# Générateur de fiction v.01 – manuel

[Générateur de fiction v.01 – manuel 1](#_Toc60605502)

[1 Généralités sur la bibliothèque genF 2](#_Toc60605503)

[a) La bibliothèque 2](#_Toc60605504)

[b) Le programme utilisateur 2](#_Toc60605505)

[c) Informations dans la console 2](#_Toc60605506)

[d) Les tables du monde 3](#_Toc60605507)

[e) La table des évènements 4](#_Toc60605508)

[f) La variable messages 4](#_Toc60605509)

[2 Accès aux items du monde 5](#_Toc60605510)

[2. 1 Accès aux tables du monde 5](#_Toc60605511)

[a) La variable **Monde** 5](#_Toc60605512)

[b) La fonction **copieTable**(table) 5](#_Toc60605513)

[2. 2 Récupération des items du monde 5](#_Toc60605514)

[a) La fonction **getEntity** (nom) 5](#_Toc60605515)

[b) La fonction **getItem** (table, prop, val) 6](#_Toc60605516)

[3 Gestion des évènements 7](#_Toc60605517)

[3. 1 Variable Event 7](#_Toc60605518)

[3. 2 Manipulation des évènements 7](#_Toc60605519)

[a) La fonction **getItem** ("Event", "function", val) 7](#_Toc60605520)

[b) La fonction **faire** (args) 7](#_Toc60605521)

[c) La fonction **faireAvecProba** (args) 9](#_Toc60605522)

[d) Modifications pérenne des probabilités : modifProba et modifProbaBranche 10](#_Toc60605523)

[e) La fonction **verifie** (evenement) 12](#_Toc60605524)

[f) la fonction **affecteValeur** (tableMonde, propID, valID, prop, val) 13](#_Toc60605525)

[4 Gestion de l’histoire 13](#_Toc60605526)

[4. 1 La variable **story** 13](#_Toc60605527)

[4. 2 La fonction **finie**() 15](#_Toc60605528)

[5 Affichage dans le navigateur 15](#_Toc60605529)

[5. 1 Affichage du vocabulaire omeka avec **showData** 15](#_Toc60605530)

[5. 2 affichage des SVG 15](#_Toc60605531)

[a) la fonction **getAffichage**() 15](#_Toc60605532)

[b) la fonction **affiche** (table, prop, nom, index) 16](#_Toc60605533)

[c) la fonction **efface**(t) 16](#_Toc60605534)

[d) la fonction **affichagePeriodique** 16](#_Toc60605535)

## Généralités sur la bibliothèque genF

##### La bibliothèque

La bibliothèque fictionGene-vx.js (x étant le numéro de version) contient les fonctions permettant de générer une histoire à partir des données Omeka. Pour l’utiliser dans votre programme, il faut en stocker une instance dans une variable globale en l’invoquant avec la fonction genF(id) où id est le numéro de votre vocabulaire Omeka.

Tous les exemples présentés sont relatifs au projet Dessetrousse

##### Le programme utilisateur

Si vous utilisez le fichier html fourni, en le renommant en fonction de votre projet, vous écrirez votre code dans le fichier main.js. La première ligne de code sera alors :

let gen = genF(x) ; où x est le numéro de votre vocabulaire (21 pour le projet Dessetrousse). La bibliothèque genF est alors stockée dans la variable gen

exemple :

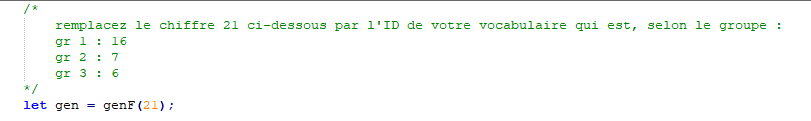


Figure  : début du fichier main.js

Les lignes en vert ne sont que du commentaire, vous pouvez les supprimer.

Je supposerai par la suite que la bibliothèque est stockée dans la variable globale gen

##### Informations dans la console

L’appel à la fonction genF initialise la bibliothèque qui récupère alors les données de Omeka et les met en forme de façon à séparer dans une liste l’ensemble des tables du monde et dans une autre l’ensemble des évènements.

Durant cette initialisation, le programme affiche dans la console la liste des données brutes issues de Omeka, puis la liste de la table du monde mise en forme puis la liste des évènements. La bibliothèque genF travaille avec ces 2 dernières listes et non avec la liste des données brutes



Figure  : affichage initial dans la console – projet Dessetrousse

Dans l’exemple Dessetrousse, la base de données contient 5 tables : les 4 premières sont les tables du monde (4 tables), la dernière est la liste des évènements (11 évènements).

Cliquer sur le petit triangle en début de chaque ligne permet de développer les listes.

Attention, la numérotation des positions dans une liste débute à 0 et non à 1.

##### Les tables du monde

La liste des tables du monde est composée d’objets, Chaque table du monde étant un objet JavaScript noté {…} dans les listes [] compactées de la . Chaque table correspond à une Entité du schéma de bases de données.

Une table du monde possède les 2 propriétés : *key* et *values*. La propriété *key* a pour valeur le nom de l’Entité (nom de la table). Cette valeur correspond à la classe du vocabulaire choisie pour définir les items dans omeka. La propriété *values* est la liste (Array) de tous les items de cette entité. Chaque item est lui-même un objet JavaScript dont les propriétés sont celles créées pour l’item dans omeka, y compris les url des media associés.

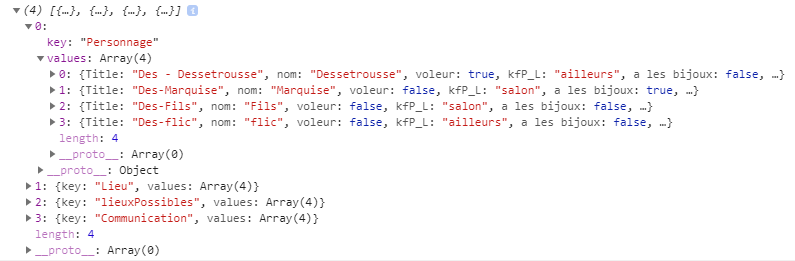


Figure  : tables du monde du projet Dessetrousse.



Figure  : détail de l’item « Dessetrousse » de la table « Personnage » du projet Dessetrousse

On retrouve bien dans cet item toutes les propriétés de l’item définies dans omeka

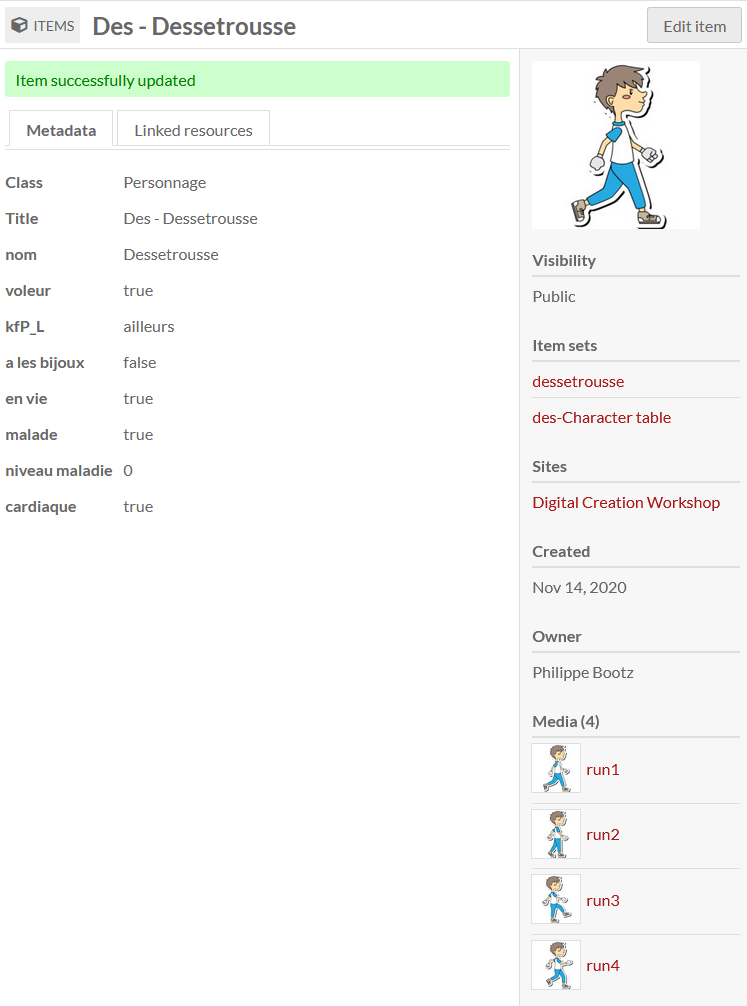


Figure  : propriétés de l’item « Dessetrousse » dans omeka

##### La table des évènements

Chaque évènement est un objet dans la table des évènements. Il n’y a pas de propriété *key* ni *values*, chaque objet de la table renferme les propriétés qui le décrivent dans omeka

##### La variable messages

lorsque la variable interne messages vaut true, la bibliothèque écrit dans la console des messages d’erreur qui peuvent aider au débogage. Par défaut elle est initialisée à false de façon à ne pas surcharger la console. On accède à cette variable par **gen.messages**

## Accès aux items du monde

### Accès aux tables du monde

##### La variable Monde

La variable Monde renferme les tables du monde. On accède à cette variable par **gen.Monde**

Type : Array d’Objects

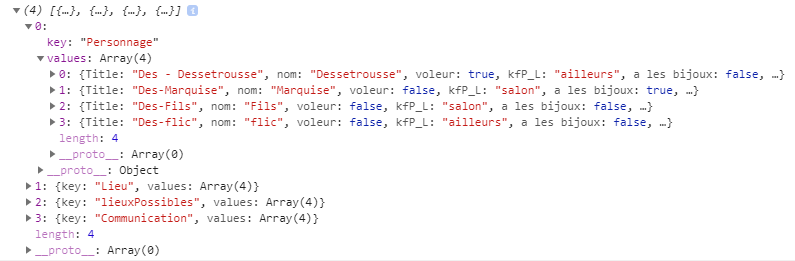


Figure  : valeur initiale de la variable Monde du projet Dessetrousse

Attention : si vous voulez afficher la table du monde dans la console par l’instruction console.log (gen.Monde), la table que vous verrez en réalité est toujours celle en cours à la fin du programme, même si l’instruction console.log se situe dans le corps du programme. Cela est dû au fait que cette table est composée d’objets et que les objets sont passés par référence et non par valeur. Lorsque vous regardez la sortie console en fin d’exécution, vous voyez donc la valeur de la table au moment où vous regardez, et non au moment où elle a été écrite dans la console. Pour voir l’état réel de la table du Monde dans la console il faut mettre un point d’arrêt dans le débogueur à l’endroit du programme où vous voulez en regarder les valeurs ou utiliser la fonction copieTable.

##### La fonction copieTable(table)

La fonction copieTable permet de capturer une copie instantanée d’une table d’objets de façon à en figer les valeurs. On l’utilise pour vérifier des valeurs en cours de programme, par exemple dans un console.log.

Syntaxe : gen.copieTable(table) ;

Paramètre d’entrée :

* table (Array d’Objets) : table dont on veut obtenir une copie. Ce peut être gen.Monde, gen.Event ou une table du monde obtenue par les fonctions getEntity et getItem

Valeur retournée : la copie de la table (Array d’Objects)

Exemple : voir l’exemple donné pour la modification pérenne d’une propriété en . En , dans l’étape « état plus probable », la propriété proba aurait eu la valeur 1 et non 2 au moment où on prenait la capture d’écran si on avait utilisé console.log(gen.Event) et non console.log (copieTable (gen.Event) car, dans l’état final du programme, proba a de nouveau la valeur 1

### Récupération des items du monde

##### La fonction getEntity (nom)

renvoie la liste des items d’une table du monde de la table du monde

Syntaxe : gen.getEntity (nom)

Paramètre d’entrée :

* nom (string) : valeur de la propriété *key* définissant la table Entité retournée

Valeur retournée : la table du monde (Array d’objets) ou une liste vide si le paramètre d’entrée est incorrect

Attention : Les items sont passés par référence, ce qui implique que toute modification d’une propriété d’un des items renvoyés correspond à une modification de la table du monde stockée dans la variable Monde et utilisée par genF. Utilisez la fonction copieTable pour récupérer une copie instantanée de la table.

Exemple : affichage dans la console de la table des personnages



Affichage console :

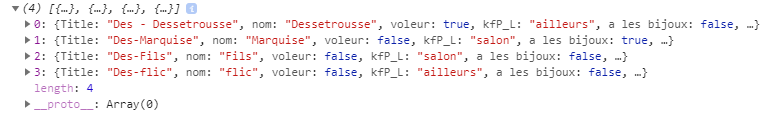


Figure  : valeur initiale de la table Personnage

##### La fonction getItem (table, prop, val)

Retourne un item particulier d’une table du monde par la connaissance de la valeur de son identifiant

Syntaxe : gen.getItem (table, prop, val)

Paramètres d’entrée :

* table (string) : valeur de la propriété *key* définissant l’Entité.
* prop (string) : nom de la propriété identifiant l’item cherché
* val : valeur de la propriété prop de l’item cherché

valeur retournée : l’objet item (Object) ou null en cas d’erreur

Exemple : affichage dans la console de l’item Dessetrousse



Affichage console :



Figure  : item Dessetrousse initial

Attention : L’items est retourné par référence, ce qui implique que toute modification d’une propriété de cet item correspond à une modification de la table du monde stockée dans Monde et utilisée par genF.

## Gestion des évènements

### Variable Event

La variable Event est la liste des évènements du projet. On y accède par **gen.Event**

Type : Array d’Objects



Figure  : valeur initiale de la variable Event du projet Dessetrousse (l’évènement "deplacer" est détaillé)

### Manipulation des évènements

##### La fonction getItem ("Event", "function", val)

La fonction getItem permet également de récupérer l’identifiant d’un évènement.

Syntaxe : gen.getItem ("Event", "function", val)

Paramètres d’entrée :

* le string "Event" : indique qu’on cherche un item de la table Event
* le string "function" : indique qu’on regarde la propriété *function* de l’évènement
* val (string) : nom de la fonction de l’évènement valeur de sa propriété *function*

valeur retournée : l’objet évènement (Object) ou null en cas d’erreur

##### La fonction faire (args)

C’est la fonction prinicipale de la bibliothèque. Elle exécute un évènement, quelque soit le type de l’évènement. La fonction commence par trouver l’Item évènement dans la table Event, vérifie ensuite que les conditions initiales requises (requirements) sont remplies, puis, si ces conditions sont remplies, appelle le programme à exécuter pour les évènements de type « pg », choisit aléatoirement une branche en tenant compte des pondérations des branches pour les évènements de type « choix » et traite toute la séquence des évènements de type « séquence ». Enfin, la fonction applique le final state aux tables du monde et met l’histoire à jour.

Syntaxe : gen.faire ([nom,para1,para2…])

Paramètres d’entrée : ils sont donnés sous la forme d’un Array []

* nom (string) : le nom de l’évènement, c’est-à-dire la valeur de la propriété *function* de l’évènement
* paramètres suivants : les paramètres à passer à l’évènement. Si l’évènement est de type « pg », para1 doit être la fonction à exécuter, les paramètres suivants correspondant aux paramètres à passer à cette fonction. Si l’évènement est de type « élémentaire », « choix » ou « séquence », les paramètres à passer sont des valeurs particulières des propriétés indiquées dans les finalStates, dans l’ordre des Entités indiquées dans la propriété t*ablesParameters* de l’évènement

Valeur retournée : un booléen indiquant si l’évènement s’est bien déroulé

Remarques :

* Aucun évènement n’est plus exécuté par faire après l’évènement de fin « \* » et la fonction renvoie alors systématiquement false.
* Pour les évènements de type « choix », si l’évènement sélectionné aléatoirement par la fonction ne peut pas être exécuté, aucun autre évènement n’est choisi et la fonction renvoie false

Exemple 1 : exécution d’un évènement non programmé



Les paramètres "deplacer","Dessetrousse","rue" correspondent à des valeurs possibles des propriétés function et tablesParamaters (cf )

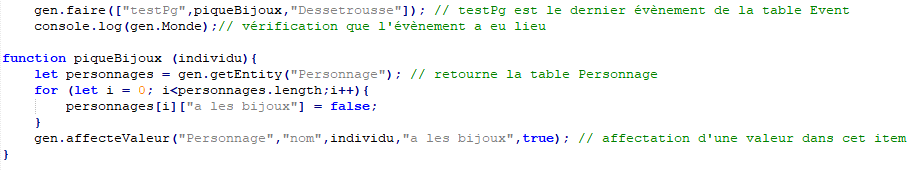
Affichage console :



Figure  : exécution d’un évènement « déplacement ».

On constate sur la sortie console suivante que la valeur de kfP\_L a bien changé par rapport à sa valeur initiale donnée en .

Exemple 2 : exécution d’un évènement de type « pg »



Affichage console :

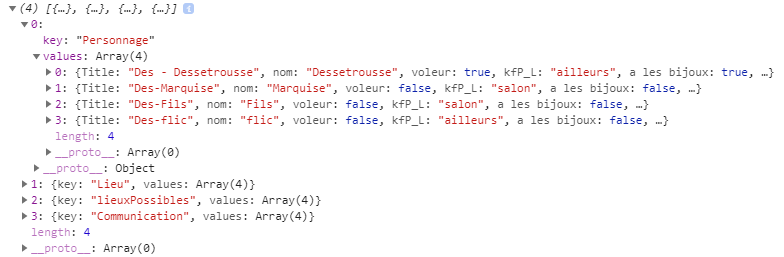


Figure  : exécution d’un évènement programmé

##### La fonction faireAvecProba (args)

Cette fonction permet de construire à la volée et exécuter un évènement temporaire en pondérant des évènements existants. les divers évènements du scénario. L’évènement créé est de type « choix ». Les seules informations à entrer en paramètres sont le détail des branches.

Syntaxe : gen. faireAvecProba (args)

Paramètre d’entrée :

* Args: liste (Array) composée des listes définissant les branches. Chaque liste de branche est elle-même une liste dont le premier élément est la liste des arguments de l’évènement considéré à passer à la fonction faire qui exécutera cet évènement et le second le poids de cet évènement dans la branche.

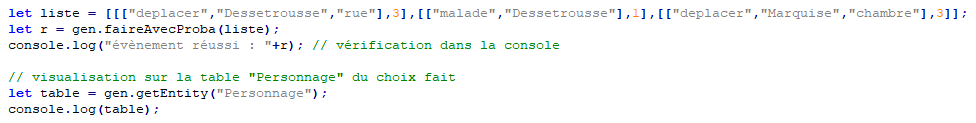
Valeur retournée : un booléen indiquant si l’évènement s’est bien déroulé

Remarques :

* Aucun évènement n’est exécuté par cette fonction après l’évènement de fin « \* » et la fonction renvoie alors systématiquement false.
* Les branches peuvent contenir tout type d’évènement, y compris des évènements programmés et des évènements de type « choix ».
* L’évènement créé par faireAvecProba est transitoire. Il n’est pas ajouté à la liste Event et ne peut donc être combiné avec d’autres évènements.

Attention : Pour modifier de façon permanente le poids d’un évènement, il ne faut pas utiliser la fonction faireAvecProba mais modifier la propriété *proba* d’un évènement élémentaire. Pour modifier de façon permanente le poids d’une branche d’un évènement de type « choix », il faut modifier la propriété *proba* de cette branche

Exemple d’utilisation de faireAvecProba : choix entre le déplacement de Dessetrousse, de la marquise ou l’aggravation de la maladie de Dessetrousse



Affichage console :



Figure  : exécution d’un évènement à la volée de type « choix »

On constate en comparant la à et que l’évènement choisi a été la dernière branche : le déplacement de la marquise, les autres propriétés des items de la table Personnage n’ayant pas varié

##### Modifications pérenne des probabilités : modifProba et modifProbaBranche

Les 2 fonctions suivantes, à programmer par vous-mêmes dans main.js si vous les utilisez, vous permettent de modifier de façon pérenne la probabilité d’un évènement de tout type ou d’une branche d’un évènement de type « choix ».

Exemple 1 : programmation d’une modification pérenne du poids d’un évènement :

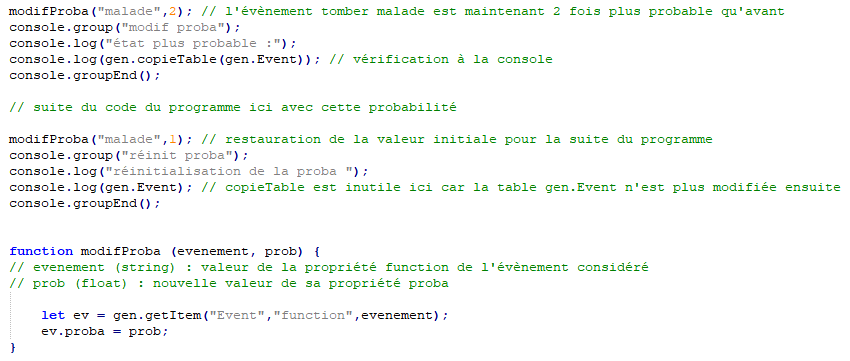
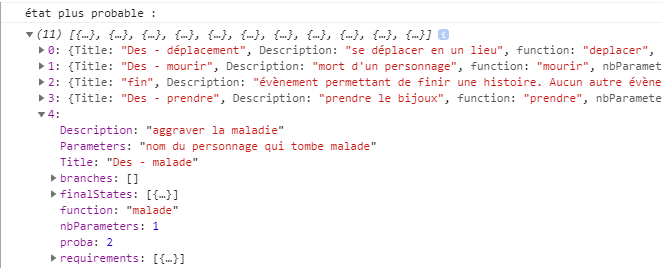


Figure  : fonction de modification pérenne d’une propriété d’un évènement

Affichage console (extraits)



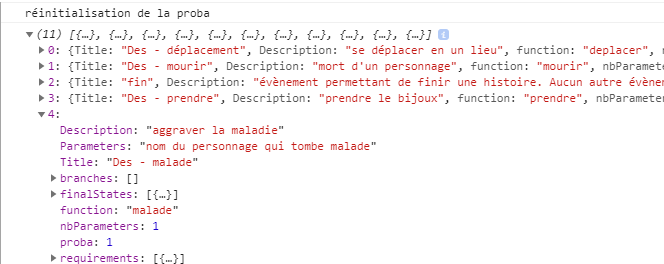


Figure  : modification directe du poids d’un évènement

Exemple 2 : programmation pérenne du poids d’une branche dans un évènement de type « choix »

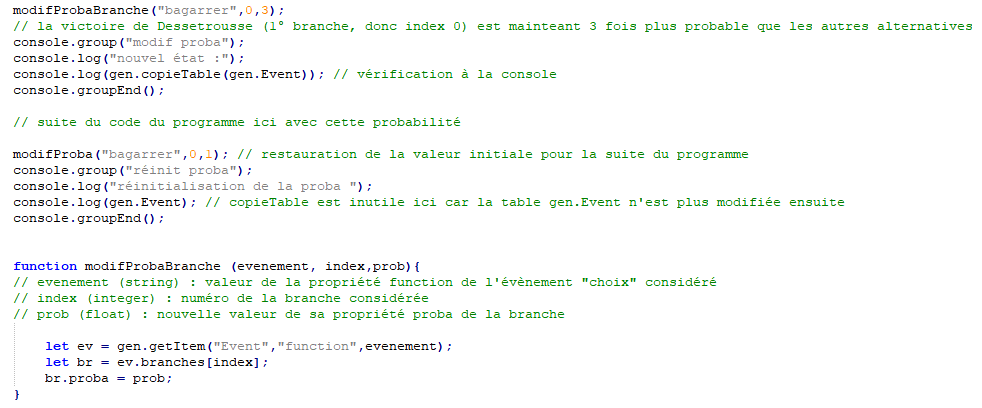
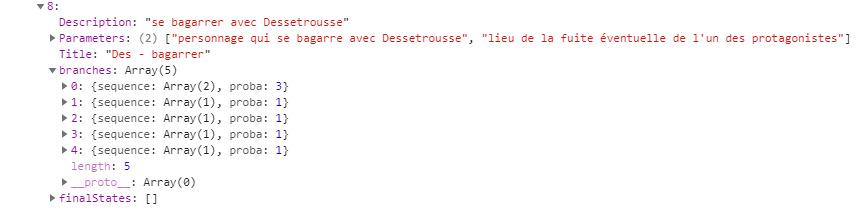


Figure  : fonction de modification pérenne du poids d’une branche

Affichage console :







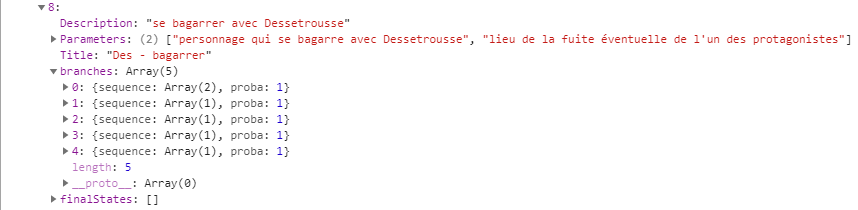


Figure  : modification directye du poids d’une branche

##### La fonction verifie (evenement)

Cette fonction verifie si les requirements d’un évènement sont vérifiés ou non. Cela permet de savoir si l’évènement est utilisable ou non à cet endroit du programme.

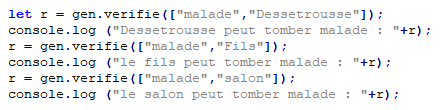
Syntaxe : gen.verifie (args)

Paramètres d’entrée : ils sont donnés sous la forme d’un Array []

* nom (string) : le nom de l’évènement, c’est-à-dire la valeur de la propriété *function* de l’évènement
* paramètres suivants : les paramètres à passer à l’évènement. Si l’évènement est de type « pg », para1 doit être la fonction à exécuter, les paramètres suivants correspondant aux paramètres à passer à cette fonction. Si l’évènement est de type « élémentaire », « choix » ou « séquence », les paramètres à passer sont des valeurs particulières des propriétés indiquées dans les finalStates, dans l’ordre des Entités indiquées dans la propriété t*ablesParameters* de l’évènement

Valeur retournée : un booléen indiquant si l’évènement est possible ou non

Exemple :



Affichage console :

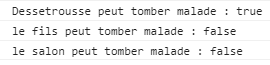


Figure  : vérification de requirements

Remarque : les paramètres d’entrée sont les mêmes que pour la fonction faire car la fonction verifie utilise en fait la fonction faire

##### la fonction affecteValeur (tableMonde, propID, valID, prop, val)

Cette fonction affecte de façon pérenne une valeur à une propriété d’un item d’une table du monde. Elle est plus simple à utiliser que l’affectation directe pour le cas où l’un des paramètres d’entrée est une variable.

Syntaxe : gen.affecteValeur (tableMonde, propID, valID, prop, val)

Paramètres d’entrée :

* table du monde (string) : la valeur de la propriété *key* de la table du monde impactée
* propID (string) : nom de la propriété qui sert d’ID pour la table, c’est-à-dire qui permet de repérer l’item de la table de façon non univoque, chaque item ayant une valeur différente pour cette propriété
* valID (string) : la valeur de propID pour l’item impacté
* prop (string) : le nom de la propriété à modifier
* val : la nouvelle valeur de prop à affecter à l’item

Valeur retournée : un booléen indiquant si l’affectation a eu lieu ou pas

Exemple : cette fonction est utilisée dans l’exemple 2 de la fonction faire ()

## Gestion de l’histoire

### La variable story

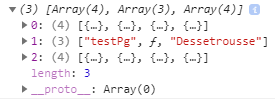
La variable story stocke la succession des états (sous forme de copies) et évènements ; donc l’histoire. Elle permet de gérer l’énoncé lorsqu’on utilise des scénarios et storyboard pour écrire l’histoire après sa génération. Elle n’est pas utilisée lorsque l’histoire est racontée en cours même de génération. Elle permet toutefois de détecter après coup quelle branche d’un évènement à choix s’est produite.

Syntaxe : gen.story

Attention : Lorsqu’on exécute les fonctions faire et faireAvecProba, cette variable est automatiquement mise à jour quel que soit le type d’évènement, mais pas bien sûr si vous modifiez les états par affecteValeur en dehors d’un évènement programmé. Il n’y a aucun problème si affecteValeur est utilisé dans un évènement programmé

Remarque : pour les évènements créés à la volée par faireAvecProba (dont la propriété *function* a toujours pour valeur "NE\*\*") et les évènements de type « séquence » ou « choix », l’information sur l’évènement suit l’état final du monde créé par l’évènement alors que pour chaque évènement élémentaire, l’état final est indiqué après l’information sur l’évènement. La variable story permet ainsi de détecter a posteriori si un état du monde est simplement dû à un évènement élémentaire ou s’il résulte d’un évènement de type plus complexe.

Exemple 1 : vérification que affecteValeur peut être exécuté dans un évènement programmé. La variable story de l’exemple 2 de la fonction faire () est :



Exemple 2 : avec une gestion automatique d’évènements de tous types

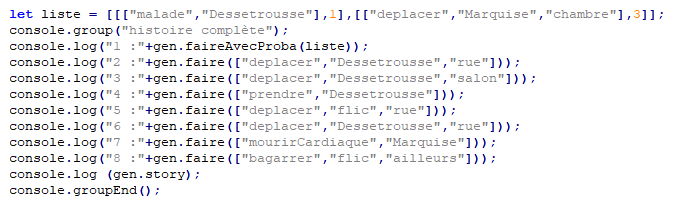


Figure  : visualisation d’une histoire dans la console

Affichage console :



Figure 19 : variable story créée lors de l’histoire. On y repère les types d’évènements (flèches rouges) et l’indication de la branche choisie pour les évènements de type « choix »

Exemple 3 : retirer l’évènement 3 (mis en commentaire dans le code) modifie profondément l’histoire. L’évènement 4 (le vol des bijoux) ne peut plus avoir lieu car Dessetrousse n’est alors pas dans le salon, lieu où sont situés les bijoux, mais dans la rue, l’évènement n’a pas lieu car Dessetrousse est déjà dans la rue et l’évènement 8 n’a pas lieu non plus car le policier ne se bagarre avec Dessetrousse que si ce dernier a volé les bijoux.

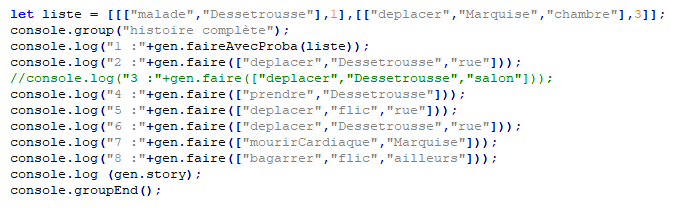


Figure  : légère modification du programme précédent

Affichage console :

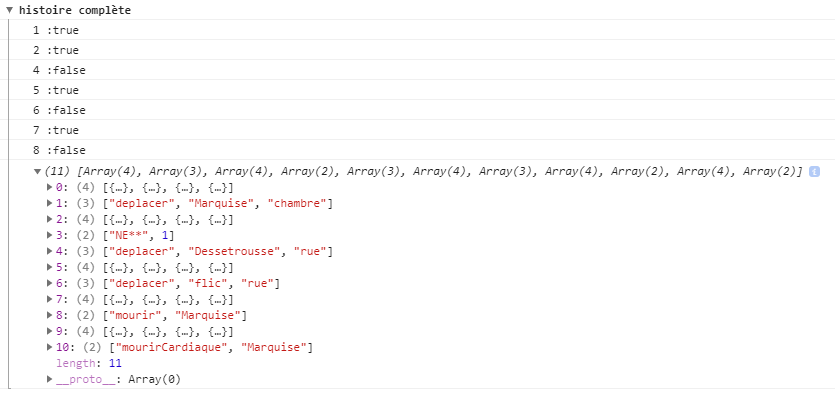


Figure  : variable story en cas d’évènements avortés

Remarque : Les évènements qui n’ont pas lieux ne sont pas stockés dans la variable story

### La fonction finie()

La fonction finie permet de repérer que l’évènement de fin « \* » a eu lieu et que l’histoire est finie puisque plus aucun évènement ne sera appliqué. Elle vaut false en cours de génération et true en fin de génération.

Syntaxe : gen.finie()

## Affichage dans le navigateur

### Affichage du vocabulaire omeka avec showData

La fonction showData (id) permet d’afficher dans le fichier HTML le vocabulaire omeka de votre projet

Syntaxe : gen.showData(id)

Paramètre d’entrée :

* id (integer) : ID du vocabulaire à afficher

Valeur retournée : aucune

### affichage des SVG

L’affichage des SVG est asynchrone : le programme charge systématiquement le SVG à afficher avant de l’afficher mais le programme se poursuit durant ce chargement.

##### la fonction getAffichage()

La fonction getAffichage permet de savoir si un SVG est actuellement affiché ou non

Syntaxe : gen.getAffichage()

Valeur de retour : un booléen indiquant si un SVG est actuellement affiché ou pas

##### la fonction affiche (table, prop, nom, index)

Cette fonction affiche un media SVG attribué dans omeka à un item ou un évènement

Syntaxe : gen.affiche (table, prop, nom, index)

Paramètres d’entrée :

* table (string) : la valeur de la propriété key pour un item d’une table du monde ou le terme "Event" pour un évènement
* prop (string) : la propriété ID de l’item du monde ou le terme "function" pour un évènement
* nom (string) : la valeur de la propriété ID de l’item ou le nom attribué à *function* pour un évènement.
* index (integer) : paramètre optionnel qui n’est utilisé que si plusieurs médias sont associés à l’item. index est alors le numéro du media à afficher dans la liste des médias de l’item. Cette liste étant un Array, le premier média se trouve à l’index 0. Si index est absent, c’est le média d’index 0 qui est affiché.

Valeur retournée : aucune

Exemple : gen.affiche("Personnage","nom","flic");

##### la fonction efface(t)

Cette fonction efface l’affichage en cours. Comme l’affichage est asynchrone, il peut arriver que l’ordre d’effacer soit donné avant l’affichage effectif qui ne serait alors pas effacer. C’est pourquoi la fonction teste la l’effacement à intervales réguliers jusqu’à ce que l’effacement ait réellement lieu. Pour éviter de tester indéfiniment, par exemple rien n’est affiché et qu’il n’y a rien à effacer, les tentatives d’effacement cessent au bout de quelques secondes.

Syntaxe : gen.efface(t)

Paramètre d’entrée :

* t (s) : paramètre optionnel. Il indique la durée en seconde pendant laquelle l’effacement est testé. Par défaut t = 2 s

Valeur de retour : aucune

Exemple : gen.efface();

##### la fonction affichagePeriodique

Cette fonction permet d’effectuer périodiquement une fonction de gestion de l’affichage. Elle contient en interne 2 fonctions qui gèrent le timer : run, qui le déclenche, et stop qui l’arrête.

Pour l’utiliser il faut la stocker dans une variable qui se comporte alors comme un objet puis exécuter run ou stop comme des méthodes de cet objet.

Les fonctions affichagePeriodique et stop ne nécessitent pas de paramètre d’entrée et ne renvoient aucune valeur.

La fonction run nécessite des paramètres en entrée mais ne renvoie aucune valeur

Paramètres d’entrée de la fonction run :

* fct : nom de la fonction à exécuter périodiquement (sans les parenthèses et sans guillemets !)
* cadence (float) : paramètre optionnel : cadence du timer en nombre d’images par seconde (fps). Par défaut la cadence est fixée à 25 fps. La cadence est limitée à 60 fps.
* Si la fonction fct nécessite des paramètres d’entrée, ceux-ci sont ajoutés après f qui est alors obligatoire

Exemple 1 : alternance de 2 SVG. La bascule s’arrète après l’affichage de 10 SVG (soit 5 alternances)

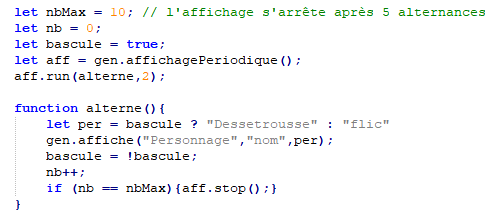


Figure  : affichage périodique

Exemple 2 : animation du tableau d’images associées à un item utilisées comme sprites. L’animation dure 4 s ; l’ordre d’arrêt étant donné 4000 ms après le démarrage de l’animation

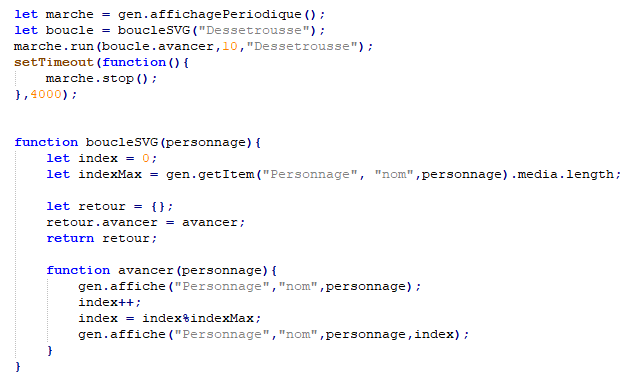


Figure  : affichage périodique avec introduction de paramètres