

Best Practices



Carlos Yáñez García

Intro

- Siempre trabajamos en grupo
- Hay que tener unas bases comunes para poder hacer código homogéneo
- Para cuando un programador hace buen código ya es senior y pasa a hacer powerpoints (como esta)
- Casi todos los programadores tienen < 5 años de experiencia
- Donde se pierde más tiempo es en revisar código (propio o ajeno)
- 25% del tiempo en corregir errores (80% manteniendo código)
- Rara vez una misma persona o grupo sigue manteniendo el software una vez desarrollado

Comentarios

- Los comentarios son tan importantes como el código
- Memoria de pez: pasado un tiempo nadie se acuerda de nada
- Tenemos herramientas de control de cambios con los registros de los cambios
- Los comentarios deben aclarar
- O se hace cuando se está escribiendo el código o te olvidas
- Se debe aclarar el propósito del código
- “El mejor código es aquel que se entiende sin necesidad de comentarios”



Dónde comentar

En general:

- ifs, bucles, operaciones con la BD...

Funciones y métodos:

- Qué hace
- Qué son los parámetros que recibe
- Qué devuelve

Classes:

- Cuál es el propósito de la clase / Qué hace la clase

```
/**
 * Simple HelloButton() method.
 * @version 1.0
 * @author john doe <doe.j@example.com>
 */
HelloButton()
{
    JButton hello = new JButton( "Hello, wor
    hello.addActionListener( new HelloBtnList

    // use the JFrame type until support for t
    // new component is finished
    JFrame frame = new JFrame( "Hello Button"
    Container pane = frame.getContentPane();
    pane.add( hello );
    frame.pack();
    frame.show();           // display the fra
}
```

Comentarios

- Los comentarios deben contener sólo la información que es relevante para la lectura y la comprensión del programa. Existen dos tipos de comentarios: de documentación y de implementación.
- Los comentarios de documentación están destinados a describir la especificación del código. Se utilizan para describir las clases Java, los interfaces, los constructores, los métodos y los campos. Debe aparecer justo antes de la declaración. Existe una herramienta llamada Javadoc que genera páginas HTML partiendo de este tipo de comentarios.
- Los comentarios de implementación son para comentar algo acerca de la aplicación particular. Pueden ser de tres tipos: de bloque, de línea y corto
- Hay que comentar en su justa medida: No nos cuentes tu vida
- Es un mensaje a otro programador

Comentarios

- De bloque:

```
/*  
 * Here is a block comment.  
 */
```

- De línea (nadie lo hace así):

```
if (condition) {  
    /* Handle the condition. */  
    ...  
}
```

Comentarios

- De final de línea:

```
if (foo > 1) {  
    // Do a double-flip.  
    ...  
}  
else  
    return false;           // Explain why here.  
  
//if (bar > 1) {  
//  
//    // Do a triple-flip.  
//    ...  
//}  
//else  
//    return false;
```

Naming Conventions

- Los nombres claros dan legibilidad
- Nombres cortos
- CTRL + F
- Si nombramos las cosas de forma consistente, es fácil encontrar las cosas
- Hay que homogeneizar los nombres dentro del workgroup
- Es posible incluso que la empresa tenga una convenciones propias
- Cada archivo generado tiene que tener un nombre descriptivo
- Cada carpeta del proyecto debe tener un nombre descriptivo y las subcarpetas que sean necesarias.
- Hazlo entendible!!!

Naming Conventions

- Es posible incluso que la empresa tenga una convenciones propias
- Hay que tener convenciones para todo: clases, métodos, variables, librerías...
- Cada lenguaje ya tendrá unas convenciones propias
- Siempre tratar de ser consistentes
- Camel case vs Snake Case



Naming Conventions

- Los nombres de las variables, métodos, clases, etc., hacen que los programas sean más fáciles de leer ya que pueden darnos información acerca de su función. Las normas para asignar nombres son las siguientes:

- 1) **Paquetes:** el nombre se escribe en minúscula, se pueden utilizar puntos para reflejar algún tipo de organización jerárquica.

Ejemplo:java.io.

- 2) **Clases:** los nombres deben ser sustantivos. Se deben utilizar nombres descriptivos. Si el nombre está formado por varias palabras la primera letra de cada palabra debe estar en mayúscula:

Ejemplo: HiloServidor.

Naming Conventions

3) **Métodos:** se deben usar verbos en infinitivo. Si está formado por varias palabras el verbo debe estar en minúscula y la siguiente palabra debe empezar en mayúscula.

Ejemplo: ejecutar(), asignarDestino().

4) **Variables:** deben ser cortas (pueden ser de una letra) y significativas. Si son varias palabras la primera debe estar en minúscula.

Ejemplos: i,j, sumaTotal.

5) **Constantes:** el nombre debe ser descriptivo. Se escriben en mayúsculas y si son varias palabras van unidas por un carácter de subrayado.

Ejemplo: MAX_VALOR.

Naming: Palabros reservados

break

case

catch

const

continue

debugger

default

delete

do

else

finally

for

function

if

in

instanceof

new

return

switch

this

throw

try

typeof

var

void

while

with

Code Format

Las herramientas utilizadas para el desarrollo de los programas suelen ayudar a formatear correctamente el código.

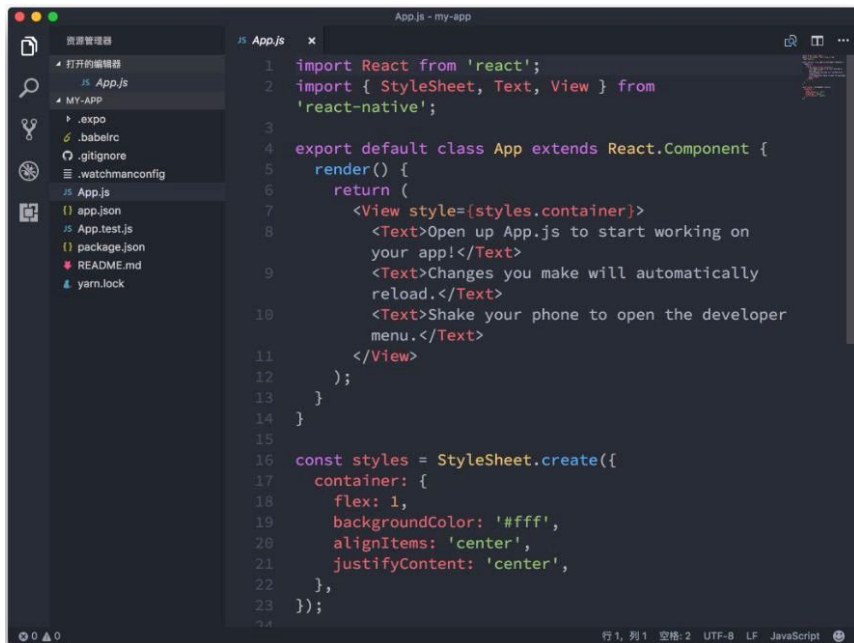
Por ejemplo, desde cualquier entorno se puede pulsar en la opción de menú para formatear el código

Una vez generado el código fuente es necesario interpretarlo para ejecutarlo o traducirlo a un lenguaje que sea entendido por la máquina, para ello se utilizan otros programas llamados intérpretes o compiladores.

Validador de código: <http://www.jshint.com/>

Estilo

- La mejor opción es elegir una estándar y ceñirse a ella.



```
1 import React from 'react';
2 import { StyleSheet, Text, View } from
  'react-native';
3
4 export default class App extends React.Component {
5   render() {
6     return (
7       <View style={styles.container}>
8         <Text>Open up App.js to start working on
          your app!</Text>
9         <Text>Changes you make will automatically
          reload.</Text>
10        <Text>Shake your phone to open the developer
          menu.</Text>
11      </View>
12    );
13  }
14 }
15
16 const styles = StyleSheet.create({
17   container: {
18     flex: 1,
19     backgroundColor: '#fff',
20     alignItems: 'center',
21     justifyContent: 'center',
22   },
23 });
```

Indentación

- Como norma general se usarán cuatro espacios un tabulador.
- La longitud de las líneas de código no debe superar 80 caracteres.
- La longitud de las líneas de comentarios no debe superar 70 caracteres.
- Cuando una expresión no cabe en una sola línea: romper después de una coma, romper antes de un operador, alinear la nueva línea al principio de la anterior

```
longName1 = longName2 * (longName3 + longName4 - longName5)
           + 4 * longname6; // PREFER
```

```
longName1 = longName2 * (longName3 + longName4  
                        - longName5) + 4 * longname6; // AVOID
```

Separaciones

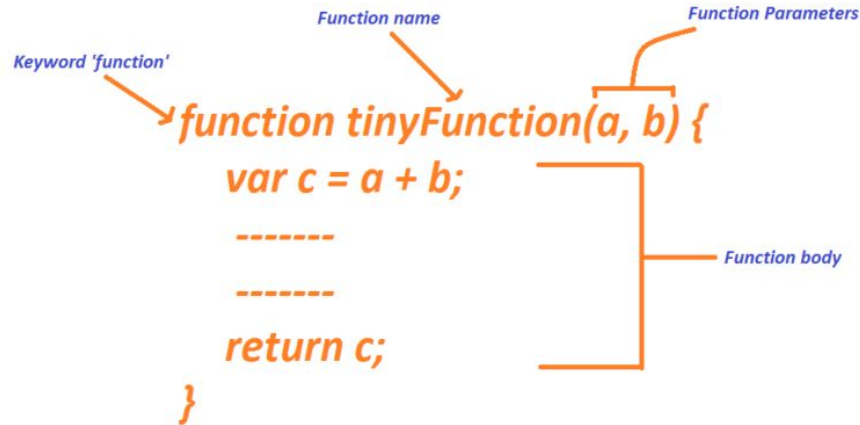
- Mejoran la legibilidad del código. Se utilizan:
 - Dos líneas en blanco: entre las definiciones de clases e interfaces.
 - Una línea en blanco: entre los métodos, la definición de las variables locales de un método y la primera instrucción, antes de un comentario, entre secciones lógicas dentro de un método para mejorar la legibilidad.
- Un carácter en blanco: después de una coma, los operadores binarios menos el punto, las expresiones del for...

Variables

- Se recomienda declarar una variable por línea.
- Inicializar las variables locales donde están declaradas y colocarlas al comienzo del bloque
- No usar variables globales
- Malos nombre de variables:
 - Demasiado cortos dan poca información: x1, fe2, a1...
 - Demasiado largos: contadorDelIncrementosBuclePrincipal...
- Mejor en inglés: evita tildes, diéresis, la letra ñ...
- Declarar una sola variable por línea!
- Inicializarlas cuando se declaran, aunque sea a null.

Funciones

- Una función debe hacer una sola cosa
- Una función debe poder ser reutilizada
- No repetir código: usar funciones
- Sólo debe haber un return
- El return no debe tener paréntesis: **return (dato + 1)**



Sentencias

- Cada línea debe contener una sentencia.
- Si hay un bloque de sentencias, este debe ser sangrado con respecto a la sentencia que lo genera y debe estar entre llaves aunque solo tenga una sentencia.
- if - else: Definen bloques a los que se aplican las normas anteriores. Todos los bloques tienen el mismo nivel de sangrado.
- Bucles. Definen un bloque, que sigue las normas anteriores. Si el bucle está vacío no se abren ni se cierran llaves.
- Las sentencias return no deben usar paréntesis

Archivos de Código

```
> class Vehicle {
  constructor(make, model, color) {
    this.make = make;
    this.model = model;
    this.color = color;
  }

  getName() {
    return this.make + " " + this.model;
  }
}

class Car extends Vehicle{
  getName(){
    return super.getName() +" - called base class function from child class.";
  }
}

let car = new Car("Honda", "Accord", "Purple");

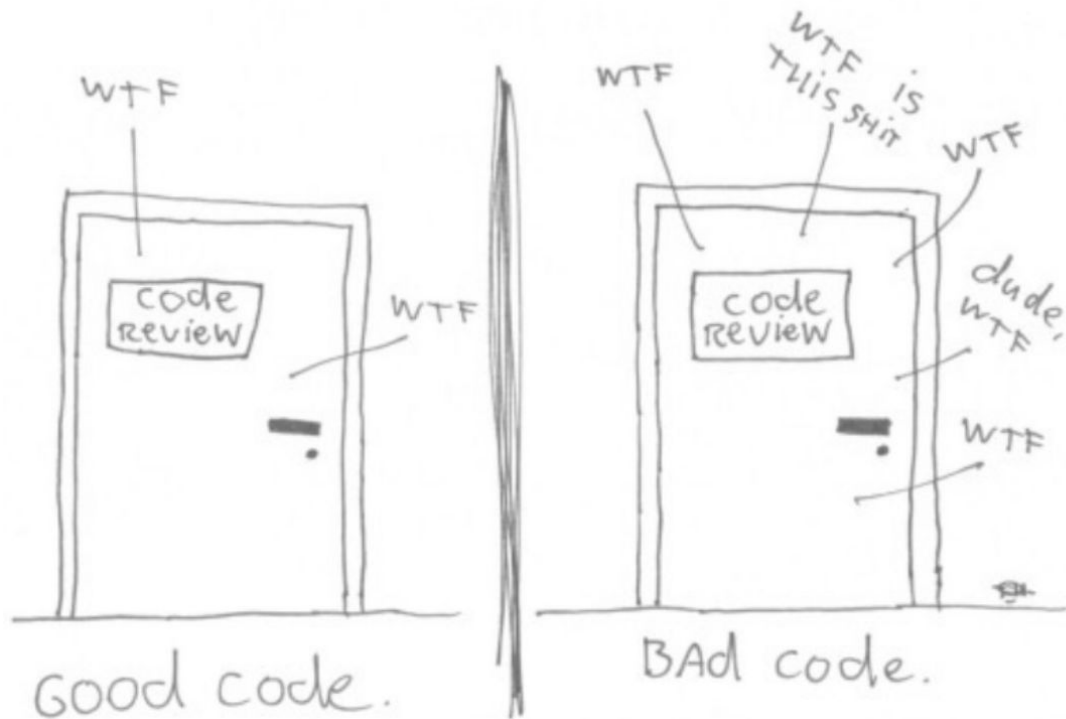
car.getName();
< "Honda Accord - called base class function from child class."
```

- Cada archivo debe contener una sola clase. Cada clase debe tener su propio fichero.
- Comentarios. Todos los ficheros fuente deben comenzar con un comentario que muestra el nombre de la clase, información de la versión, la fecha, y el aviso de derechos de autor.
- Una clase deberá estar en un archivo ella sola!

Código Modular

- Funciones pequeñas que hagan una sola cosa
- Mantenimiento sencillo
- Pruebas sencillas
- Una función larga es síntoma de código malo
- Se separan las funciones en archivos
- Reutilizar las funciones
- Errores en cascada: una función que llama a otra función, que llama a otra función...
- El diseño ya debe fomentar el desarrollo modular

The ONLY valid measurement
of code quality: WTFs/minute



Bucles

- Examinar con detenimiento por si hubiese bucles infinitos
- Un sólo bucle infinito tira un server
- Revisar bien las variables, su scope y sus valores antes y después del bucle (reseteo)
- Tratar de no hacer más de 3 niveles de bucles (i, j, k)
- Cortar los bucles cuando se pueda
- No usar break



Exceptions

- Algunas generadas por el programador, otras por el sistema
- try-catch
- Hay que estar preparado para que falle cualquier cosa en cualquier momento
- Archivos: no existe, ya está abierto, no tiene permisos...
- DB: la DB está caída, no hay conexiones disponibles...
- ...

Performance

- Tener un ojo puesto en CPU, memoria y red
- La memoria la ocupan las variables (objetos, arrays, Strings, Dates...)
- La CPU se consume con archivos, DBs y dispositivos
- DB: Tratar de hacer consultas eficientes, usar ids, indexación...
- Se debe liberar la memoria: poner variables a null al acabar



Lógica del programa

- No saltar a escribir código directamente.
- Pararse un poco a pensar primero
- Hacer esbozos en un papel para aclarar las ideas
- Se ahorra tiempo pensando, peor es equivocarse y tener corregir.
- Intuitivamente damos muchas cosas por hechas que luego no son así
- Comenta con alguien la lógica que vas a aplicar
- Documentar el código: comentarios diagramas

