

## *Práctica 3. Elementos básicos del lenguaje y del compilador*

Profesores de IIP  
Departamento de Sistemas Informáticos y Computación  
Universitat Politècnica de València



### Índice

1. Objetivos y trabajo previo a la sesión de prácticas	1
2. Descripción del problema	1
3. Compilación de código Java. Detección de errores	1
4. Errores de ejecución. Prueba de los programas	2
5. Normas de estilo en la escritura de código	3
6. Actividades de laboratorio	3

## 1. Objetivos y trabajo previo a la sesión de prácticas

El objetivo principal de esta práctica es iniciar al alumno en el trabajo de desarrollo de código, compilación y prueba de los programas, al mismo tiempo que se ejercitan algunos de los conceptos básicos introducidos en el tema 3 (*Variables: definición, tipos y usos*) acerca de la declaración y uso de las variables, los tipos numéricos enteros de Java y el uso de algunas de sus librerías fundamentales. En particular, se incidirá en:

- Introducción de datos por teclado y escritura de resultados en el terminal.
- Manejo de la aritmética de enteros.
- Compatibilidad de tipos y conversión forzada de la representación (*casting*).
- Sobrecarga del operador + (concatenación de cadenas de caracteres además de la operación suma).
- Consulta y uso de las librerías `String`, `System` y `Math`.

## 2. Descripción del problema

Es habitual en muchas aplicaciones informáticas que se precise manejar como dato una hora, poder calcular el tiempo transcurrido entre diferentes horas, saber si una hora dada es anterior o posterior a otra, ...

En concreto, en esta práctica se debe implementar una Clase Programa `Test3` que escriba en la salida estándar los datos correspondientes a una hora (horas y minutos) en el formato "`hh:mm`". El programa calculará la diferencia en minutos entre un par de horas: una cuyos datos deberán

introducirse por teclado, y la hora actual, que deberá calcularse automáticamente. Además, ambas horas se mostrarán en el terminal en el formato arriba indicado.

Las actividades propuestas están orientadas a discutir las dificultades que pueden surgir al realizar estos cálculos.

### 3. Compilación de código Java. Detección de errores

El proceso de obtención de código ejecutable a partir del fuente precisa que éste sea sintácticamente correcto; sino, la traducción no es posible y la compilación falla. El programa compilador, como consecuencia del análisis sintáctico parcial o totalmente realizado, produce una lista de las líneas en las que ha encontrado los errores y alguna indicación del origen del error. Dada la complejidad del análisis, estas indicaciones pueden no ser todo lo precisas que sería deseable, necesitándose en ocasiones cierta experiencia para saber interpretar adecuadamente estos mensajes.

Se recomienda en cualquier caso, leer los mensajes con detenimiento. A continuación se citan unos pocos ejemplos de entre la amplia casuística que suele presentarse:

- No se reconoce un identificador o símbolo. Puede ser que se haya tecleado mal el nombre de una variable, un método o una clase, y no coincida con el de su declaración. En el caso de una variable, puede ser que se haya olvidado incluir dicha declaración. En el caso de usar métodos de una clase que no sea de `java.lang`, puede ser que se haya olvidado la importación del paquete que la contiene.
- En una expresión sintácticamente incorrecta se puede detectar la falta de un paréntesis u otro signo, la falta de un operando, etc. A veces, la interpretación del mensaje de error debe ir más allá de una lectura al pie de la letra. Considérese por ejemplo, la siguiente línea de código en la que se pretendía sumar `x + y` y en la que por error se ha tecleado un blanco en lugar de `+`:

```
int z = x y;
```

El análisis sintáctico avanza hasta `int z = x`, notándose en este punto que la instrucción no está completa. El compilador puede dar en el mensaje de error una indicación de que un `;` finalizaría de forma sintácticamente correcta la asignación; en este ejemplo concreto, la intención del programador era otra y la expresión se debe completar con el símbolo de operación que falta.

- En el análisis de un bloque mal parentizado se puede detectar una llave que falta (olvidada, o que podría corresponder a una llave anterior superflua o mal ubicada). Una llave descolocada puede también provocar que falte o sobre alguna sección de código en un determinado lugar y que sea este el error que se detecte. Por ejemplo:

```
import java.util.Scanner;
{ public class Hola
    public static void main(String[] args) {
        ...
    }
}
```

La primera llave no está en su sitio, y en el análisis lo que se advierte es que después de la cláusula de importación debería seguir una clase.

Según el caso, puede ser preciso que el programador examine una zona de código por delante o por detrás de la línea en la que el compilador se ha percatado del error.

## 4. Errores de ejecución. Prueba de los programas

Un programa sintácticamente correcto y que se compila sin problemas puede producir errores de ejecución: resultados erróneos, o estados de fallo o *excepciones* que provocan la interrupción abrupta de la ejecución.

La fuente de los errores puede ser de diversa naturaleza; en programas pequeños son habitualmente debidos a una estrategia de resolución del problema equivocada, algún caso del problema no contemplado, uso incorrecto de operaciones por un conocimiento impreciso de su significado, o incluso erratas cometidas al teclear el código que no necesariamente implican errores sintácticos.

Es por ello que de forma rutinaria se debe someter al programa a ejecuciones de prueba contrastando los resultados obtenidos con los esperados.

## 5. Normas de estilo en la escritura de código

Además de que un código fuente sea sintáctica y semánticamente correcto, es deseable que esté escrito de forma clara y ordenada para facilitar su lectura y mantenimiento. Sucede lo mismo con la edición de textos convencionales, como libros y periódicos, que adoptan normas de estilo comúnmente aceptadas o sobreentendidas, que estructuran el texto y facilitan su comprensión.

Ello es especialmente importante en el caso de desarrollo de código, porque a lo largo de la vida de los programas, es habitual que estos sean mantenidos por diversos programadores. Es por eso que se han desarrollado unas convenciones de escritura de código que, sin ser de obligado cumplimiento, se recomienda que adopte la comunidad de programadores en Java.

Estas convenciones, que afectan tanto al código como al uso de comentarios, fueron desarrolladas por el anterior propietario de Java, Sun Microsystems Inc., y actualmente permanecen accesibles para su consulta en:

<http://www.oracle.com/technetwork/articles/javase/codeconvtoc-136057.html>

A continuación se repasan algunas de las más elementales.

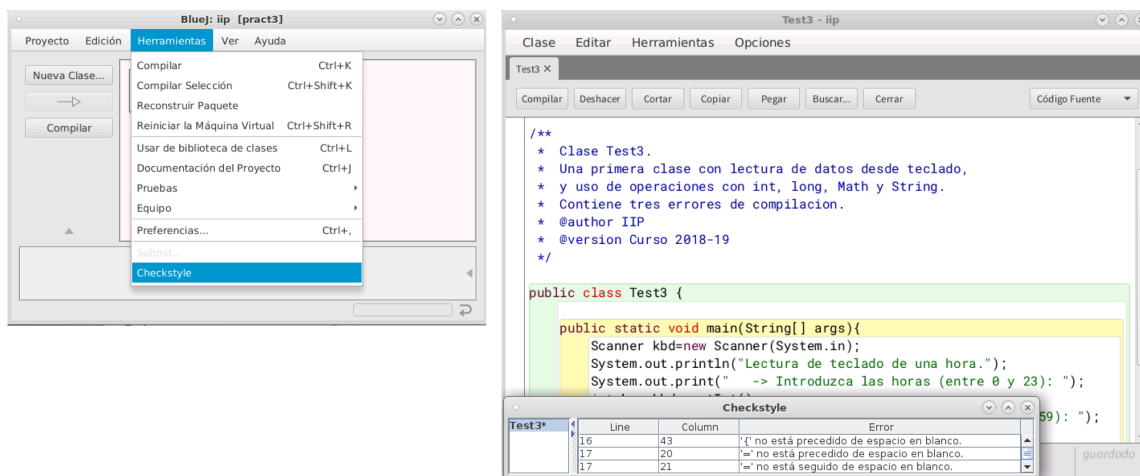
- **Nombres de clases, métodos y variables.** Se deben escoger identificadores indicadores de su significado. Métodos y variables empiezan en minúscula, mientras que la inicial de las clases debe estar en mayúscula. Cuando un nombre proviene de la yuxtaposición de dos o más palabras, la segunda y sucesivas deben empezar por mayúscula como, por ejemplo, en `MiPrimeraClase`, `horaActual`.
- **Parentización de bloques y sangrado del texto.** La llave que abre el bloque de una clase o un método, como por ejemplo `main`, debe aparecer al final de la línea en la que se declara la clase o método. La llave que cierra el bloque debe aparecer con el mismo sangrado que la clase o método. Cuando se escriben las sentencias componentes de un bloque, deben aparecer sangradas cuatro espacios hacia la derecha, para que sea más reconocible dicha estructuración, como se observa en el siguiente ejemplo:

```
public class Hola {  
    public static void main(String[] args) {  
        System.out.println("Hola a todos");  
    }  
}
```

- **Escritura de expresiones.** El operador de asignación `=` y los operadores binarios aritméticos, relaciones y lógicos `+`, `*`, `...`, `<`, `<=`, `...`, `&&`, `&`, `...`, deben estar precedidos y seguidos por un espacio.
- **Longitud y ruptura de las líneas.** Las líneas demasiado largas son desaconsejables. Cuando hay que partir líneas, se deben seguir las normas que facilitan reconocer la estructura de las expresiones e instrucciones, como en el siguiente ejemplo:

```
System.out.println("La primera raíz real de la ecuación vale "  
    + (-b + Math.sqrt(b * b - 4.0 * a * c)) / (2.0 * a));
```

Para promover el uso de estas convenciones se ha instalado la extensión de *BlueJ Checkstyle*. En concreto, si una vez resueltos los errores de compilación se marca la opción *Checkstyle* del desplegable de *Herramientas* de *BlueJ*, aparece una ventana que proporciona información sobre las normas de estilo incumplidas por el código que se está escribiendo, como en el ejemplo de la siguiente figura:



## 6. Actividades de laboratorio

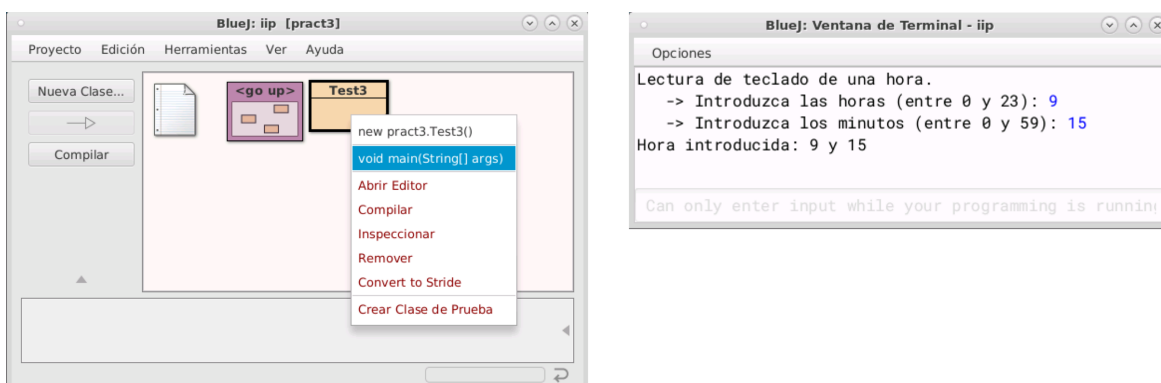
### Actividad 1: Creación del paquete BlueJ pract3

Abrir el proyecto *BlueJ* de trabajo de la asignatura (iip) y crear un nuevo paquete **pract3**. Agregar al paquete el fichero **Test3.java** que se habrá descargado previamente de la carpeta Recursos/Laboratorio/Práctica 3 de la PoliformaT de IIP.

### Actividad 2: Corrección de errores sintácticos de la clase Test3

El código fuente de la clase **Test3.java** agregada al paquete contiene deliberadamente tres errores sintácticos que impiden obtener el código compilado. Se debe editar la clase para buscar y corregir dichos errores atendiendo a los mensajes de error producidos al intentar la compilación.

En la siguiente figura se muestra un ejemplo de ejecución de la clase, una vez corregida y compilada correctamente:

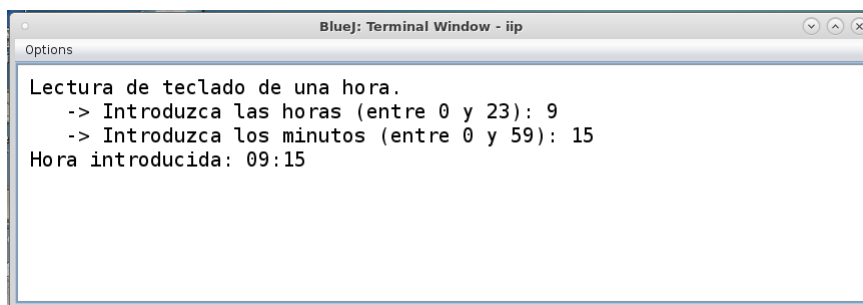


### Actividad 3: Comprobación de las normas de estilo

Utilizar la opción de comprobar estilo del menú *Herramientas*, y aplicar las sugerencias de mejora de estilo que se indiquen.

## Actividad 4: Escritura de la hora introducida en el formato hh:mm

Se debe modificar el programa anterior para que la hora introducida se escriba en la salida en el formato de cuatro dígitos **hh:mm**, completando con ceros si es preciso, como en la ejecución ejemplo de la siguiente figura:



```
Bluej: Terminal Window - iip
Options
Lectura de teclado de una hora.
-> Introduzca las horas (entre 0 y 23): 9
-> Introduzca los minutos (entre 0 y 59): 15
Hora introducida: 09:15
```

Para resolver el problema se recomienda usar las operaciones de la clase **String**, de modo que para un par horas y minutos se componga un **String** de la forma "**hh:mm**" que sea el que se escriba en la salida.

Una manera directa y sencilla de conseguirlo es anteponer siempre "0" a las horas (respectivamente a los minutos) quedando un **String** de dos o tres caracteres según el caso. Ejemplos: si **h** vale 9, la expresión "0" + **h** da "09"; si **h** vale 10, el **String** resultante es "010". Independientemente del valor de **h**, basta con obtener el substring formado por los dos últimos caracteres de dicho **String** para obtener la representación en dos dígitos que se precisa. Para ello, es conveniente recordar que, según la documentación de **String**:

- El método **int length()** permite obtener el número  $n$  de caracteres de un **String**, que se suponen numerados de 0 a  $n - 1$ .
- El método **String substring(int i)**, obtiene el substring formado por los caracteres del **String** desde el **i** hasta el último, ambos inclusive. En este problema, el substring que interesa es el que se extiende desde el índice  $n - 2$  en adelante.

Suponiendo que en la variable **String hh** se hubiesen obtenido las horas en formato de dos dígitos y en la variable **mm** los minutos en el mismo formato, la hora en el formato solicitado se obtiene mediante la concatenación **hh + ":" + mm**.

Se deberá comprobar la corrección del programa probando a introducir para las horas y minutos casos como los de la siguiente tabla y verificando la hora que se escribe en la salida:

horas	minutos	"hh:mm"
0	0	"00:00"
9	5	"09:05"
9	35	"09:35"
19	5	"19:05"
19	35	"19:35"

## Actividad 5: Cálculo de la hora actual (Tiempo Universal Coordinado)

El programa se deberá completar calculando las horas y minutos correspondientes al momento de ejecución del programa. Para ello, se puede hacer una llamada al método

**System.currentTimeMillis()**

que, como se indica en la documentación de la clase **System**, devuelve el número de milisegundos transcurridos entre las 00:00 UTC <sup>1</sup> del 1 de enero de 1970 y el momento actual UTC.

<sup>1</sup> *Tiempo Universal Coordinado*, estándar mundial equivalente en la práctica a la *hora Greenwich*. En España, salvo Canarias, rige la *hora Greenwich* + 2 en horario de verano, y la *hora Greenwich* + 1 en horario de invierno.

El método devuelve una cantidad entera de tipo `long`, dado que su valor es muy grande y no cabe en una variable de tipo `int`. Como en el cálculo que se va a realizar no interesan unidades de tiempo inferiores a un minuto, la siguiente instrucción Java de tipo `long` calcula el número de minutos completos como el cociente de la división entera entre esa cantidad y el valor  $60 * 1000$  que son el total de milisegundos de un minuto y lo almacena en una variable `tMinTotal` de tipo `long`:

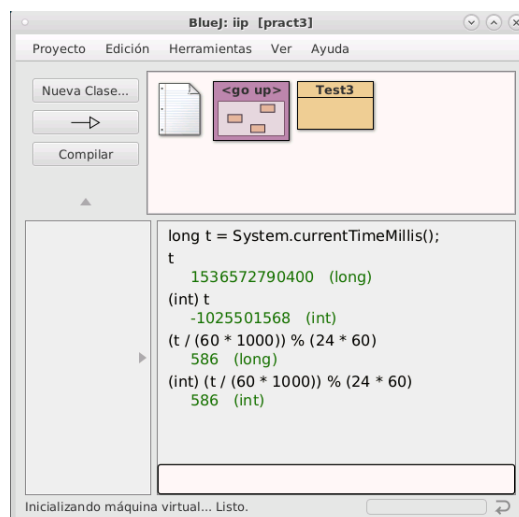
```
long tMinTotal = System.currentTimeMillis() / (60 * 1000);
```

Así, en `tMinTotal` se tienen, por lo tanto, los minutos transcurridos UTC desde las 00:00 del 1 de enero de 1970 y el día actual hasta el momento en que se ejecuta la instrucción anterior. De todos esos minutos obtenidos, sólo nos interesan los correspondientes al día actual, que se puede calcular como el resto de la división entera entre el resultado de la operación anterior y el número de minutos de un día completo que es  $24 * 60$ . Los minutos del día actual son un número mucho más pequeño que el obtenido anteriormente y sí se puede representar como `int`.

En la siguiente instrucción se fuerza mediante un *casting* dicha transformación de representación sabiendo que el valor entero correspondiente se preserva, y pudiendo asignarse además a una variable de tipo `int` sin que se produzca un error de compilación por incompatibilidad de tipos:

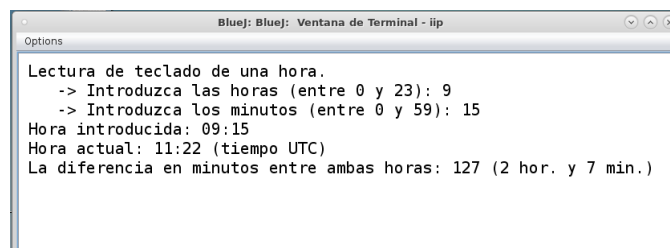
```
int tMinCurrent = (int) (tMinTotal % (24 * 60));
```

En la figura que viene a continuación se muestran ejemplos del comportamiento del *casting* de `long` a `int` según el rango del valor entero transformado.



A partir del valor de `tMinCurrent` calculado como en la instrucción anterior, se pueden obtener mediante sencillas operaciones aritméticas las horas y minutos del momento actual que se precisan para resolver el enunciado.

Finalmente, se debe calcular y mostrar también por pantalla el valor absoluto de la diferencia en minutos entre la hora introducida por teclado y la hora actual (consultar en la documentación de la clase `Math` los métodos disponibles para calcular el valor absoluto de enteros y reales), expresados a continuación en horas y minutos. El resultado de la ejecución debe ser como el que se muestra en el ejemplo de la figura siguiente:



## Actividades extra.

Una vez resueltas las actividades básicas de la práctica, se proponen las siguientes actividades extra que se pueden resolver en el laboratorio si queda suficiente tiempo. En cualquier caso, constituyen unos ejercicios que pueden permitir al alumno repasar en su tiempo de estudio algunas peculiaridades de la concatenación de **String**, y conceptos básicos acerca de la sintaxis de expresiones de tipo **boolean**.

1. Resolver el cálculo del formato "**hh:mm**" de escritura de una hora de una manera alternativa, teniendo en cuenta las siguientes consideraciones:
  - Dada la variable entera **h** que contiene las horas, un valor entre 0 y 23, **h / 10** obtiene la cifra de las decenas (0, si **h < 10**), y **h % 10** obtiene las unidades.
  - Ambas cifras se calculan como valores **int**, pero en la expresión **h / 10 + "" + h % 10** aparecen en un contexto de concatenación + de **String**, con lo que Java los convierte a su representación textual. El uso del literal "", objeto **String** de cero caracteres, consigue que ambas cifras aparezcan una a continuación de la otra en el **String** resultante.

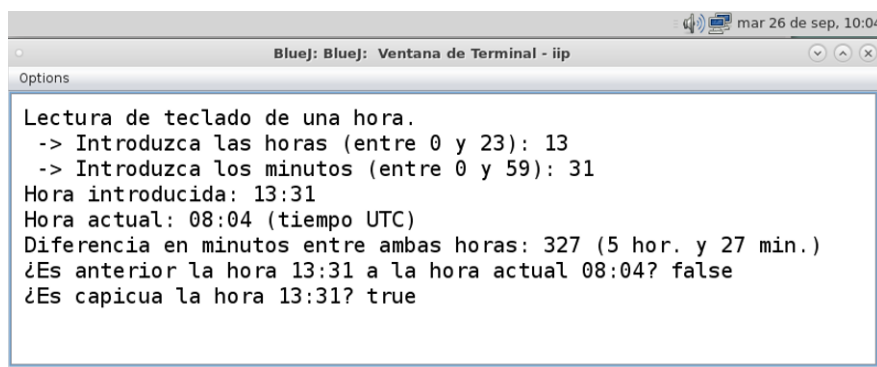
Un cálculo análogo es aplicable a las cifras de los minutos, con lo que se puede completar la escritura de la hora en el formato deseado.

2. Añadir al método **main** de la clase **Test3** las instrucciones necesarias para que se escriban en la salida los siguientes resultados:
  - Valor lógico correspondiente a que la hora introducida por teclado sea anterior a la hora actual.
  - Valor lógico correspondiente a que la hora introducida por teclado, en su formato "**hh:mm**", sea un capicúa. Consultar en la documentación de **String** qué método permite consultar el carácter que aparece en una posición dada.

Se deberán probar casos como los de la siguiente tabla:

horas	minutos	"hh:mm"	¿Capicúa?
0	0	"00:00"	true
2	20	"02:20"	true
13	31	"13:31"	true
19	21	"19:21"	false
13	35	"13:35"	false
19	35	"19:35"	false

Los mensajes resultantes deben ser como los que se muestran en la siguiente ejecución ejemplo:



```
Lectura de teclado de una hora.
-> Introduzca las horas (entre 0 y 23): 13
-> Introduzca los minutos (entre 0 y 59): 31
Hora introducida: 13:31
Hora actual: 08:04 (tiempo UTC)
Diferencia en minutos entre ambas horas: 327 (5 hor. y 27 min.)
¿Es anterior la hora 13:31 a la hora actual 08:04? false
¿Es capicua la hora 13:31? true
```