

Exceptions

1. Erreurs et exceptions

Un programme informatique peut générer deux types d'erreurs :

1. les erreurs qui sont détectées lors de la phase de compilation (s'il s'agit d'un langage compilé)
2. les erreurs qui se produisent lors de l'exécution du programme : on les appelle *exceptions*

Le programme ci-dessous ne pose pas de problème lors de la compilation :

```
function puissanceRec(x: number, n: number): number {  
  if (n === 0) return 1;  
  else return x * puissanceRec(x, n - 1);  
}
```

Cependant l'appel suivant génère une exception :

```
puissanceRec(10, -2);
```

Cette exception est détectée lors de l'exécution du programme. Il s'agit d'une exception de type **RangeError** qui est levée par *deno* suite à l'exécution de la fonction **puissanceRec** lorsqu'elle est appelée avec un paramètre **n** négatif. Cette exception est capturée par le programme et affichée dans la console. **L'exécution du programme est alors immédiatement stoppée.**

```
error: Uncaught RangeError: Maximum call stack size exceeded  
  if (n === 0) return 1;  
  ^  
  at puissanceRec (file:///home/laroche5/Cours/BUT1/src/TestRecherche.ts:13:5)  
  at puissanceRec (file:///home/laroche5/Cours/BUT1/src/TestRecherche.ts:14:21)  
  at puissanceRec (file:///home/laroche5/Cours/BUT1/src/TestRecherche.ts:14:21)  
  ...
```

Remarque : il se peut aussi qu'un programme ne produise aucune erreur à la compilation ni aucune exception pendant l'exécution, mais :

- que son comportement soit erroné,
- ou qu'il soit algorithmiquement faux, mais donne une exécution correcte grâce à une largesse du langage utilisé.

Voyez-vous le problème avec la fonction ci-dessous ?

```
function rechercheErronee(t: Array<number>, x: number): boolean {  
    for (let i = 0; i <= t.length; i++) {  
        if (t[i] === x) return true;  
    }  
    return false;  
}  
  
console.log(rechercheErronee([1, 2, 3, 4, 5], 6));
```

2. L'instruction `try...catch`

L'instruction `try...catch` permet de capturer une exception. Le ou les lignes de code qui peuvent lever une exception sont à mettre dans le premier bloc (`try`), et le code constituant la réaction à cette exception est à mettre dans le second bloc (`catch`).

```
try {  
    puissanceRec(10, -2);  
} catch (e) {  
    console.log(e.message);  
}
```

L'intérêt est que cela permet au programme de continuer de se dérouler. Sans le bloc `try...catch`, le programme se serait arrêté !

Dans cet exemple, lorsque l'exception est détectée, le problème est simplement signalé à l'utilisateur.

Cependant, dans certains cas, la partie `catch` du bloc va permettre au programmeur de solutionner l'erreur (exemple : en cas de saisie erronée de l'utilisateur, on utilise un booléen pour faire boucler la saisie.)

3. Levée d'exceptions par le programmeur

Lors de l'analyse du programme `puissanceRec` vu ci-dessus, on a détecté qu'il n'est pas utilisable avec un second paramètre négatif. Le code que nous avons fourni n'est pas sûr. Deux types d'améliorations possibles :

- soit on fournit un code qui fonctionne également avec une puissance négative ;
- soit on prévient l'utilisateur que notre code n'est pas prévu pour cela.

```
// Fonction originelle  
function puissanceRecPeuSure(x: number, n: number): number {
```

```

    if (n === 0) return 1;
    else return x * puissanceRec(x, n - 1);
}

function puissanceRec(x: number, n: number): number {
    if (n === 0) return 1;
    else if (n < 0) throw new Error("n doit être positif");
    else return x * puissanceRec(x, n - 1);
}

```

L'instruction `throw` permet de lancer une exception. Elle prend en paramètre un objet qui représente l'exception. Cet objet doit être une instance d'une classe qui hérite de la classe `Error`.

Quel est l'intérêt ? Les deux fonctions génèrent une exception si on les utilise avec une puissance négative !

- pour le programmeur, mon code est plus sûr : en le lisant, un programmeur verra tout de suite que la puissance négative lève une exception ;
- pour l'utilisateur, le message de l'exception est plus clair :

```

// rappel avant d'utiliser throw
error: Uncaught RangeError: Maximum call stack size exceeded
    if (n === 0) return 1;

```

```

// avec l'ajout du throw
error: Uncaught Error: n doit être positif
    else if (n < 0) throw new Error("n doit être positif");

```

4. Les classes d'exceptions

Les classes d'exceptions sont des classes qui héritent de la classe `Error`. Elles sont utilisées pour représenter des exceptions spécifiques. Il existe plusieurs classes d'exceptions définies dans la bibliothèque standard de TypeScript :

- `Error` : classe parente de toutes les classes d'exceptions
- `RangeError` : tableau trop grand, récursivité non maîtrisée
- `TypeError` : appel d'une fonction sur `undefined`
- `URIError` : erreur dans une url lors d'un accès réseau
- `ReferenceError`
- `SyntaxError`
- ...

L'instruction `throw` peut prendre en paramètre une instance d'une classe d'exception.

```
throw new RangeError("n doit être positif");
```

Lors de la phase `catch`, on peut tester si l'exception est de type `RangeError` :

```
try {  
  puissanceRec(10, -2);  
} catch (e) {  
  if (e instanceof RangeError) {  
    console.log("Erreur de type RangeError");  
  } else {  
    console.log(e.message);  
  }  
}
```

Intérêt : réagir différemment selon le type de l'exception détectée.

5. Définition d'une classe d'exception

On peut définir ses propres classes d'exceptions. Cela peut servir à mieux documenter son code, à le rendre plus clair, à bien expliciter le type de l'exception.

```
class RecursivitéInfinie extends Error {  
  constructor(message: string) {  
    super(message);  
    this.name = "RécurivitéInfinie";  
  }  
}
```

Remarque : `extends` signifie que la classe `RecursivitéInfinie` hérite de la classe `Error`. la fonction `super` permet d'appeler le constructeur de la classe parente. `this.name` permet de modifier le nom de la classe.

Exemple d'utilisation :

```
function puissanceRec(x: number, n: number): number {  
  if (n === 0) return 1;  
  else if (n < 0) throw new RecursivitéInfinie("n doit être positif");  
  else return x * puissanceRec(x, n - 1);  
}
```

6. L'instruction `try...catch` avec une classe d'exception définie par l'utilisateur

L'instruction `try...catch` peut capturer une exception d'une classe d'exception définie par l'utilisateur. Il n'y a aucune différence par rapport au traitement d'une exception définie par TypeScript.

```
try {
    puissanceRec(10, -2);
} catch (e) {
    if (e instanceof RecursivitéInfinie) {
        console.log("Erreur de type RecursivitéInfinie");
    } else {
        console.log((e as Error).message);
    }
}
```

Remarque : `e as Error` permet de convertir l'objet `e` en objet de type `Error`. Sinon le compilateur considère que `e` est de type `any` et ne peut donc pas accéder à la propriété `message`.

7. L'instruction `finally`

L'instruction `finally` permet d'exécuter un bloc de code à la fin d'un bloc `try...catch`.

Elle est utilisée pour exécuter du code quel que soit le cas :

- exception levée,
- exception non levée.

Par exemple, si vous travaillez avec une base de données, le bloc `finally` va permettre de fermer la connexion au serveur, qui doit se faire dans tous les cas, que la requête lancée ait échoué ou qu'elle ait réussi. Plus généralement, toute ressource allouée dans le code peut être libérée dans ce bloc (fermeture de fichier, déconnexion d'une ressource réseau, etc.)

```
let laConnexion;
try {
    laConnexion = await Connexion.getInstance().getConnection();
    const lignes = await laConnexion.query(
        "SELECT * FROM promotion ORDER BY id",
    );
} catch (e) {
    console.log((e as Error).message);
} finally {
    if (laConnexion) laConnexion.close();
}
```

