Técnicas de optimización de código

Ángel González M.

¿Por Qué optimizar código?

El primero es que si optimizas la manera en que escribes código se generará menos código intermedio (DEX code o byte code en Java) con lo cual la máquina virtual tendrá que transformar menos código a código máquina y la aplicación irá más rápida.

El segundo motivo es que como efecto colateral de optimizar el código, tu aplicación consumirá menos batería (en caso de dispositivos móviles o portátiles)

Tercer motivo gastaremos menos memoria RAM y el ordenador tendrá mas recursos disponibles.

Qué tenemos que optimizar

Factores a optimizar: (Ct/Ce)

- Coste temporal (Ct): tiempo de ejecución.
- Coste espacial (Ce): espacio de memoria RAM utilizado.

Optimizar preferiblemente

En las rutinas que se ejecutan con mayor frecuencia, es decir, en las que obedecen la regla 90-10, lo cual significa que siempre hay un 10% del código que se ejecuta el 90% del tiempo. Para identificar estas rutinas se pueden utilizar las siguientes técnicas:

- Sesiones de profiling: Procedimientos que se efectúan para de conocer cuanto tiempo demora la ejecución de un programa en cada instrucción.
- Sentido común: Depende principalmente de los conocimientos y experiencia del programador.

Técnicas de optimización

Ángel González M.

Pasar objetos por referencia mejor que por valor

Al pasar un objeto por referencia damos la posibilidad a la función de cambiar el valor de la variable pasada, por lo tanto si pasamos los objetos por referencia ahorraremos a la máquina de copiar una y otra vez el valor de un objeto.

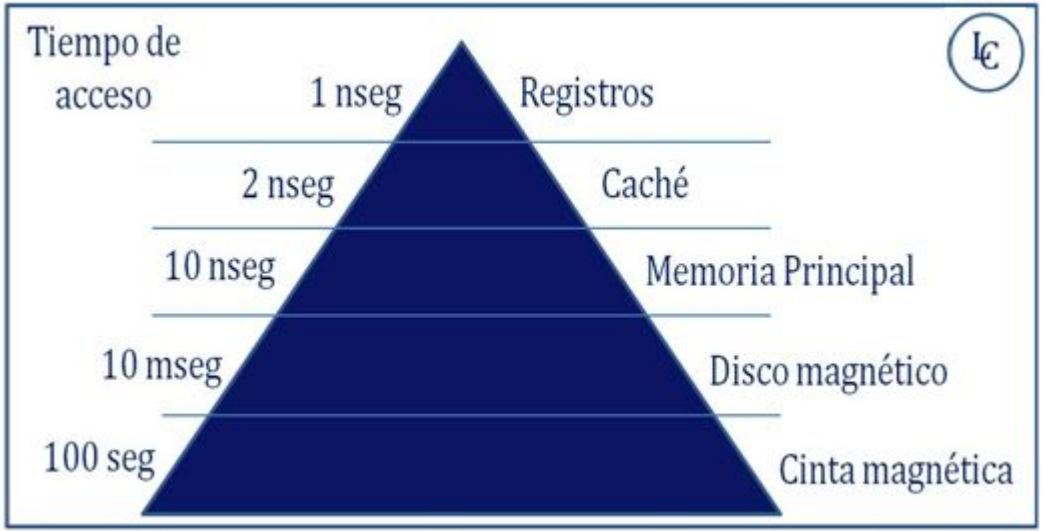
Recuerda que en Java, los tipos primitivos (int, short, long, boolean, double, float) se pasan por valor y los objetos se pasan por referencia.

```
public void miFuncion(int parametro, Alumno alum) {


}
```

Minimiza y optimiza el acceso a disco.

Manipular datos de los discos duros o de las memorias flash es mucho más lento que manipular datos almacenados en memoria RAM por eso si vas a manejar archivos ten en cuenta este punto.



Reutilización de expresiones comunes

Reutilización de expresiones comunes: regularmente tiende a favorecer el Ct.

Las expresiones(cálculos) que se repiten pueden agruparse en una variable y hacer ese cálculo una sola vez.

Ejemplo básico:

```
public class Ejemplo1 {
	void ejemplo1SinOptimizar() {
		int b=0,c=0,d=0;
		int a= b +c;
		d=a-d;
		int e=(b+c)*d;
	}

	void ejemplo1Optimizado() {
		int b=0,c=0,d=0;
		int a=b+c;
		d=a-d;
		int e=a*d;
	}
}
```

Ángel González M.

Reduce variables innecesarias.

Reducir declaración de variables innecesarias sobre todo dentro de bucles: tiende a favorecer tanto el Ct y Ce.

```
void ejemplo2SinOptimizar() {
        int valor=0,item;
        do {
            item=10;
            valor += valor + item;
        }while(valor<100);
    }

    void ejemplo2Optimizado() {
        int valor=0,item;
        item=10;
        do {
            valor += valor + item;
        }while(valor<100);
    }
```

Ángel González M.

Elimina código innecesario de los bucles

Es probablemente una de los errores que puede consumir más CPU, por supuesto sé más cuidadoso con aquellos bucles que tengan más iteraciones (Iteración significa repetir un proceso varias veces), con lo de sacar código no se trata de modificar la función final del bucle sino de optimizar al máximo su rendimiento, con esto queremos decir no incluir la declaración de una variable, no incluir comentarios, etc …

```
void ejemplo3SinOptimizar() {
        String total="";
        for(int i=0;i<10;i++) {
                // otras operaciones....
                String m = "Mensaje hola";
                int contador = i;
                total = m + contador;
        }
        System.out.println(total);
    }
```

```
void ejemplo3Optimizado() {
        String total="";
        String m = "Mensaje hola";
        int i;
        for(i=0;i<10;i++) {
                // otras operaciones....
        }
        total = m + (i-1);
        System.out.println(total);
}
```

Ángel González M.

Pre-calcular expresiones constantes

Tiende a favorecer el Ct.

Si hay algún cálculo que puedas pre-calcular en tiempo de codificación hará que tu código se ejecute más rápido

Ejemplo básico:

```
void ejemplo4SinOptimizar() {
        int i=2+3;
        int j=4;
        float f = j + 2.5f;

        /* aquí hay más código que usa i,j,f*/
        }

        void ejemplo4Optimizado() {
                int i=5;
                int j=4;
                float f = 6.5f;
        }
```

Pre-calcular expresiones constantes (continuación)

Si se van a realizar operaciones complejas como, senos, cosenos u otras operaciones complicadas de coma flotante y se sabe de antemano cuál va a ser el resultado, es conveniente pre-calcular estos valores y utilizarlos como constantes.

Ángulo	seno	coseno	tangente	Ángulo	seno	coseno	tangente
0º	0	1	0	46º	0,719	0,695	1,036
1º	0,018	1	0,018	47º	0,731	0,682	1,072
2º	0,035	0,999	0,035	48º	0,743	0,669	1,111
3º	0,052	0,999	0,052	49º	0,755	0,656	1,15
4º	0,07	0,998	0,07	50º	0,766	0,643	1,192
5º	0,087	0,996	0,088	51º	0,777	0,629	1,235
6º	0,105	0,995	0,105	52º	0,788	0,616	1,28
7º	0,122	0,993	0,123	53º	0,799	0,602	1,327
8º	0,139	0,99	0,141	54º	0,809	0,588	1,376
9º	0,156	0,988	0,158	55º	0,819	0,574	1,428
10º	0,174	0,985	0,176	56º	0,829	0,559	1,483
11º	0,191	0,982	0,194	57º	0,839	0,545	1,54
12º	0,208	0,978	0,213	58º	0,848	0,53	1,6
13º	0,225	0,974	0,231	59º	0,857	0,515	1,664
14º	0,242	0,97	0,249	60º	0,866	0,5	1,732
15º	0,259	0,966	0,268	61º	0,875	0,485	1,804
16º	0,276	0,961	0,287	62º	0,883	0,47	1,881
17º	0,292	0,956	0,306	63º	0,891	0,454	1,963
18º	0,309	0,951	0,325	64º	0,899	0,438	2,05
19º	0,326	0,946	0,344	65º	0,906	0,423	2,145
20º	0,342	0,94	0,364	66º	0,914	0,407	2,246
21º	0,358	0,934	0,384	67º	0,921	0,391	2,356
22º	0,375	0,927	0,404	68º	0,927	0,375	2,475
23º	0,391	0,921	0,425	69º	0,934	0,358	2,605
24º	0,407	0,914	0,445	70º	0,94	0,342	2,747
25º	0,423	0,906	0,466	71º	0,946	0,326	2,904
26º	0,438	0,899	0,488	72º	0,951	0,309	3,078
27º	0,454	0,891	0,51	73º	0,956	0,292	3,271
28º	0,47	0,883	0,532	74º	0,961	0,276	3,487
29º	0,485	0,875	0,554	75º	0,966	0,259	3,732
30º	0,5	0,866	0,577	76º	0,97	0,242	4,011
31º	0,515	0,857	0,601	77º	0,974	0,225	4,331
32º	0,53	0,848	0,625	78º	0,978	0,208	4,705
33º	0,545	0,839	0,649	79º	0,982	0,191	5,145
34º	0,559	0,829	0,675	80º	0,985	0,174	5,671
35º	0,574	0,819	0,7	81º	0,988	0,156	6,314
36º	0,588	0,809	0,727	82º	0,99	0,139	7,115
37º	0,602	0,799	0,754	83º	0,993	0,122	8,144
38º	0,616	0,788	0,781	84º	0,995	0,105	9,514
39º	0,629	0,777	0,81	85º	0,996	0,087	11,43
40º	0,643	0,766	0,839	86º	0,998	0,07	14,3
41º	0,656	0,755	0,869	87º	0,999	0,052	19,081
42º	0,669	0,743	0,9	88º	0,999	0,035	28,64
43º	0,682	0,731	0,933	89º	1	0,018	57,289
44º	0,695	0,719	0,966	90º	1	0	Inf.
45º	0,707	0,707	1				

Reducir propagación de copias

Tiende a favorecer el Ce. Ante instrucciones f = a, como se verá en el ejemplo básico, sustituir todos los usos de f por a.

Ejemplo básico:

```
void ejemplo5SinOptimizar() {
        int i=10, c=10, m=10;
        int a = 3 + i;
        int f=a;
        int b=f+c;
        int d=a+m;
        m=f+d;
}

void ejemplo5Optimizado() {
        int i=10, c=10, m=10;
        int a = 3 + i;
        //int f=a;
        int b=a+c;
        int d=a+m;
        m=a+d;
}
```

Métodos con pocos parámetros

Crear métodos con el menor número de parámetros posible. El código limpio dice que más de 2 puede que necesites revisar tu código.

Tiempo de acceso

La encriptación y conexiones https tienen peor rendimiento.

La apertura de las bases de datos también tienen un rendimiento pobre.

El acceso al disco duro también tiene un alto coste

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

        // Los accesos al disco duro son muy lentos, evítalos si puedes
        try {
                File f= new File("fichero.txt");
                FileInputStream fis = new FileInputStream(f);
                int valor = fis.read();
        } catch ( IOException e) {
                e.printStackTrace();
        }

}
```

Eliminar redundancias en acceso matrices

Eliminar redundancias en acceso matrices: tiende a favorecer el Ct.

Acceder a una matriz o a una colección ocupa tiempo, si vas a acceder varias veces es mejor guardar el resultado en una variable y acceder solo una vez

```
void ejemplo7SinOptimizar() {
        int i=1;
        float array[] = new float[5];
        array[i] = array[8+i] + (i+1)*5*8 + (5+1);
        array[i-1] = array[8+i] + (i+1) *5*8 + (6+1);
}

void ejemplo7Optimizado() {
        int i=1;
        float array[] = new float[5];
        float temporal = array[8+i];
        //tofix (i+1)*5*8 se puede a□adir a temporal
        //tofix potimizar calculos constantes 5+1 y 6+1
        array[i] = temporal + (i+1)*5*8 + (5+1);
        array[i-1] = temporal + (i+1) *5*8 + (6+1);
}
```

Ángel González M.

Eliminar redundancias en acceso matrices

Siempre que vayamos a acceder a la misma posición de un array varias veces, es mejor guardar esa posición en una variable local y así evitar el acceso repetido al índice del array.

```
void ejemplo8SinOptimizar() {
        int pos = 3;
        int contador = 0;
        int array[] = new int[5];
        for(int i=0;i<2000;i++) {
                //cosas
                contador = array[pos] + i;
                //cosas
        }
        System.out.println(contador);
        //cosas
    }
```

```
void ejemplo8Optimizado() {
        int pos = 3;
        int contador = 0;
        int array[] = new int[5];
        int temporal = array[pos];
        for(int i=0;i<2000;i++) {
                //cosas
                contador = temporal + i;
                //cosas
        }
        System.out.println(contador);
        //cosas
}
```

Ángel González M.

Mejor multiplicar que dividir

El ordenador es capaz de hacer más rápido las multiplicaciones que las divisiones.

```
void ejemplo9SinOptimizar() {
	int a = 10;
	float b = a / 2;
}

void ejemplo9Optimizado() {
	int a = 10;
	float b = a * 0.5f;
}
```

Código muerto

Eliminar el código muerto: tiende a favorecer tanto el Ct y Ce.

- O nunca ejecutados o inalcanzable,
- O si se ejecuta, su producción nunca se utiliza.

```
int ejemplo10SinOptimizar() {
        int x =10, y =20;
        int z = x/y;
        return x/y;
}

int ejemplo10Optimizado() {
        int x =10, y =20;
        return x/y;
}
```

Reutilización de variables

En ocasiones puedes reutilizar variables que has definido antes, en lugar de volver a crear nuevas variables.

```
void ejemplo11SinOptimizar() {
        int i = 0;
        while(i<10) {
                System.out.println("hola:"+i);
                i++;
        }
        int j = 0;
        while(j<10) {
                System.out.println("adios:"+j);
                j++;
        }
}

void ejemplo11Optimizado() {
        int i = 0;
        while(i<10) {
                System.out.println("hola:"+i);
                i++;
        }
        i=0;
        while(i<10) {
                System.out.println("adios:"+i);
                i++;
        }
}
```

Nombrar como corresponde

Poner nombres largos a las variables y métodos no hace que el programa sea más lento.

- **camelCase**: En esta, la primera letra de cada palabra nueva lleva mayúscula, a excepción de la primera. Ej unaFuncionCualquiera
- **Underscore**: En la cual cada palabra es separada por un underscore o guión bajo. Ej una_funcion_cualquiera
- **KebabCase**: Separamos cada palabra por un guión medio. Ej una-funcion-cualquiera
- **snake_case**: Las palabras se separan por guiones bajo (_) sin espacios, con normalmente todas las palabras en minúscula. Usado en ocasiones en Ruby en la definición de variables y métodos. En las librerias estandar de C y C++
- **Hungarian** (Systems) notation: iNumberOfPeople en esta notación precedemos con un caracter para indicar el tipo de variable (Ej i=integer)

- **Hungarian** (Apps) notation: cntNumberOfPeople: En este caso precedemos la variable de un prefijo indicando su aplicación: (Ej cnt = variable usada como contador)

```
void ejemplo9(){
        final int ESTO_ES_UNA_CONSTANTE = 3;
        int numeroDisparosPorSegundo = 7;  //camel case
        //int maximo-enemigos-pantalla = 9; // kebab case
        int frames_por_segundo=45;                    // underscore
}
void ejemplo9NoOptimizado(){
        int nds = 7;
        int mcp = 9;
        int fps = 45;
}
```

Ángel González M.

Demasiado a menudo intenta evitar la creación de objetos de Java

Los métodos de evitar a menudo objeto de nuevo en la llamada, en la práctica, porque el sistema no sólo necesita tiempo para crear un objeto, sino también el tiempo de recuperación y tratamiento de residuos a estos objetos, en el contexto de nuestro control, el límite máximo de objeto de reutilización, mejor que con el tipo de datos o la sustitución de la matriz básica de objetos.

Trata de usar variables locales

Es mejor usar variable locales que variables globales.

```
public class Ejemplo13 {
	int j=5;
	void ejemplo13SinOptimizar() {
		int i = 10;
		int v = i+j;
	}

	void ejemplo13Optimizado() {
		int j=5;
		int i = 10;
		int v = i+j;
	}
}
```

Ángel González M.

Evita el uso de métodos sincronizados (si es posible)

Usamos métodos sincronizados cuando creamos aplicaciones multi-hilo y así evitamos que 2 o más hilos accedan a la vez a un método (con todos los problemas que puede acarrear esto).
Pero sincronizar método penaliza haciendo que se ejecute más lentamente el código, así que usa métodos sincronizados solamente cuando sea necesario.

```
synchronized void ejemplo14SinOptimizar() {
        int i=0;
        int c=4;
}

void ejemplo14Optimizado() {

}
```

Evita crear variables innecesarias

```
void ejemplo15SinOptimizar() {
        int i=5;
        ArrayList lista = new ArrayList();
        Elemento e = new Elemento()
        if(i==1) {
                lista.add(v);
        }
}

void ejemplo15Optimizado() {
        int i=5;
        ArrayList lista = new ArrayList();
        if(i==1) {
                lista.add( new Elemento() );
        }
}
```

Evita el uso de expresiones complejas en las condiciones del bucle

```
void ejemplo17SinOptimizar() {
	ArrayList lista = new ArrayList();
	for(int i=0;i< lista.size();i++) {
		// hacemos cosas
	}
}

/*TOFIX: usar metodos mejores para recorrer colecciones*/
void ejemplo17Optimizado() {
	ArrayList lista = new ArrayList();
	int tamannoLista = lista.size();
	for(int i=0;i< tamannoLista;i++) {
		// hacemos cosas
	}
}
```

Usar las clases adecuadas en cada momento

Por ejemplo, para guardar algo en una colección, existen multitud de clases que puedes usar, es muy recomendable usar la ***adecuada,*** porque unas son más lentas que otras; o tienen unas características que otras no tienen.

```
void ejemplo18SinOptimizar() {
		ArrayList datos = new ArrayList();
		for(int i=0;i<10000;i++) {
			datos.add(i);
		}
	}

	/*El objetivo es almacenar cosas lo m�s r�pido posible*/
	void ejemplo18OptimizadoA() {
		HashSet datos = new HashSet();
		for(int i=0;i<10000;i++) {
			datos.add(i);
		}
	}
	/*Acceder a la informaci�n lo m�s rapido posible*/
	void ejemplo18OptimizadoB() {
		TreeSet datos = new TreeSet();
		for(int i=0;i<10000;i++) {
			datos.add(i);
		}
	}
```

Usar ArrayList no siempre es lo más eficiente. Existen otras estructuras de datos como los hash o los árboles que en ciertas circunstancias son mucho más óptimos.

Usar desplazamiento de bits en lugar de divisiones

```
void ejemplo11(){
        int a = 44;
        int num1 = a / 2;
        int num1 = a / 4;
        int num2 = a / 8;
        int num3 = a * 4;
        int num4 = a * 8;
    }
    void ejemplo11Optimizado(){
        int a = 44;
        int num1 = a >> 1;
        int num1 = a >> 2;
        int num2 = a >> 3;
        int num3 = a << 2;
        int num4 = a << 3;
    }
```

Utilizar desplazamiento de bits en vez de la multiplicación o división si es posible.

Por ejemplo: x >> 2 es equivalente a x / 4

x << 10 es equivalente a x * 1024

1 << 20 es equivalente a Math.pow(2, 20).

Apertura y cierre de ficheros, conexiones de base de datos y de red.

Puedes optimizar mucho una aplicación abriendo y cerrando en el momento adecuado las conexiones a bases de datos, recursos de red y ficheros.

```
void guardarEnFicheroSinOptimizar(String ruta, int valor) {
        // Abrir el fichero (ruta)
        // guardar valor dentro del fichero
        // cerrar el fichero
}
void ejemploSinOptimizar() {
        for(int i=0;i<1000;i++) {
                guardarEnFicheroSinOptimizar("miFichero.txt", i);
        }
}
```

```
void guardarEnFicheroOptimizado(File fichero, int valor) {
        // guardar valor dentro del fichero
}
void ejemploOptimizado() {
        // Abrir la conexi□n con el fichero
        File fichero = null;
        for(int i=0;i<1000;i++) {
                guardarEnFicheroOptimizado(fichero, i);
        }
        // Cerrar la conexion con el fichero
}
```

Salir del bucle cuando sea necesario

Cuando recorremos algo, no siempre es necesario llegar hasta el final, puedes finalizar el bucle cuando lo estimes oportuno con **break** o usando **continue**

```
boolean ejemplo20SinOptimizar() {
		ArrayList palabras = new ArrayList();
		String palabraABuscar = "Ana";

		boolean encontrado = false;
		for(int i=0;i<palabras.size();i++) {
			if(palabras.get(i) == palabraABuscar) {
				encontrado = true;
			}
		}
		return encontrado;
	}
```

```
boolean ejemplo20Optimizado() {
		ArrayList palabras = new ArrayList();
		String palabraABuscar = "Ana";

		//boolean encontrado = false;
		for(int i=0;i<palabras.size();i++) {
			if(palabras.get(i).equals(palabraABuscar) ) {
				return true;
			}
		}
		return false;
	}
```

Evita el uso de variables innecesarias dentro de los bucles

```
void ejemplo21SinOptimizar() {
	int item = 0;
	int valor = 0;
	do {
		item=10;
		valor = valor + item;
	}while(valor<100);
}

void ejemplo21Optimizado() {
	int item = 0;
	int valor = 0;
	item=10;
	do {
		valor = valor + item;
	}while(valor<100);
}
```

Concatenar String

Evitar concatenaciones de Strings, utilizar StringBuffer(sincronizado) o StringBuilder para tal caso. La concatenación de Strings produce cada vez un nuevo objeto y por tanto, mayor consumo de memoria y mayor recolección de basura. (Por favor, esta hacedla siempre)

```
void ejemplo22SinOptimizar() {
        String nombres = "";
        nombres = nombres + "Angel";
        nombres = nombres + "Rosa";
        nombres = nombres + "Paula";
        nombres += "Juan";
}
```

```
// Mejoramos el con StringBuffer (sincroniced).
// Lo usaremos solo si estoy creando aplicaciones con varios hilos
void ejemplo22OptimizadoA() {
        StringBuffer sb = new StringBuffer();
        sb.append("Angel");
        sb.append("Rosa");
        sb.append("Paula");
        sb.append("Juan");
}
// Mejoramos el con StringBuffer (no sincroniced)
// Lo usaremos si no estoy usando hilos
void ejemplo22OptimizadoB() {
        StringBuilder sb = new StringBuilder();
        sb.append("Angel");
        sb.append("Rosa");
        sb.append("Paula");
        sb.append("Juan");
}
```

Pool de objetos

Reutilizar y hacer pool de objetos siempre que sea posible para evitar crear nuevas instancias.

Ver el patrón de diseño objet pool

Liberar recursos tan pronto como sea posible

Liberar recursos tan pronto como sea posible, como conexiones de red, a streams o a ficheros. Normalmente, se suele liberar este tipo de recursos dentro de la cláusula finally para asegurarnos de que los recursos se liberan aún cuando se produzca alguna excepción.

Referenciar a null

Referenciar a null instancias de objetos que ya no se van a usar, para que el recolector de basura libere memoria.

```
public class ejemplo24 {
	void ejemplo24SinOptimizar() {
		Alumno pepe = new Alumno();
		// trabajo con pepe
		Profesor juan = new Profesor();
		// trabajo con juan
	}

	void ejemplo24Optimizado() {
		Alumno pepe = new Alumno();
		// trabajo con pepe
		pepe=null;
		Profesor juan = new Profesor();
		// trabajo con juan
		juan=null;

	}
}

class Alumno{

}
class Profesor{

}
```

Usar buffers

Utilizar buffers para leer datos a través de la red y leer los datos en porciones en lugar de byte a byte que es más lento.

```
void ejemplo25SinOptimizar() {
        File fichero = new File("miFichero.txt");
        try {
            FileReader fr = new FileReader(fichero);
            int miByte1 = fr.read();
            int miByte2 = fr.read();
            int miByte3 = fr.read();
            //tofix colocar el cierre del fichero en el finally
            fr.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
void ejemplo25Optimizado() {
        File fichero = new File("miFichero.txt");
        try {
            FileReader fr = new FileReader(fichero);
            BufferedReader br = new BufferedReader(fr);
            String linea = br.readLine();
            //tofix colocar el
```

Evitar métodos sincronizados, usar métodos estáticos

Los métodos sincronizados son los más lentos, a continuación los métodos de interfaz, los métodos de instancia, los métodos finales y por último los métodos estáticos son los más rápidos. Hay que tener en cuenta esta clasificación para evitar siempre que sea posible la sincronización e interfaces.

Evitar métodos sincronizados en los bucles

Evitar en cualquier caso sincronización dentro de bucles.

Variables, arrays o colecciones

Usar variables es más eficiente que arrays.

Los arrays son más eficientes que Vector o HashTable y en cualquier caso, arrays unidimensionales siempre mejor que bidimensionales.

Tener en cuenta también que hay que inicializar la clase Vector y HashTable con un tamaño que se ajuste a nuestras necesidades.

Atributos VS getters y setters

El acceso a los atributos de una clase es más rápido que encapsular con getter y setter.

Contar hacia atrás

Contar hacia atrás es más rápido en los bucles.

Porque el ciclo ya no tiene que evaluar una propiedad cada vez que se comprueba para ver si está terminado y simplemente se compara con el valor numérico final.

Es decir

for(int i = array.length-1; i!=-1 ;i--)
Evalúa .length solo una vez, cuando declara i , mientras que para este ciclo

for(int i = 0; i < array.length; i++)

evalúa .length cada vez que incrementa i , cuando comprueba si i <= array.length

Ángel González M.

Evitar bucles

Cuando sea posible evitar bucles ya que evitaremos toda la sobrecarga de control de flujo en cada iteración. Por ejemplo, si tenemos una operación que se va a realizar 5 veces, en vez de utilizar un bucle podemos realizar las 5 operaciones secuencialmente.

(nota: esto ensuciará mucho nuestro código)

Usar tipos básico de datos (int, float, boolean, ...) en lugar de objetos Java (Integer, Float, Boolean, ...) siempre que sea posible.

Perclipse

https://github.com/sebastiangraf/perclipse