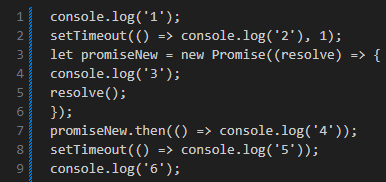
1).

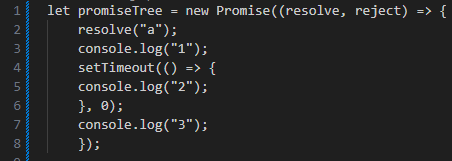


1. 1я строка – **выводит 1 в консоль**
2. 2я строка – **устанавливает таймер, чтобы вывести 2 через 1 миллисекунду**
3. 3я-6я строки – **создаёт новый Promise. Однако, 5я строка вызывается сразу, поэтому 3 выводится в консоль сразу же**
4. 7я строка – **добавляет обработчик для разрешения обещания, который выводит 4 в консоль. Поскольку обещание уже разрешено, этот обработчик будет вызван сразу же**
5. 8я строка – **устанавливает таймер, чтобы вывести 5 в консоль**
6. 9я строка – **выводит в консоль 6**

Таким образом, порядок вывода в консоли будет следующим:

**1, 3, 6, 4, 2, 5**

2).



1. 1я строка – **создаёт новый Promise с функцией-исполнителем, которая принимает два аргумента**
2. 2я строка – **вызывается сразу же. Это означает, что обещание разрешается со значением “а”.**
3. 3я строка – **выводит 1 в консоль. Однако, это происходит сразу после вызова ‘resolve(“a”);’, так что, не отражается в обещании.**
4. 4я-6я строки – **устанавливает таймер, чтобы вывести 2 через 0 миллисекунд. Это добавляет задачу в очередь макрозадач, которая будет выполнена после завершения текущей макрозадачи.**
5. 7я строка – **выводит 3 в консоль**

Рассмотрим события:

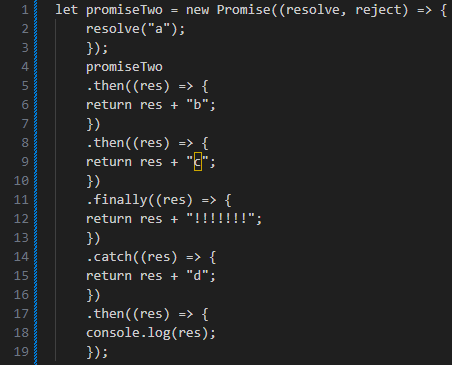
* После вызова **‘resolve(“a”);’**, обещание **‘promiseTree’** разрешается, и вызывается обработчик **‘.then()’**, если он присутствует. Однако, здесь обработчик не определён.
* Выполняются команды **‘console.log(“1”);’** и **‘console.log(“3”);’** , выводя 1 и 3 соответственно.
* Выполняется обработчик таймера, установленного в

**‘setTimeout(() => { console.log(“2”); }, 0);’**, и 2 выводится в консоль

Таким образом, порядок вывода в консоли будет следующим:

**1, 3, 2**

3).



1. 1я строка – **создаёт новый Promise с функцией-исполнителем, которая принимает два аргумента**
2. 2я строка – **внутри исполнителя вызывается ‘resolve(“a”);’, что разрешает обещание со значением “a”**
3. 5я строка – **вызывается метод ‘.then()’, который добавляет обработчик к этому обещанию. Обработчик принимает результат разрешения обещания “a” и добавляет к нему строку “b”**
4. 8я строка – **к полученному значению добавляется строка “c” в следующем обработчике метода ‘.then()’**
5. 11я строка – **метод ‘.finally()’ не принимает результат разрешения, поэтому при разрешении обещания его обработчик не изменяет результат, но выполняется всегда, независимо от того, разрешено ли обещание или возникла ошибка**
6. 14я строка – **если в каком-то из предыдущих обработчиков возникнет ошибка или вызовется метод ‘reject()’, выполнение перейдёт к обработчику метода ‘.catch()’. Здесь такого не происходит**
7. 17я строка – **вызывается последний метод ‘.then()’, который выводит результат в консоль**

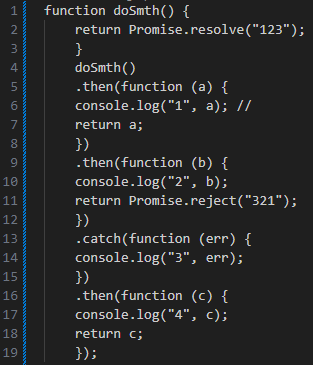
Получается, что результат выполнения кода зависит от порядка выполнения обещаний и обработчиков, которые вызываются последовательно. В данном случае, обещание разрешается успешно и все обработчики **‘then()’** выполняются последовательно.

Метод **‘.finally()’** не изменяет результат разрешения обещания и не влияет на последующие обработчики.

Таким образом, порядок вывода в консоли будет следующим:

**abc**

4).



1. 1я-2я строки – **функция возвращает обещание, которое сразу разрешается значением 123 с помощью ‘Promise.resolve(“123”)’**
2. 5я-7я строки – **выполняется метод ‘.then()’, который добавляет обработчик к разрешённому обещанию. В этом обработчике выводится сообщение 1 вместе с полученным значением ‘a’, которое равно 123. Затем возвращается значение ‘a’**
3. 9я-11я строки – **следующий метод ‘.then()’ добавляет обработчик к предыдущему обещанию. В этом обработчике выводится сообщение 2 вместе с полученным значением ‘b’, которое также равно 123. Затем возвращается отклонённое обещание ‘Promise.reject(“321”)’**
4. 13я-14я строки – **если в процессе выполнения цепочки обещаний возникает ошибка или вызывается метод ‘.reject()’, выполнение переходит к методу ‘.catch()’. Здесь в методе ‘.catch()’ выводится сообщение 3 вместе с полученной ошибкой ‘err’, которая равна 321**
5. 16я-18я строки – **следующий метод ‘.then()’ добавляет обработчик к предыдущему отловленному исключению. Однако, поскольку в предыдущем обещании была вызвана ‘Promise.reject(“321”)’, которое не было разрешено, а было отклонено, значение ‘c’ в этом обработчике будет ‘undefined’. Поэтому выводится сообщение 4 undefined, а не 4 321**

Метод **‘.catch()’** обрабатывает только отклонения в цепочке обещаний, и после него выполнение продолжается с методом **‘.then()’**.

Таким образом, порядок вывода в консоли будет следующим:

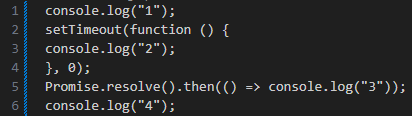
**1 123**

**2 123**

**3 321**

**4 undefined**

5).



1. 1я строка – **выводит 1 в консоль**
2. 2я-4я строки – **устанавливает таймер для выполнения функции через 0 миллисекунд. Важно отметить, что это не означает немедленное выполнение, а добавляет задачу в очередь событий, которая будет выполнена после завершения текущей макрозадачи**
3. 5я строка – **создаёт обещание, которое немедленно разрешается и добавляет обработчик, который выводит 3 в консоль. Поскольку обработка обещания является микрозадачей, она будет выполнена перед выполнением следующей макрозадачи**
4. 6я строка – **выводит 4 в консоль**

Рассмотрим события:

* Выполняется **‘console.log(“1”);’**, выводя 1 в консоль
* Создаётся микрозадача, связанная с обещанием **‘Promise.resolve()’**. Поскольку микрозадачи имеют более высокий приоритет, чем макрозадачи, обработчик

**‘.then(() = > console.log(“3”))’** выполняется сразу после разрешения обещания, выводя 3 в консоль

* Макрозадача, связанная с **‘setTimeout’**, добавляется в очередь событий. Поскольку таймер установлен на 0 миллисекунд, он будет выполнен сразу после выполнения текущей макрозадачи.
* Выполняется **‘console.log(“4”);’**, выводя 4 в консоль

Таким образом, порядок вывода в консоли будет следующим:

**1, 4, 3, 2**