Objet JavaScript utilisant Modèle FACTORY PATTERN :

Il existe également d'autres façons de créer un objet à partir du modèle de création d'objet de javascript – Modèle Factory, Modèle Constructeur, Modèle Prototype ...)
Modèle Factory est un modèle orienté objet, les instances d'objet sont créées à l'aide de la fonction Factory.
Comme dans le modèle Constructeur, les objets sont créés avec l'opérateur new ; dans le modèle Factory l'opérateur utilisé est la fonction factory.
Le modèle Factory utilise une fonction pour masquer le processus de création d'objets spécifiques et renvoyer leur référence, il renvoie une nouvelle instance chaque fois que la fonction factory est appelée.

➔ Ou utiliser Factory PATTERN ?

Utilisable dans le cas ou le type d'objet est décidé au moment de l'exécution
Utilisable dans une application qui gère, maintient et modifie la collection d'objets différentes mais ayant des méthodes et des propriétés en commun.
Il peut agir sur une collection de documents avec combinaison de document XML, ou PDF.

➔ Les avantages FACTOTY PATTERN

Il offre la réutilisation et une fonctionnalité principale
Permet d'ajouter ou supprimer de nouvelles fonctions d'objet de manière transparente sans modifier beaucoup de code.

➔ Exemple

```
function createFlower(name) {
    const obj = new Object();
    obj.name = name;
    obj.showlowerName = function () {
        return "I am " + obj.name;
    }
    return obj;
}

const flowerOne = createFlower('rose');
const flowerTwo = createFlower('lily');

console.log("Using Factory Pattern methode  - flowerOne Description: ", flowerOne);
console.log("using factory Pattern methode  - flolwerOne longueur : ", Object.getOwnPropertyNames(flowerOne).length);
console.log("using factory Pattern methode  - flowerone typeof :", typeof flowerOne);

console.log("Using Factory Pattern methode  - flowerTwo Description: ", flowerTwo);
console.log("using factory Pattern methode  - flolwerTwo longueur : ", Object.getOwnPropertyNames(flowerTwo).length);
console.log("using factory Pattern methode  - flowerTwo typeof :", typeof flowerTwo);

console.log("Using Factory Pattern methode - creation objet flowerOne : ", flowerOne.showlowerName());
console.log("Using Factory Pattern methode - creation objet FlowerTwo : ", flowerTwo.showlowerName());
```

```
Using Factory Pattern methode  - flowerOne Description:
▼ {name: 'rose', showLowerName: ƒ} i
    name: "rose"
  ▶ showlowerName: ƒ ()
  ▶ [[Prototype]]: Object
using factory Pattern methode  - flolwerOne longueur :  2
using factory Pattern methode  - flowerone typeof : object
Using Factory Pattern methode  - flowerTwo Description:
▼ {name: 'lily', showLowerName: ƒ} i
    name: "lily"
  ▶ showlowerName: ƒ ()
  ▶ [[Prototype]]: Object
using factory Pattern methode  - flolwerTwo longueur :  2
using factory Pattern methode  - flowerTwo typeof : object
Using Factory Pattern methode - creation objet flowerOne :  I am rose
Using Factory Pattern methode - creation objet FlowerTwo :  I am lily
>
```

Objet JavaScript utilisant un modèle de prototype :

Le modèle prototype est un modèle basé sur l'héritage prototypique,
L'objet prototype agit comme un modèle de plan directeur à partir d'autres objets en héritant lorsque le constructeur les instancie.
Le prototype ajoute des propriétés de l'objet prototype ces propriétés sont ensuite disponible et partagées entre toutes les instances.

➔ Ou utiliser le modèle de prototype en JS

La fonction prototype est utilisé pour augmenter les performances
Il est utilisé pour fournir l'extension aux opérations de base de données en créant un objet pour les autres parties de l'application logicielle – si d'autres parties de l'application ont besoin d'accéder a cet objet, cela peut être fait en créant une copie superficielle de l'objet d'origine ou de l'objet précédemment créé.

➔ Les inconvénients du modèle prototype en JS

L'utilisation de ''this '' peut créer de la confusion.
Il nécessite à la fois un constructeur et un prototype de fonction.

```
// prototype function  syntax
// constructor
function flower(name) {
    this.name = name;
}

// prototype function
// Extend Object

flower.prototype.showFlowerName = function () {
    return 'this is ' + this.name;
}

// calling prototype
// create instance and call methods
const flowerOne = new flower('rose');
const flowerTwo = new flower('lila');

var flower1 = flowerOne.showFlowerName();
var flower2 = flowerTwo.showFlowerName();

console.log(" ------------ using prototype pattern method ---------- ");
console.log("Object flowerone : ", flowerOne);
console.log(" length : " + Object.getOwnPropertyNames(flowerOne).length + "\n type : " + typeof flowerOne + "\n object create of flowerOne : " + flower1);

console.log(" ---------------------------- ");
console.log("Object flowerTwo : ", flowerTwo);
console.log(" length : " + Object.getOwnPropertyNames(flowerTwo).length + "\n type : " + typeof flowerTwo + "\n object create of flowerOne : " + flower2);
```

```
------------ using prototype pattern method ----------
Object flowerone :  ▼ flower {name: 'rose'}
                      name: "rose"
                    ▼ [[Prototype]]: Object
                      ▶ showFlowerName: ƒ ()
                      ▶ constructor: ƒ flower(name)
                      ▶ [[Prototype]]: Object
length : 1
type : object
object create of flowerOne : this is rose
------------------------------
Object flowerTwo :  ▼ flower {name: 'lila'}
                      name: "lila"
                    ▼ [[Prototype]]: Object
                      ▶ showFlowerName: ƒ ()
                      ▶ constructor: ƒ flower(name)
                      ▶ [[Prototype]]: Object
length : 1
type : object
object create of flowerOne : this is lila
>
```