

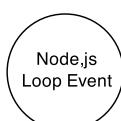
### Programmation asynchrone

#### ■ ■ ■ Exécution asynchrone et boucle d'événement

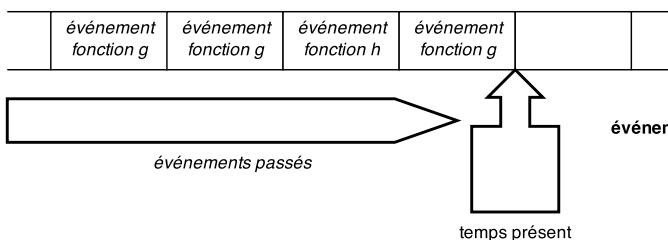
1 –

#### Source

```
function ma_fonction()
{
    ...
    setTimeout(fonction f,0)
    ...
}
```



- a. À quoi correspond `setTimeout (fonction, 0)` ?
- b. Montrer les transformations de la liste d'événements ;
- c. Qu'est-ce que la « boucle d'événements », « *loop event* » ?  
Comment utilise-t-elle la liste d'événements ?
- d. Quand un événement est-il pris en compte ?



#### ■ ■ ■ Fonction à conservation d'état

2 – Soit le code du « producteur/consommateur » :

```
1 var produit = 0;
2
3 function appel_producteur() {
4     producteur.next();
5 }
6
7 function* p() {
8     var compteur = 0;
9     while(true)
10    {
11        compteur++;
12        produit = compteur;
13        console.log("Production " + pro
14        duit);
15        setTimeout(appel_consommateur,0);
16        yield null;
17    }
18 }
```

```
1 function appel_consommateur() {
2     consommateur.next();
3 }
4
5 function* c() {
6     while(true)
7     {
8         console.log("Consommation "+pro
9         duit);
10        setTimeout(appel_producteur,
11        0);
12        yield null;
13    }
14 var producteur = p();
15 var consommateur = c();
16
17 producteur.next();
```

- a. À quoi sert le caractère « \* » dans la déclaration `function *p()` ?
- b. À quoi sert l'instruction « `yield` » ?
- c. Pourquoi déclare-t-on une variable en lui affectant l'exécution d'une fonction :  
`var producteur = p();`
- d. Qu'est-ce que fait la méthode « `next()` » ?
- e. Pourquoi a-t-on besoin de la fonction « `appel_producteur()` » ?
- f. Étudier l'exécution du « producteur/consommateur » où vous détaillerez l'évolution de la liste d'événement en fonction de l'exécution du producteur et du consommateur.

## ■ ■ ■ ■ Équité et « Scheduling »

3 – Soit le code des philosophes mangeur de ramen :

```
1 var liste_philosophes = [];
2 var liste_philosophes_a_traiter = null;
3 var philosophe_a_reveiller = null;
4
5 var compteur_chaises = 4;
6
7 function reveil_philosophe()
8 {
9     philosophe_a_reveiller.next();
10}
11
12 function* philosophe(n)
13 {
14     while(true)
15     {
16         if (compteur_chaises > 0)
17         {
18             compteur_chaises--;
19             console.log("Philosophe : "+n+" prend une chaise");
20             yield null;
21             console.log("Philosophe : "+n+" mange");
22             yield null;
23             console.log("Philosophe : "+n+" se leve");
24             compteur_chaises++;
25         }
26         else
27         {
28             console.log("Philosophe : "+n+" reflechit");
29         }
30         yield null;
31     }
32 }
33
34 for(i=0; i<5; i++)
35     liste_philosophes.push(philosophe(i));
36
37 liste_philosophes_a_traiter = liste_philosophes.slice(0);
38
39 function scheduler()
40 {
41     var numero_philosophe = Math.floor(Math.random() * liste_philosophes_a_traiter.length);
42     philosophe_a_reveiller = liste_philosophes_a_traiter[numero_philosophe];
43     liste_philosophes_a_traiter.splice(numero_philosophe, 1);
44     if (liste_philosophes_a_traiter.length == 0)
45     {
46         liste_philosophes_a_traiter = liste_philosophes.slice(0);
47     }
48     setTimeout(reveil_philosophe, 0);
49     setTimeout(scheduler, 0);
50 }
51
52 scheduler();
```

Étudier l'exécution du programme résolvant le problème des philosophes :

- Quelle activité est réalisée par chaque philosophe ?
- À quoi servent les différents « `yield` » ?
- Comment est utilisée la liste d'événements et la boucle d'événement ?
- Pourquoi parle-t-on de « *scheduling* », « *ordonnancement* » ?
- Est-il équitable ? Est-ce qu'il y a des risques de famine ? Pourquoi ?

4 – Proposez une solution au problème du producteur/consommateur :

- équitable ;
- utilisant un buffer de 5 cases.