olvidan de que el método debe devolver un dato de tipo `int` y de hecho no incluyen ninguna instrucción `return`.
4. La clase de nombre `Subclase` declara como `abstract` el método `raro`, y por tanto la clase misma debe ser declarada también como `abstract`.
5. En `ClaseHoja` el método `raro` con dos parámetros concreta la implementación del método abstracto `raro` de `Subclase`, pero esa implementación intenta asignar visibilidad `protected` mientras que la especificación abstracta correspondiente le asignaba mayor visibilidad, `public`, y eso no es posible. Se puede restringir la visibilidad dar implementación a un método abstracto pero no se puede ampliarla. Obviamente, esto no tiene nada que ver con la visibilidad `protected` que se asigna en la `ClaseHoja` al método `raro` de un solo parámetro, ya que este método no es una implementación del método abstracto `raro` sino que se trata de una sobrecarga del mismo nombre de método.
6. La última clase, de nombre `Clase`, no puede ser una subclase de la `ClaseHoja` que está definida como `final`.
7. Por esa misma razón no es correcto (ni tiene sentido) declarar el método `misterio` como `abstract` en la clase `final ClaseHoja`, ya que no esta clase no tendrá subclases que puedan dar implementación a dicho método.

31) Analiza el siguiente programa y explica todos los errores que detectará el compilador. Elimina todas las líneas del método `main` que contengan algún error y explica el resultado que se obtendría al ejecutar el programa después de eliminar las líneas erróneas.

```
class Sorpresa {
  private final void flipar() { System.out.println("Incognita"); }

  public Sorpresa(String s) { flipar(); }
  public Sorpresa() { }
}

public class Misterio extends Sorpresa {
   public void flipar() { System.out.println("Misterio"); }

   public static void main(String [] args) {
          Misterio x = new Misterio();
          x.flipar();
          Sorpresa y = new Sorpresa("¡Hola!");
          // eliminada por error: método flipar de Sorpresa no visible
          // y.flipar();

   }
}
```

Solución: Aunque pudiera pensarse que el método `flipar()` de `Sorpresa` no se puede sobrescribir en `Sorpresa`, ya que se declaró como método final, lo cierto que al haberse declarado como privado pues *no hay error en declarar otro método con el mismo nombre* en

Sorpresa. Nótese el énfasis: no se sobrescribe del método original porque no está visible en `Sorpresa`, sino que se declara un nuevo método con el mismo nombre. Es por esta razón que `y.flipar()` intenta invocar a un método `flipar` de la clase `Sorpresa`, que ha sido declarado como privado y el compilador detecta el error. Tras eliminar este error, la ejecución del programa inicializa `x` con el constructor sin parámetros de `Misterio` (que invoca implícitamente al constructor sin parámetros de `Sorpresa`), luego tan solo se imprime `Misterio`, y la declaración de `y` invoca al constructor de `Sorpresa` que recibe un parámetro String; luego la salida del programa es:

```
Misterio
Incognita
```

La pregunta que surge ahora es si la ligadura dinámica de Java haría que la siguiente invocación a `flipar()` fuese correcta, pero no lo es, sigue produciendo error de compilación:

```
Sorpresa z = new Misterio();
z.flipar();
```

32) Analiza el siguiente programa y explica todos los errores que detectará el compilador. Elimina todas las líneas del método `main` que contengan algún error y explica el resultado que se obtendría al ejecutar el programa después de eliminar las líneas erróneas.

```
class Alpha {
   static String s = "=";
   protected Alpha() { s += "alpha "; }
 }

class SubAlpha extends Alpha {
   private SubAlpha() { s += "sub "; }
 }

public class PruebaAlpha extends Alpha {
   private PruebaAlpha() { s += "subsub "; }

   public static void main(String[] args) {
       // eliminada por error: constructor SubAlpha() no visible
       // new SubAlpha();
       System.out.println(s);
        new PruebaAlpha();
        System.out.println(s);
        new Alpha();
        System.out.println(s);
 }
}
```

Solución: En primer lugar, que no se nos olvide que estamos concatenando Strings con el operador **+=** que modifica el valor de su primer argumento, por lo que los constructores van añadiendo su texto sobre lo ya almacenado en `s`. Hay dos subclases de `Alpha`, que se llaman `SubAlpha` y `PruebaAlpha`. Esta última es la que tiene el método `main` estático y por lo tanto, hereda el atributo `s` declarado en `Alpha` que también es estático, por lo que no hay problema en ninguno de las instrucciones que lo imprimen. Ahora bien, hemos tenido que comentar el primer `new` que se pretendía hacer dentro de `main`, porque no se puede crear una nueva instancia de `SubAlpha` dado que su constructor es privado (lo cual en este caso hace inútil la existencia de esa clase, pero en general no se puede descartar la utilidad de los constructores privados). Aunque el constructor de la superclase `Alpha` se declara como protected y estará accesible desde sus dos subclases, al no poderse ejecutar el constructor privado de `SubAlpha` tampoco se invoca implícitamente el constructor de la superclase `Alpha`. En cambio, esta invocación automática del constructor de `Alpha` sí que se produce al crear una instancia de `PruebaAlpha`. Tras eliminar el error comentado arriba, la salida es:

```
=
=alpha subsub
=alpha subsub alpha
```

33) Analiza el siguiente programa y justifica el resultado que produciría su ejecución (en caso de ser correcto), o bien explica los errores que contiene, propón una forma razonable de corregir cada error y explica el resultado que produciría su ejecución tras la corrección propuesta.

```
class Mamifero {
  String nombre = "(peludo)";
  String hacerRuido() { return "(gruñido)"; }
  String getNombre() { return nombre; }
}

class Elefante extends Mamifero {
  String nombre = "(trompa)";
  String hacerRuido() { return "(barrita)"; }
}

public class PruebaZoo {

  public static void main(String[] args) {
     new PruebaZoo().vamos();
  }

  void vamos() {
     Mamifero m = new Elefante();
     System.out.println(m.nombre + m.hacerRuido() + m.getNombre());
     Mamifero n = new Mamifero();
     System.out.println(n.nombre + n.hacerRuido() + n.getNombre());
     Elefante p = new Elefante();
     System.out.println(p.nombre + p.hacerRuido() + p.getNombre());
  }
}
```

Solución: No hay errores que eliminar en ese programa. En este programa se crean tres objetos: un `Elefante` almacenado en la variable `m` de tipo `Mamifero`; un `Mamifero` almacenado en la variable `n` de tipo `Mamifero`; y un `Elefante` almacenado en la variable `p` de tipo `Elefante`. En cada instrucción de salida se imprime la concatenación de los Strings resultantes de tres operaciones sobre uno de esos objetos: el primer String resulta de acceder al atributo no privado `nombre` declarado en la clase `Mamifero` y también en su subclase `Elefante`; el segundo resulta de invocar al método `hacerRuido()` declarado en `Mamifero` y sobrescrito en la subclase `Elefante`; y el tercero resulta de invocar al método `getNombre` declarado únicamente en la clase `Mamifero`. El primer valor del primer String, accede a un atributo o variable de instancia de la clase, es decir, depende del tipo de la variable en cuestión y es por ello que solo se imprime `(trompa)` para la variable de tipo `Elefante`. En cambio, por la ligadura dinámica que emplea Java en la invocación a métodos, los otros dos Strings dependen de la clase del objeto que almacenen las variables. Así, obviamente el tercer String siempre es `(peludo)`, ya que solo hay un método `getNombre()`, el definido en la clase `Mamifero` y que no puede acceder al atributo nombre la subclase `Elefante`. El caso del segundo String es menos obvio pero tampoco es complicado, ya que el método `hacerRuido()` definido en la clase `Mamifero` se sobrescribe en la subclase `Elefante`, y *cada objeto ejecuta dinámicamente* el método de *su* clase. Es decir, la salida que se obtiene es:

```
(peludo)(barrita)(peludo)
(peludo)(gruñido)(peludo)
(trompa)(barrita)(peludo)
```

34) Analiza el siguiente programa y justifica el resultado que produciría su ejecución (en caso de ser correcto), o bien explica los errores que contiene, propón una forma razonable de corregir cada error y explica el resultado que produciría su ejecución tras la corrección propuesta.

```
class Top {
    public Top(String s) {
              System.out.println("TOP: " + s);
    }
    // añadimos construtor al que se llamaba por defecto más abajo
    // podría estar vacio public Top(){} pero añadimos algo para ver su ejecución
    public Top() { System.out.println("TOP: null"); }
}

public class Test extends Top {
  public Test(String s) {
         // se llama por defecto a super() pero no existe constructor Top()
         System.out.println("Test: " + s);
  }
  public static void main(String[] args) {
    new Test("MAIN");
    System.out.println("END");
  }
}
```

Solución: El error en el código original está en que el constructor `Test(String s)` intenta invocar implícitamente al constructor por defecto de la superclase `Top`, es decir, `Top()`, pero éste constructor no está definido porque al definirse un constructor `Test(String s)` en `Top` ya no se define automáticamente el constructor por defecto `Top()`. La invocación implícita siempre es al constructor por defecto `Top()`, es decir, que aunque estemos ejecutando el constructor `(String s)` de `Test` no se invoca automáticamente al constructor con parámetros `(String s)` de `Top`. Dicha invocación podría haberse hecho explícitamente con `super(s)` como primera instrucción en el constructor de `Test`. No obstante en este caso, hemos optado por corregir el programa de la forma más sencilla posible, añadiendo el constructor `Top()` que el compilador reclamaba en la clase `Top()`, y aunque se podía haber implementado dicho constructor sin ninguna instrucción en su cuerpo, hemos optado por poner una instrucción de salida para que se note su ejecución. Tras eliminar así e error comentado arriba, la salida es:

```
TOP: null
Test: MAIN
END
```

35) Analiza el siguiente programa y explica todos los errores que detectará el compilador. Elimina todas las líneas del método `main` que contengan algún error y explica el resultado que se obtendría al ejecutar el programa después de eliminar las líneas erróneas.

```
package paquete;
public class Simple {
       int a = 5;
       protected int b = 6;

       public Simple() { System.out.println("New Simple"); }
}

package paquito;
import paquete.*;
public class Magica extends Simple {

   public Magica() { System.out.println("New Magica"); }

   public static void main(String[] args) {
     Simple f = new Simple();
     Magica m = new Magica();
     Simple s = new Magica();
     // eliminamos solamente la líneas con error
     // Magica r = new Simple();
     // System.out.println("1: " + f.a);
     // System.out.println("2: " + f.b);
     // System.out.println("3: " + m.a);
     System.out.println("4: " + m.b);
     // System.out.println("5: " + s.a);
     // System.out.println("6: " + s.b);
     // System.out.println("7: " + r.a);
     // System.out.println("8: " + r.b);
   }
}
```

Solución: Hay muchos errores que eliminar en ese programa. En este programa se crean tres objetos: un `Simple` almacenado en la variable `f` de tipo `Simple`; un `Magica` almacenado en la variable `m` de tipo `Magica`; y un `Magica` almacenado en la variable `s` de tipo `Simple`. El intento de construir un `Simple` para almacenarlo en la variable `r` de tipo `Magica` es uno de los errores que detecta el compilador, porque el objeto pertenece a la clase `Simple` que es superclase de `Magica`, es decir, es más general que el tipo de la variable a la que se iba a ser asignado. Obviamente, al eliminar dicha variable, tuvimos que eliminar las dos líneas en las que se hacía referencia a ella. El resto de los errores provienen del hecho de que la superclase y la subclase se han definido en dos paquetes diferentes, por lo que resulta relevante la diferencia entre la visibilidad `protected` (que significa visible en objetos de las subclases de la clase actual) y la visibilidad *package* (que se obtiene por defecto, sin usar modificador de visibilidad, y significa visibilidad en el paquete actual). Dado que el atributo o variable de instancia `a` declarado en la clase `Simple` del paquete `paquete` tiene visibilidad *package*, resultan erróneos todos los intentos de acceder a dicho componente desde `main` que pertenece a un paquete distinto. En cambio, el atributo `b` ha sido declarado con visibiliadad `protected`, lo cual le hace obviamente visible en la variable `m` declarada de tipo `Magica`, subtipo de `Simple`, pero no así en las otras variables declaradas de tipo `Simple` (ya que éstas podrían no almacenar objetos de la subclase `Magica`, como es el caso de la variable `f`). En cada instrucción de salida Tras eliminar todos los errores con los comentarios de arriba, y recordando que cada invocación del constructor `Magica()` incluye una invocación implícita del constructor `Simple()`, la salida que se obtiene es:

```
New Simple
New Simple
New Magica
New Simple
New Magica
4: 6
```