Resumen de Buenas Prácticas para la Programación

Asignatura Programación II

Clara Benac Earle, Manuel Collado Machuca, Ángel Lucas González Martínez,
Susana Muñoz Hernández, Jaime Ramírez Rodríguez

El objetivo de este documento es mostrar a los alumnos de la asignatura de Programación II algunas convenciones que se siguen en dicha asignatura y que se consideran buenas prácticas de programación. En algunos casos estas buenas prácticas se aplican a cualquier lenguaje de programación y, en otros, son convenciones específicas del lenguaje Java. No seguir estas recomendaciones en los ejercicios y prácticas de la asignatura será penalizado en las notas de los mismos, pudiendo variar el valor de dicha penalización en cada ejercicio.

Las reglas que se indican no son algo absoluto. Hay situaciones en la práctica profesional en que está justificado saltárselas, pero eso debería hacerse sólo si se estima que hay razones suficientes para ello (y se documentan dichas razones).

Aspectos de abstracción

Cada pieza independiente del código (módulo) debe hacer visible la interfaz que define los servicios que ofrece, y mantener ocultos los detalles de implementación.

Visibilidad de atributos de instancia

Los atributos de instancia deben declararse siempre como privados (con el modificador `private`). Si es necesario acceder a ellos individualmente se pueden declarar métodos *getters* o *setters*.

Incorrecto	Correcto

`class Clase {`
`  public int estado;`
`  ...`
`}`
	`class Clase {`
	`  private int estado;`
	`  ...`
	`  public int getEstado() {`
	`  return estado;`
	`  }`
	`  ...`
	`}`

Visibilidad de atributos de clase

Los atributos de clase deberían seguir las reglas de los atributos de instancia. Sin embargo es perfectamente razonable declarar valores constantes de utilidad como atributos de clase públicos.

Correcto

`class Clase {`
`  public static final int MINUTOS_HORA = 60;`
`  public static final int MINUTOS_DIA = 24*60;`
`  ...`
`}`

Visibilidad de métodos

Los métodos de instancia y de clase que ofrecen los servicios deseados deber ser públicos. Los métodos auxiliares usados para estructurar el código dentro de la clase deben ser privados.

Incorrecto	Correcto

`class HoraDelDia {`
`  private int hora, minuto, segundo;`
`  ...`
`  // Ajustar h/m/s al rango 24/60/60`
`  public normalizar() {`
`  ...`
`  }`
`}`
	`class HoraDelDia {`
	`  private int hora, minuto, segundo;`
	`  ...`
	`  // Ajustar h/m/s al rango 24/60/60`
	`  private normalizar() {`
	`  ...`
	`  }`
	`}`

Uso de expresiones y construcciones booleanas

Se puede operar con valores booleanos de manera similar a como se opera con valores numéricos u otros valores simples.

Estructuras if-else innecesarias

No hace falta usar una construcción **`if`** para evaluar un valor booleano que puede darse como valor de una expresión.

Incorrecto	Correcto

`if (numero > 3) {`
`  return true;`
`} else {`
`  return false;`
`}`
	`return numero > 3;`

`boolean aux = false;`
`if (numero > 3) {`
`  aux = true;`
`}`
`return aux;`
	`return numero > 3;`

No hace falta duplicar la condición en las dos ramas de un **`if/else`**.

Incorrecto	Correcto

`if (numero > 3) {`
`  ... acción 1 ...`
`} else if (numero <= 3) {`
`  ... acción 2 ...`
`}`
	`if (numero > 3) {`
	`  ... acción 1 ...`
	`} else {`
	`  ... acción 2 ...`
	`}`

Expresiones booleanas redundantes

En general nunca hace falta combinar las constantes **`true`** y **`false`** como parte de una expresión booleana con más términos.

Incorrecto	Correcto

`if (esBisiesto()==true) {`
`  diasFebrero = 29;`
`} else {`
`  diasFebrero = 28;`
`}`
	`if (esBisiesto()) {`
	`  diasFebrero = 29;`
	`} else {`
	`  diasFebrero = 28;`
	`}`

Claridad del código

El código debe ser legible por personas, de manera similar a un documento técnico, para entenderlo y actualizarlo sin esfuerzos excesivos cuando haya que hacerlo.

Documentación del código

Se debe documentar el código de los **elementos públicos** para indicar los servicios que ofrece (interfaz). Una forma precisa de hacerlo es mediante *precondiciones* y *postcondiciones*. Esta documentación debe poder recopilarse y publicarse de manera separada del resto del código para que sirva como documento de referencia de uso de un paquete o librería.

En la asignatura usaremos *javadoc* para documentar los servicios o métodos.

También se deben documentar internamente los **elementos privados** y la parte de **implementación de los elementos públicos**, para facilitar su comprensión y las tareas de mantenimiento.

Uso de comentarios significativos

Los comentarios deben aclarar el sentido del código, y no repetir literalmente lo que hace.

Incorrecto	Correcto

`// Método ajustarMinuto`
`private void ajustarMinuto() {`
`  if (minuto < 0) {`
`  minuto = minuto + 60;`
`  hora--;  // decrementa hora`
`  } else if (minuto >= 60) {`
`  minuto = minuto - 60;`
`  hora++;  // incrementa hora`
`  }`
`}`
	`/**`
	` * Ajustar los minutos al rango 0-59,`
	` * corrigiendo la hora si es necesario`
	` */`
	`private void ajustarMinuto() {`
	`  if (minuto < 0) {`
	`  minuto = minuto + 60;`
	`  hora--;`
	`  } else if (minuto >= 60) {`
	`  minuto = minuto - 60;`
	`  hora++;`
	`  }`
	`}`

Convenciones de nombrado de Sun/Oracle

En la asignatura seguimos las convenciones de nombrado de Sun/Oracle excepto para el nombrado de paquetes, donde no usamos los prefijos edu, com, gov, etc.

http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html#367

De manera resumida:

- Los nombres compuestos alternan mayúsculas y minúsculas (`ejemploDeNombre`).
- Los nombres de clases empiezan en mayúscula (`class HoraDelDia`, `enum TipoMes`).
- Los nombre de atributos y métodos empiezan en minúscula (`void ajustarHora()`, `int hora`).
- Los nombres de constantes van todo en mayúsculas (`enum TipoMes {ENERO, FEBRERO, ...}`).
- Los nombres de paquetes van todo en minúsculas (`package ficherodetexto`).

Uso de identificadores significativos

Se deben usar identificadores significativos que ayuden a entender el código. Hacen que el código se pueda mantener más fácilmente.

- Ejemplos de identificadores no significativos: `a, b, z, p, aux, temp, n, ...`
- Ejemplos de identificadores significativos: `numero, temperaturaActual, mecanicoJefe`
- Ejemplo de nombre significativo de constante: `TEMPERATURA_MAXIMA`

Se pueden usar identificadores cortos no significativos si tienen un ámbito reducido. Ejemplo, variable de control de un `for`.

Para que sean significativos conviene que los identificadores correspondan a la categoría gramatical de lo que representan:

- Valores (variables, funciones, …): Sustantivos (`horaInicial, distancia, …`)
- Predicados (valores booleanos): Nombres con “es” o “está” (`esBisiesto, estaVacio, ...`)
- Operaciones (métodos `void`): Verbos o nombres de acciones (`imprimir, desplazar, …`)
- Clases o tipos: Sustantivos generales (`Hora, Punto, ...`)

Indentado del código

Para facilitar la lectura, el código debe tener una presentación uniforme:

- Indentado consistente.
- Las líneas de texto deben reflejar la estructura lógica.

La uniformidad es más importante que los detalles particulares del formato (paso de indentado, saltos de línea antes o después de las llaves, espacio antes o después de operadores y paréntesis, …). Se recomienda usar una utilidad de formateo automático. En Eclipse se invoca mediante los atajos de teclado `ctrl-i` (ajustar indentado) y `ctrl+may-f` (reformar código).

Organización del código

En general, cada bloque o sentencia estructurada debe tener sólo un punto de entrada (inicio de ejecución) y un punto de salida (fin de ejecución). Es difícil entender código que se interrumpe a la mitad (salidas intermedias).

No se debe interrumpir un bucle con un break, continue o return

Para interrumpir las repeticiones de un bucle antes de realizar todas la repeticiones posibles es preferible usar variables booleanas y condiciones lógicas en la cabecera del bucle (estas condiciones se llaman “condiciones de parada”).

Incorrecto	Correcto

`int[] miArray;`
`......`
`for (int i=0; i<max; i++) {`
`  if (miArray[i]==VALOR_BUSCADO) {`
`  return i;`
`  }`
`}`
`return -1;`
	`int[] miArray;`
	`......`
	`int pos = -1;`
	`for (int i=0; i<max && pos<0; i++) {`
	`  if (miArray[i]==VALOR_BUSCADO) {`
	`  pos = i;`
	`  }`
	`}`
	`return pos;`

Exceso de código

Conviene evitar escribir más código del necesario.

Evitar la duplicación de código

Si un cálculo o tarea se realiza muchas veces en un programa, se debe escribir un método (se llama método auxiliar) que realice esa tarea, en vez de escribir el mismo código una y otra vez. De esta manera se minimiza el riesgo de introducir errores y, además, si hay que hacer algún cambio en ese código común, sólo hay que hacer el cambio en un sitio. Los métodos auxiliares normalmente no son públicos puesto que forman parte de otros métodos y no tienen sentido por sí solos.

Por otra parte, si en varias ramas de un `if` o `switch` hay cálculos parciales comunes, puede ser preferible realizar primero esos cálculos en variables temporales, y luego usar el `if` o `switch` para operar con los valores precalculados.

Incorrecto	Correcto

`if (condicion) {`
`  x = a * 7 - b + 5;`
`} else {`
`  x = a * 7 - b + 8;`
`}`
	`int aux = a * 7 – b;`
	`if (condicion) {`
	`  x = aux + 5;`
	`} else {`
	`  x = aux + 8;`
	`}`

Evitar código inalcanzable

Se denomina código inalcanzable o muerto al código que nunca se va a ejecutar, porque es imposible que se den las condiciones para ello. Por ejemplo:

Incorrecto	Correcto

`if (edad > 10) {`
`  ....`
`} else if (edad <= 10) {`
`  ....`
`} else { // rama inalcanzable`
`  ....`
`}`
	`if (edad > 10) {`
	`  ....`
	`} else if (edad <= 10) {`
	`  ....`
	`}`

Aspectos de eficiencia

Se debe procurar que el código sea eficiente, siempre que se consiga sin complicarlo excesivamente.

Evitar repeticiones de bucle innecesarias

Por ejemplo, si se está buscando un valor concreto dentro de un array, no se debe seguir recorriendo el array una vez que se ha encontrado dicho valor.

Incorrecto	Correcto

`int[] miArray;`
`......`
`boolean encontrado;`
`for (int i=0; i<max; i++) {`
`  if (miArray[i]==VALOR_BUSCADO) {`
`  encontrado = true;`
`  }`
`}`
`return encontrado;`
	`int[] miArray;`
	`......`
	`boolean encontrado;`
	`for (int i=0; i<max && !encontrado; i++) {`
	`  encontrado = miArray[i]==VALOR_BUSCADO);`
	`}`
	`return encontrado;`

Evitar repetir el mismo cálculo

No debe haber operaciones redundantes, es decir, operaciones que repiten el cálculo del mismo valor más de una vez. Si la primera vez que se calcula el valor se almacena en una variable auxiliar, ya no será necesario repetir el cálculo.

Incorrecto	Correcto

`factor = factorial(n) * 100 + 332;`
`......`
`if (factorial(n) < 2378) {`
`  ......`
`}`
	`int fn = factorial(n);`
	`......`
	`factor = fn * 100 + 332;`
	`......`
	`if (fn < 2378) {`
	`  ......`
	`}`