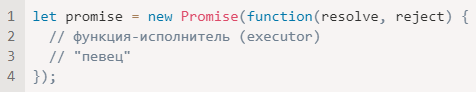
1. Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети.
2. Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции.
3. Promise– это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе.

Синтаксис создания Promise:



Функция, переданная в конструкцию new Promise, называется исполнитель (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат.

Её аргументы resolve и reject – это колбэки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.

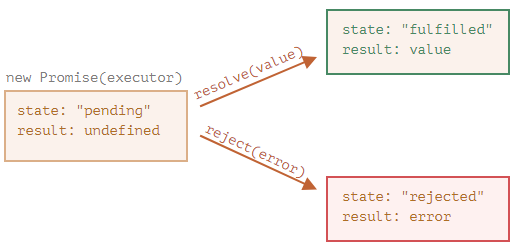
Когда он получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:

* resolve(value) — если работа завершилась успешно, с результатом value.
* reject(error) — если произошла ошибка, error – объект ошибки.

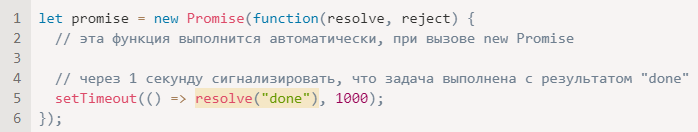
У объекта promise, возвращаемого конструктором new Promise, есть внутренние свойства:

* state («состояние») — вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve или на "rejected" («выполнено с ошибкой») при вызове reject.
* result («результат») — вначале undefined, далее изменяется на value при вызове resolve(value) или на error при вызове reject(error).

Так что исполнитель по итогу переводит promise в одно из двух состояний:



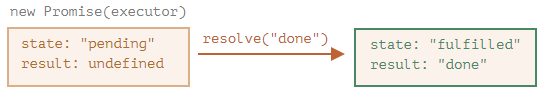
Пример конструктора Promise и простого исполнителя с кодом, дающим успешный результат с задержкой (через setTimeout):



Мы можем наблюдать две вещи, запустив код выше:

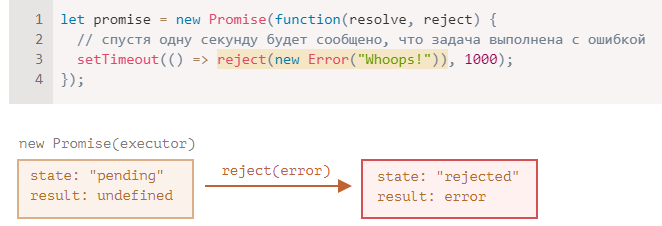
1. Функция-исполнитель запускается сразу же при вызове new Promise.
2. Исполнитель получает два аргумента: resolve и reject — это функции, встроенные в JavaScript, поэтому нам не нужно их писать. Нам нужно лишь позаботиться, чтобы исполнитель вызвал одну из них по готовности.

Спустя одну секунду «обработки» исполнитель вызовет resolve("done"), чтобы передать результат:



В результате мы получили «успешно выполненный» промис.

Пример, в котором исполнитель сообщит, что задача выполнена с ошибкой:



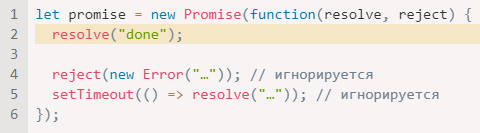
Промежуточные итоги: исполнитель выполняет задачу (что-то, что обычно требует времени), затем вызывает resolve или reject, чтобы изменить состояние соответствующего Promise.

Промис – и успешный, и отклонённый будем называть «завершённым», в отличие от изначального промиса «в ожидании».

**Может быть что-то одно: либо результат, либо ошибка**

Исполнитель должен вызвать что-то одно: resolve или reject. Состояние промиса может быть изменено только один раз.

Все последующие вызовы resolve и reject будут проигнорированы:



Идея в том, что задача, выполняемая исполнителем, может иметь только один итог: результат или ошибку.

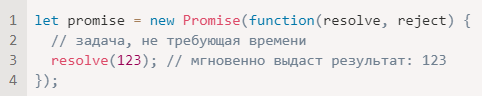
Также заметим, что функция resolve/reject ожидает только один аргумент (или ни одного). Все дополнительные аргументы будут проигнорированы.

**Вызывайте reject с объектом Error**

В случае, если что-то пошло не так, мы должны вызвать reject. Это можно сделать с аргументом любого типа (как и resolve), но рекомендуется использовать объект Error (или унаследованный от него).

**Вызов resolve/reject сразу**

Обычно исполнитель делает что-то асинхронное и после этого вызывает resolve/reject, то есть через какое-то время. Но это не обязательно, resolve или reject могут быть вызваны сразу:



Это может случиться, например, когда мы начали выполнять какую-то задачу, но тут же увидели, что ранее её уже выполняли, и результат закеширован.

Такая ситуация нормальна. Мы сразу получим успешно завершённый Promise.

**Свойства state и result – внутренние**

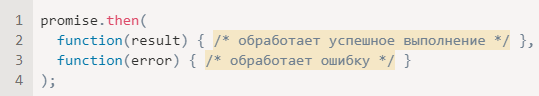
Свойства state и result – это внутренние свойства объекта Promise и мы не имеем к ним прямого доступа. Для обработки результата следует использовать методы .then/.catch/.finally

**[Потребители: then, catch](https://learn.javascript.ru/promise-basics" \l "potrebiteli-then-catch)**

Объект Promise служит связующим звеном между исполнителем («создающим» кодом) и функциями-потребителями, которые получат либо результат, либо ошибку. Функции-потребители могут быть зарегистрированы (подписаны) с помощью методов .then и .catch.

### [then](https://learn.javascript.ru/promise-basics" \l "then)

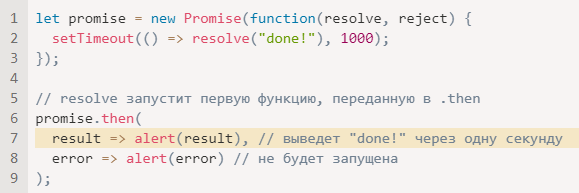
Синтаксис:



Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат.

Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку.

Например, успешно выполненный промис:



## [Очистка: finally](https://learn.javascript.ru/promise-basics" \l "ochistka-finally)

По аналогии с блоком finally из обычного try {...} catch {...}, у промисов также есть метод finally.

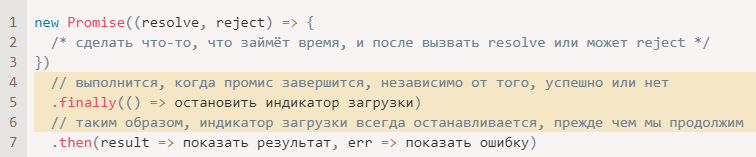
Вызов .finally(f) похож на .then(f, f), в том смысле, что f выполнится в любом случае, когда промис завершится: успешно или с ошибкой.

Идея finally состоит в том, чтобы настроить обработчик для выполнения очистки/доведения после завершения предыдущих операций.

Например, остановка индикаторов загрузки, закрытие больше не нужных соединений и т.д.

Думайте об этом как о завершении вечеринки. Независимо от того, была ли вечеринка хорошей или плохой, сколько на ней было друзей, нам все равно нужно (или, по крайней мере, мы должны) сделать уборку после нее.

Например:

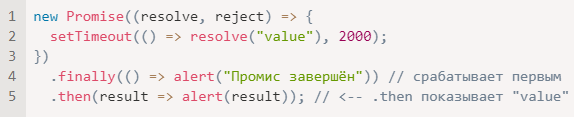


Обратите внимание, что finally(f) – это не совсем псевдоним then(f,f), как можно было подумать.

Есть важные различия:

* 1. Обработчик, вызываемый из finally, не имеет аргументов. В finally мы не знаем, как был завершён промис. И это нормально, потому что обычно наша задача – выполнить «общие» завершающие процедуры.
  2. Обработчик finally «пропускает» результат или ошибку дальше, к последующим обработчикам.

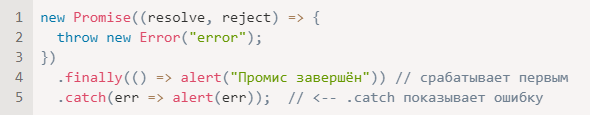
Например, здесь результат проходит через finally к then:



Как вы можете видеть, значение возвращаемое первым промисом, передается через finally к следующему then.

Это очень удобно, потому что finally не предназначен для обработки результата промиса. Как уже было сказано, это место для проведения общей очистки, независимо от того, каков был результат.

А здесь ошибка из промиса проходит через finally к catch:



* 1. Обработчик finally также не должен ничего возвращать. Если это так, то возвращаемое значение молча игнорируется.

Единственным исключением из этого правила является случай, когда обработчик finally выдает ошибку. Затем эта ошибка передается следующему обработчику вместо любого предыдущего результата.

итог:

* Обработчик finally не получает результат предыдущего обработчика (у него нет аргументов). Вместо этого этот результат передается следующему подходящему обработчику.
* Если обработчик finally возвращает что-то, это игнорируется.
* Когда finally выдает ошибку, выполнение переходит к ближайшему обработчику ошибок.

Эти функции полезны и заставляют все работать правильно, если мы используем finally так, как предполагается: для общих процедур очистки.

**На завершённых промисах обработчики запускаются сразу**

Если промис в состоянии ожидания, обработчики в .then/catch/finally будут ждать его.

Иногда может случиться так, что промис уже выполнен, когда мы добавляем к нему обработчик.

В таком случае эти обработчики просто запускаются немедленно:

