**E) Aspect fonctionnel du client Web**

Le client web est composé d’une page HTML, d’un fichier de style, d’une applet Java gérant les sockets et de divers scripts, écrits en JavaScript, gérant la communication avec le serveur et l’affichage.

1. Aspect fonctionnel du client Web
2. HTML et CSS

Tous les éléments de l’interface tels que les formulaires, tableaux, menus ou le chat sont définis à la suite dans le fichier « index.html ». Leur position, leur taille, leur mise en forme et leur style sont défini quant à eux dans le fichier « style.css ».

Sans les scripts, tous ces éléments seraient affichés en même temps de manière anarchique. Ce sont eux qui gèrent l’affichage dynamique des éléments en fonctions des actions de l’utilisateur.

1. L’applet Java

Lors de mes recherches pour établir une communication via socket avec le serveur en JavaScript, je n’ai pas trouvé de solution satisfaisante ou du moins je n’ai pas réussi à les mettre en œuvre. C’est pourquoi je me suis tourné vers une applet Java.

Son principe est simple, la gestion des sockets à proprement parler est faite en Java et la communication entre les deux langages est faite via la page HTML. En Java, on accède à la page grâce à l’instruction « browser=JSObject.getWindow(this); ». Pour JavaScript, l’accès à l’applet se fait grâce à « document.getElementById('JavaSocketBridge'); »

Par exemple lors de la réception d’un message par l’applet, l’instruction suivante est exécutée : « browser .call("on\_socket\_get", arguments); » ce qui a pour effet d’appeler la méthode « on\_socket\_get(message) » du script.

Autre exemple, lors de l’envoi d’un message, l’instruction est exécutée: «document.getElementById('JavaSocketBridge').send(message); ». Ainsi la méthode « send() » de l’applet est appelée.

Toute la documentation de ce pont entre Java et JavaScript se trouve sur cette page : <http://stephengware.com/projects/javasocketbridge/>

1. La communication avec le serveur

L’ensemble des fonctions de communications sont séparées en trois fichiers « emission\_compte.js », « emission\_partie.js » et « reception.js ». Celles-ci pourraient se trouver dans un seul et même fichier mais je les ai séparées dans un souci de clarté.

Dans « emission\_compte », est gérée la vérification des champs remplis par l’utilisateur, et, dans le cas où ils sont validés, l’envoi des messages correspondants au serveur. Ces messages concernent uniquement la gestion du compte (connexion, création, changement de pseudo et de mot de passe) autrement dit, le mode non connecté.

Le script « emission\_partie » a le même rôle, mise à part le fait qu’il envoi des messages relatifs au partie (création de partie, demande de liste de partie, chat, demande d’informations sur un joueur…).

La gestion des messages reçus depuis le serveur se fait dans « reception ». Ceux-ci sont triés en fonction des messages précédemment envoyés et les fonctions correspondantes sont appelées.

1. Gestion de l’affichage

Toutes les fonctions de modification de l’affichage sont contenues dans le fichier « gestion.js ». C’est-à-dire qu’en fonction des messages reçus depuis le serveur on affiche les éléments HTML, tels que les formulaires, la liste des parties ou les menus. Par exemple la réception du message « CONNECTOK » entraine l’effacement du formulaire de connexion, et le remplacement du menu hors connexion par le menu en mode connecté.

1. Affichage d’une partie

Celui-ci est géré par le script « affichage\_partie.js » où tous les éléments HTML ainsi que leur style sont créés. La programmation de ce script est orienté objet.

Lors de la création d’une partie, un objet *Partie* est instanciéet lui-même va contenir des objets *Joueur, Cartes* ou encore *Mise*.

1. Programmation de l’application

La programmation de tout le client sauf la partie de l’affichage d’une partie de poker a été simplifiée par l’utilisation de JQuery. jQuery est un Framework JavaScript libre qui porte sur l'interaction entre JavaScript (comprenant l'AJAX) et HTML, et a pour but de simplifier des commandes communes de JavaScript. Ainsi, pour donner un exemple, une instruction du type « document.getElementById(‘element’) » est remplacée par « $(‘#element’ ) ».

1. Initialisation

A l’ouverture de la page la fonction suivante est appelée :

$(function(){

tout\_cacher();

$(".connecte").hide(); //cacher tous les elements de la classe connecte

$("#gestionjeu").hide();

$("#acceuil").hide(); //cacher l’element dont l’id est acceuil

$("#boutons").hide();

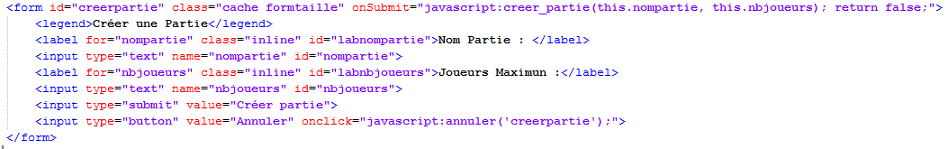
$("#commencer").hide();

});

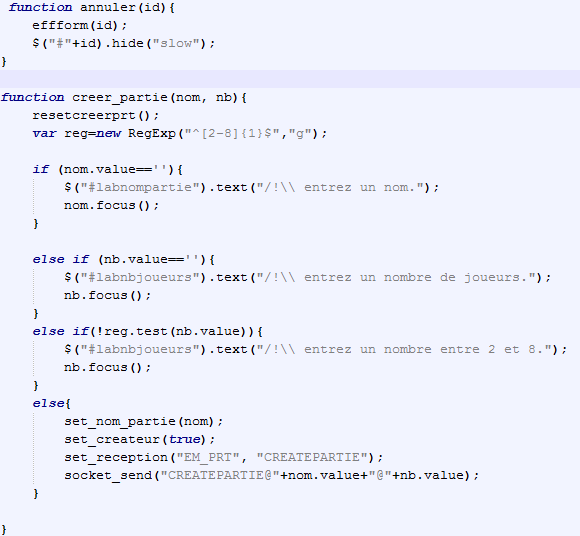
Avec JQuery, le dollar placé en-dehors d’une fonction reviens revient à utiliser l’évènement « onDocumentReady ». Ainsi, dès que la page est chargée, les instructions ci-dessus sont exécutées. Elles permettent de ne pas afficher les éléments HTML qui ne corresponde pas au mode non connecté. Vous pouvez remarquer la puissance de JQuery, qui avec une petite instruction, cacher un élément.

1. Détection et gestion d’un évènement

La plus grande partie de ces évènements est déclenchée par des éléments HTML. Ils sont programmés lors de la création d’une balise HTML, comme dans l’exemple qui suit.



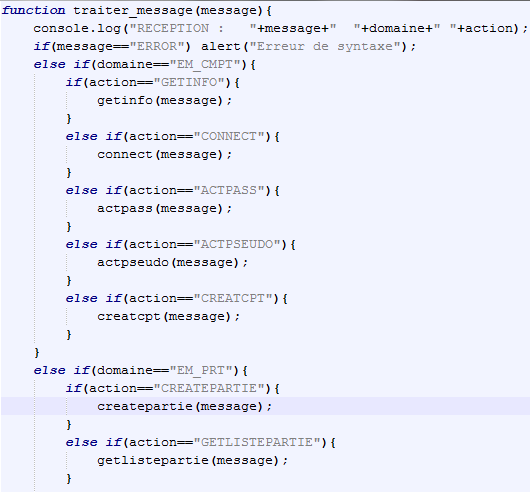
On observe ici la création d’un formulaire de création de partie avec un bouton de validation et d’annulation. Lorsque le bouton dont le type est « submit » est actionné, la méthode *creer\_partie* est appelée. La méthode *annuler* est appelée lors d’un clic sur le bouton correspondant.



La méthode *annuler* consiste à simplement à réinitialiser et cacher l’élément passé en paramètre.

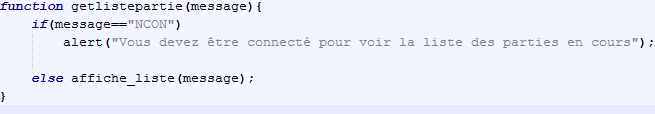
La méthode *créer\_partie*  contrôle la validité des champs remplis par l’utilisateur et notifie celui-ci en cas d’erreur. Si les champs sont valides, il règle le domaine de réception pour la redirection des messages reçus et envoi le message correspondant à l’action effectuée.

Un autre type d’évènements sont ceux déclenchés par la réception d’un message via le socket. Ce message est analysé et la fonction correspondante est lancée. Voici un morceau la fonction qui trie les messages :

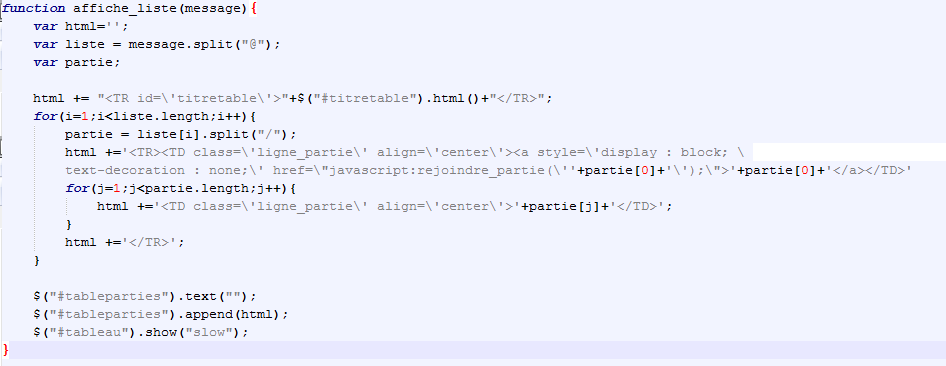


Celle-ci appelle une méthode qui traitera le message en fonction de l’action effectuée précédemment. On tri d’abord en fonction du domaine (EM\_CMPT, EM\_PRT, PARTIE) puis en fonction de l’action effectuée dans ce domaine.

Par exemple lors de la demande de la liste des parties, le domaine est réglé sur « EM\_PRT » et l’action sur « GETLISTEPARTIE », ainsi le message sera traité par la méthode *getlistepartie* qui se présente comme suit :



Le message est alors comparé aux messages possibles listés dans le protocole. Si le message correspond à la validation de l’action par le serveur, le message est alors traité à proprement parlé. Prenons *affiche\_liste* comme exemple :



Cette fonction traite un message de la forme :

SETLISTEPARTIE@partie/nb joueurs/joueurs max@partie/nb joueurs/joueurs max

Le message est découpé selon les « @ » dans un premier tableau. On boucle sur le tableau. A chaque tour, on découpe la chaine contenu dans le tableau selon les « / » que l’on place dans un nouveau tableau. Puis on ajoute du code HTML, complété avec les informations contenues dans notre tableau, dans une chaine de caractère.

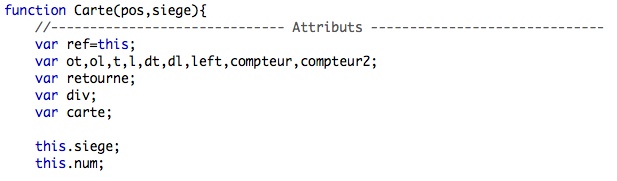
Enfin, ce code est inséré dans notre page grâce à la simple instruction *$("#tableparties").append(html);* générant ainsi un affichage dynamique !

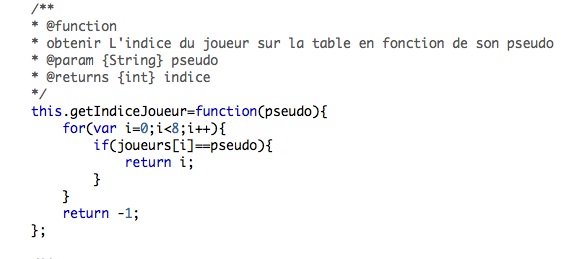
1. Gestion d’une partie

Pour la programmation de l'affichage d’une partie nous avons essayé de nous rapprocher au maximum d'une programmation objet de classes notamment en rendant les attributs inaccessibles hors de l'objet  et en mettant en place des méthodes pour manipuler les objets :

Voici notamment comment déclarer un constructeur d'objet, ici l'objet Div :



Ici les variables précédés de « var » sont inaccessibles hors de l'objet. Seul des getters et setters permettraient d'y accéder. Les variables *this.siege* et *this.num* sont public donc directement accessibles et modifiables hors de l'objet. Par exemple, si « *C* » était une instance de new Carte() on pourrait faire *C.siege=10* mais pas *C.compteur=10*.

Voici un exemple de méthode :

s



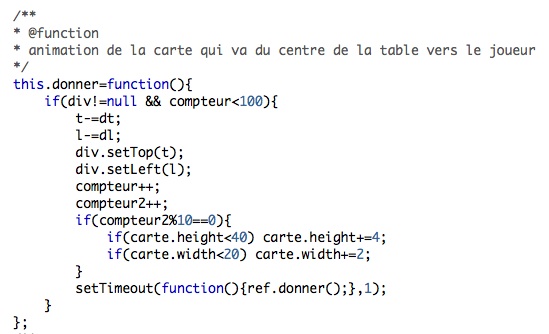
Voici comment appeler une méthode :

Pour simplifier la manipulation de la page html nous avons créé un objet Div ayant en attribut une variable stockant un élément <div> que l'on veut ajouter à la page html. Ainsi il suffit d'instancier un objet Div et d'appeler les différentes méthodes « créer » pour interagir avec le contenu ou l'affichage de la balise. Il ne restera plus qu'a l'ajouter à la page en ajoutant en tant que fils d'une Div déja existante ou en utilisant la fonction « document.appendChild(div) ».

Ci-dessous une méthode de l'objet Div permettant de modifier son arrière-plan. Elle éclairci le code, permet de changer facilement et rapidement les caractéristiques de l'arrière-plan.

Un des aspects pratiques de JavaScript est la facilité à mettre en des animations peu complexes. Il suffit de mettre un place deux variables ayant comme valeur de départ la position horizontal et verticale de l'objet, puis d'incrémenter/décrémenter ces variables selon un certain intervalle de temps et de donner à l'objet ses nouvelles positions.

Voici un exemple de méthode créant une animation :



Ici la méthode « donner() » influe sur la position de la variable *div*, qui est une instance du prototype Div. Les variable *t* et *l* représentent la position de *div.* On les décrémentent puis on affecte a *div* sa nouvelle position. Puis on rappelle dans un intervalle de temps établit à 1 milliseconde la méthode « donner() » sur l'objet. En plus de la position ici la méthode agit aussi sur la taille d'une image (effet d'agrandissement).