CONNECT! & WHERE IS MY FRIEND?
ESTÁNDAR DE IMPLEMENTACIÓN
VERSIÓN 3.0

Historia de revisiones

Fecha	Versión	Descripción	Autor
28/08/2013	1.0	Primera versión	Nicolás Díaz
09/10/2013	2.0	Agregado JavaScript	Emiliano Conti
23/11/2013	3.0	Look & Feel	Leonardo Clavijo

CONTENIDO

1. CONVENCIONES GENERALES

Dado que la implementación será en varios lenguajes, se definirá un standard para cada lenguaje a utilizar. Algunas de estas pautas son compartidas por todos los lenguajes por lo que se incluirán en esta sección.

1.1. BLOQUES DE CÓDIGO

Los bloques de código entre llaves serán indentados en forma de bloque, con las llaves de apertura y cerrado en el mismo nivel de tabulación.

Incorrecto:

```
if (a > 0) {
 Console.WriteLine("hola");
  }
```

Correcto:

```
if (a > 0)
{
 Console.WriteLine("hola");
}
```

1.2. SENTENCIAS

Todas las sentencias iterativas como condicionales deberán encontrarse dentro de un bloque de código definido entre llaves.

Incorrecto:

```
if (a > 0) Console.WriteLine("hola");
```

Correcto:

```
if (a > 0)
{
 Console.WriteLine("hola");
}
```

1.3. COMENTARIOS

Los comentarios entre código (es decir, los que comentan una o varias sentencias de código) se agregarán antes de las sentencias comentadas y no a continuación de alguna de las líneas propiamente dichas.

Si el comentario requiere más de una línea, se utilizarán // al comienzo de cada línea y nunca /*

Incorrecto (comentario a continuación de una sentencia):

```
if (a > 0) // Si a es mayor que 0
{
 Console.WriteLine("hola");
  }
```

Incorrecto (usa /* */):

```
/* Si a es mayor que 0
 entonces imprimo la línea */
if (a > 0)
{
 Console.WriteLine("hola");
  }
```

Correcto:

```
// Si a es mayor que 0
// entonces imprimo la línea
if (a > 0)
{
 Console.WriteLine("hola");
 }
```

1.4. CLASES

Para el nombrado de clases se utilizará en todos los lenguajes el mismo estándar, comenzando siempre con letra mayúscula y separando (si aplica) las palabras que definen la clase por cambio de minúscula a mayúscula.

Incorrecto (comienza con minuscula):

```
class miClase
{

}
```

Incorrecto (no cambia de min a mayúscula al comenzar otra palabra):

```
class Miclase
{

}
```

Correcto:

```
class MiClase
{

}
```

1.5. VARIABLES

Las variables tanto de clase (atributos) como en el cuerpo de los métodos comenzarán con minúscula, podrán tener letras y números (nunca símbolos), y la separación entre palabras que forman el concepto comenzará con mayúscula como en las clases.

Incorrecto (comienza con mayúscula):

```
int Variable = 7;
```

Incorrecto (separación entre palabras en minúscula):

```
int mivariable = 7;
```

Correcto:

```
int miVariable = 7;
```

1.6. CONSTANTES

Las constantes se nombrarán completamente en mayúscula, y se utilizarán underscores (_) para separar las palabras que componen el concepto.

Incorrecto (no esta toda en minuscula):

```
const int miConstante = 0;
```

Incorrecto (sin separación entre palabras):

```
const int MICONSTANTE = 0;
```

Correcto:

```
const int MI_CONSTANTE = 0;
```

1.7. INDENTACIÓN GENERAL

Los bloques de código deberán ser indentados de acuerdo a la jerarquía a la que pertenecen con TAB.

Incorrecto:

```
class MiClase
{
void MetodoEjemplo()
{
if (a > 0)
{
Console.WriteLine("hola");
}
}
}
```

Correcto:

```
class MiClase
{
  void MetodoEjemplo()
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

2. CONVENCIONES ESPECÍFICAS C# Y C++ DE VISUAL STUDIO .NET

2.1. NOMBRES DE OPERACIONES

Los nombres de las operaciones deberán utilizar exclusivamente letras o números, nunca símbolos, y deberá comenzar en mayúscula, y cambiará a mayúscula en cada nombre que la componga.

Incorrecto (comienza con minúscula):

```
class MiClase
{
  void metodoEjemplo()
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

Incorrecto (no cambia a mayúscula al cambiar palabra):

```
class MiClase
{
  void Metodoejemplo()
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

Incorrecto (tiene caracteres no alfanuméricos):

```
class MiClase
{
  void Metodo_Ejemplo()
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

Correcto:

```
class MiClase
{
  void MetodoEjemplo()
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

2.2. COMENTARIOS DE OPERACIONES Y CLASES

Las operaciones y clases deberán estar comentadas antes de la definición de la misma, con un comentario cuyas líneas comienzan con /// y tienen una serie de definiciones que la propia IDE generará y se deberá rellenar para todos los parámetros.

Incorrecto (no usa comentarios de VS) :

```
class MiClase
{
  // Método de ejemplo
  void MetodoEjemplo()
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

Incorrecto (El comentario del método no define la variable a) :

```
class MiClase
{
   /// <summary>
/// Método de ejemplo
/// </summary>
    /// <param name="a"></param>
  void MetodoEjemplo(int a)
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

Correcto:

```
   /// <summary>
/// Clase de ejemplo
/// </summary>
   class MiClase
   {
      /// <summary>
   /// Método de ejemplo
   /// </summary>
       /// <param name="a">Variable de decisión</param>
     void MetodoEjemplo(int a)
     {
       if (a > 0)
       {
         Console.WriteLine("hola");
       }
     }
   }
```

2.3. TIPOS DE DATOS BÁSICOS (C#)

Los tipos de datos básicos de C# deberán ser utilizados con su nomenclatura unboxed, es decir el data type y no la clase que lo representa.

Incorrecto (se utiliza la clase Int32 en vez del tipo básico int):

```
class MiClase
{
  void MetodoEjemplo(Int32 a)
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

Correcto:

```
class MiClase
{
  void MetodoEjemplo(int a)
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

3. CONVENCIONES ESPECÍFICAS JAVA

3.1. NOMBRES DE OPERACIONES

Los nombres de las operaciones deberán utilizar exclusivamente letras o números, nunca símbolos, y deberá comenzar en minúscula, y cambiará a mayúscula en cada nombre que la componga.

Incorrecto (el nombre de la operación comienza con mayúscula):

```
public class MiClase
{
  void MetodoEjemplo(boolean a)
  {
    if (a)
    {
      Console.WriteLine("hola");
    }
  }
}
```

Incorrecto (tiene caracteres no alfanuméricos):

```
class MiClase
{
  void metodo_ejemplo(boolean a)
  {
    if (a > 0)
    {
      Console.WriteLine("hola");
    }
  }
}
```

Correcto:

```
public class MiClase
{
  void metodoEjemplo(boolean a)
  {
    if (a)
    {
      Console.WriteLine("hola");
    }
  }
}
```

3.2. COMENTARIOS DE OPERACIONES Y CLASES

Las operaciones y clases deberán estar comentadas utilizando el estándar javadoc, antes de la definición de la misma, con un comentario cuyas líneas están entre /* */, y tienen una serie de definiciones para cada parámetro que se deberá rellenar para todos los parámetros.

Incorrecto (no brinda información sobre el parámetro a):

```
/**
 * Método de ejemplo
 *
 * @return retorna -1 si a es TRUE
 */
int MetodoEjemplo(boolean a)
{
    if (a)
  {
    return -1;
  }
}
```

Correcto:

```
/**
 * Método de ejemplo
 *
 * @param  a  indica si la condición es válida
 * @return retorna -1 si a es TRUE
 */
int MetodoEjemplo(boolean a)
{
    if (a)
  {
    return -1;
  }
}
```

4. CONVENCIONES ESPECÍFICAS JAVASCRIPT

4.1. NOMBRES DE OPERACIONES

Los nombres de las operaciones deberán utilizar exclusivamente letras o números, nunca símbolos, y deberá comenzar en minúscula, y cambiará a mayúscula en cada nombre que la componga.

Incorrecto (el nombre de la operación comienza con mayúscula):

```
function MetodoEjemplo(a)
{
  if (a)
  {
    Console.Log("hola");
  }
}
```

Incorrecto (tiene caracteres no alfanuméricos):

```
function metodo_ejemplo(a)
{
  if (a > 0)
  {
    Console.Log("hola");
  }
}
```

Correcto:

```
function metodoEjemplo(a)
{
  if (a)
  {
    Console.Log("hola");
  }
}
```

4.2. COMENTARIOS DE OPERACIONES Y CLASES

Las operaciones y clases deberán estar comentadas antes de la definición de la misma, con un comentario cuyas líneas están entre /* */, y tienen una serie de definiciones para cada parámetro que se deberá rellenar para todos los parámetros. Además se debe indicar qué tipo de parámetros se espera que reciba la operación así como una descripción. Además en caso de que uno de los parámetros de entrada o salida sea un objeto JSON, se deberá describir de esta misma manera los componentes de dicho objeto.

Incorrecto (no brinda información sobre el parámetro a):

```
/**
 * Método de ejemplo
 *
 * @return retorna -1 si a es TRUE
 */
function metodoEjemplo(a)
{
    if (a)
  {
    return -1;
  }
}
```

Correcto:

```
/**
 * Método de ejemplo
 *
 * @param  a   es un objeto JSON compuesto por un booleano con identificador  *
 “condición” que indica si la condición es válida
 *
 * @return      retorna -1 si la componente “condición” del objeto pasado  *
 *como parámetro es es TRUE
 */
function metodoEjemplo(a)
{
    if (a[“condicion”])
  {
    return -1;
  }
}
```

4.3. ACCESO A OBJETOS JSON

Los objetos JSON son muy flexibles y pueden ser utilizados de múltiples maneras. Sin embargo, para homogeneizar el código se deberán acceder mediante la misma notación con la que se accede a un arreglo. Es decir, mediante los corchetes (“[]”). Además los identificadores dentro de un objeto JSON serán escritos como etiquetas y no como strings.

Incorrecto (Usa la notación del punto):

```
var json = {
  clave = “un valor para la clave”
};
Console.log(json.clave);
```

Correcto:

```
var json = {
  clave = “un valor para la clave”
};
Console.log(json[“clave”]);
```

Incorrecto (Define un JSON usando un string para el identificador):

```
var json = {
  “identificador” : “un valor para el identificador”
};
```

Correcto:

```
var json = {
  identificador : “un valor para el identificador”
};
```

4.4. USO DE LITERALES

Cuando se utilice una literal para, por ejemplo definir un string u obtener un atributo de un objeto, se utilizarán las comillas simples (‘). Esto se hace para homogeneizar el código de modo que no hayan lugares con una comilla simple (‘) y otros con comilla doble (“).

4.5. CONSIDERACIONES DE PERFORMANCE

Las siguientes prácticas son útiles para mejorar la performance del código JavaScript.

- **Mantener Variables que Apunten a Elementos del DOM:** Si se accede a un nodo del DOM varias veces en el mismo método, se debe declarar una variable que contenga dicho nodo y utilizarla. No se debe buscar el nodo varias veces en el DOM para evitar recorrerlo innecesariamente:

  Incorrecto (No guarda el objeto en una variable):

```
document.getElementById("id_elem").value = 5;
document.getElementById("id_elem").text = "5";
```

  Correcto:

```
var nodo = document.getElementById("id_elem");
nodo.value = 5;
nodo.text = "5";
```

- **Crear los Nodos, Procesarlos y Luego Agregarlos al DOM:** Por ejemplo en caso de crear un lista, lo que se debe hacer es crear el elemento "ul", asignarle los valores correspondientes a sus atributos. Luego agregarle los hijos "li" también con sus valores correspondientes. Por ultimo, cuando se terminó de crear la lista se agrega al DOM. Esto evita que el DOM tenga que reestructurarse constantemente.

  Incorrecto (Agrega estructuras al DOM en varios pasos):

```
var lista = document.createElement("ul");
document.body.appendChild(lista);
for(var i = 0;i<10;i++)
{
 var elem = document.createElement("li");
document.body.appendChild(elem);
}
```

  Correcto:

```
var lista = document.createElement("ul");
for(var i = 0;i<10;i++)
{
var elem = document.createElement("li");
document.body.appendChild(elem);
}
document.body.appendChild(lista);
```

- **Utilizar Variables Locales en Lugar de Pasadas por Parámetro o Propiedades de Objetos:** JavaScript encuentra más efectivamente las variables que son creadas de forma local al contexto actual, por lo que lo más recomendable es que si un valor va a ser usado multiples veces, se defina una variable local que lo contenga.