**Лабораторная работа №25**

**Тема работы:** Разработка программ с применением promises.

**Цель работы:** закрепить навыки по работе с promises.

**Теоретические сведения**

**Промисы**

Представьте, что вы известный певец, которого фанаты постоянно донимают расспросами о предстоящем сингле.

Чтобы получить передышку, вы обещаете разослать им сингл, когда он будет выпущен. Вы даёте фанатам список, в который они могут записаться. Они могут оставить там свой e-mail, чтобы получить песню, как только она выйдет. И даже больше: если что-то пойдёт не так, например, в студии будет пожар и песню выпустить не выйдет, они также получат уведомление об этом.

Все счастливы! Вы счастливы, потому что вас больше не донимают фанаты, а фанаты могут больше не беспокоиться, что пропустят новый сингл.

Это аналогия из реальной жизни для ситуаций, с которыми мы часто сталкиваемся в программировании:

1. Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети. В нашей аналогии это – «певец».
2. Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции. Это – «фанаты».
3. Promise (по англ. promise, будем называть такой объект «промис») – это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе. В терминах нашей аналогии – это «список для подписки». «Создающий» код может выполняться сколько потребуется, чтобы получить результат, а промис делает результат доступным для кода, который подписан на него, когда результат готов.

Аналогия не совсем точна, потому что объект Promise в JavaScript гораздо сложнее простого списка подписок: он обладает дополнительными возможностями и ограничениями. Но для начала и такая аналогия хороша.

Синтаксис создания Promise:

let promise = new Promise(function(resolve, reject) {

// функция-исполнитель (executor)

// "певец"

});

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. В терминах нашей аналогии: исполнитель – это «певец».

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.

Итак, исполнитель запускается автоматически, он должен выполнить работу, а затем вызвать resolve или reject.

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:

Позже мы рассмотрим, как «фанаты» узнают об этих изменениях.

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

let promise = new Promise(function(resolve, reject) {

// эта функция выполнится автоматически, при вызове new Promise

// через 1 секунду сигнализировать, что задача выполнена с результатом "done"

setTimeout(() => resolve("done"), 1000);

});

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:

Это был пример успешно выполненной задачи, в результате мы получили «успешно выполненный» промис.

А теперь пример, в котором исполнитель сообщит, что задача выполнена с ошибкой:

let promise = new Promise(function(resolve, reject) {

// спустя одну секунду будет сообщено, что задача выполнена с ошибкой

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

Подведём 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:

let promise = new Promise(function(resolve, reject) {

resolve("done");

reject(new Error("…")); // игнорируется

setTimeout(() => resolve("…")); // игнорируется

});

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). Почему так? Скоро нам станет понятно.

**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:

let promise = new Promise(function(resolve, reject) {

// задача, не требующая времени

resolve(123); // мгновенно выдаст результат: 123

});

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, про них речь пойдёт дальше.

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

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

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

Наиболее важный и фундаментальный метод – .then.

Синтаксис:

promise.then(

function(result) { /\* обработает успешное выполнение \*/ },

function(error) { /\* обработает ошибку \*/ }

);

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, вот реакция на успешно выполненный промис:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done!"), 1000);

});

// resolve запустит первую функцию, переданную в .then

promise.then(

result => alert(result), // выведет "done!" через одну секунду

error => alert(error) // не будет запущена

);

Выполнилась первая функция.

А в случае ошибки в промисе – выполнится вторая:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

// reject запустит вторую функцию, переданную в .then

promise.then(

result => alert(result), // не будет запущена

error => alert(error) // выведет "Error: Whoops!" спустя одну секунду

);

Если мы заинтересованы только в результате успешного выполнения задачи, то в then можно передать только одну функцию:

let promise = new Promise(resolve => {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(alert); // выведет "done!" спустя одну секунду

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

Если мы хотели бы только обработать ошибку, то можно использовать null в качестве первого аргумента: .then(null, errorHandlingFunction). Или можно воспользоваться методом .catch(errorHandlingFunction), который сделает тоже самое:

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Ошибка!")), 1000);

});

// .catch(f) это тоже самое, что promise.then(null, f)

promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду

Вызов .catch(f) – это сокращённый, «укороченный» вариант .then(null, f).

**[finally](https://learn.javascript.ru/promise-basics" \l "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.

finally хорошо подходит для очистки, например остановки индикатора загрузки, его ведь нужно остановить вне зависимости от результата.

Например:

new Promise((resolve, reject) => {

/\* сделать что-то, что займёт время, и после вызвать resolve/reject \*/

})

// выполнится, когда промис завершится, независимо от того, успешно или нет

.finally(() => остановить индикатор загрузки)

.then(result => показать результат, err => показать ошибку)

Но это не совсем псевдоним then(f,f), как можно было подумать. Существует несколько важных отличий:

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:

new Promise((resolve, reject) => {

setTimeout(() => resolve("result"), 2000)

})

.finally(() => alert("Промис завершён"))

.then(result => alert(result)); // <-- .then обработает результат

А здесь ошибка из промиса проходит через finally к catch:

new Promise((resolve, reject) => {

throw new Error("error");

})

.finally(() => alert("Промис завершён"))

.catch(err => alert(err)); // <-- .catch обработает объект ошибки

Это очень удобно, потому что finally не предназначен для обработки результата промиса. Так что он просто пропускает его через себя дальше.

Мы более подробно поговорим о создании цепочек промисов и передаче результатов между обработчиками в следующей главе.

1. Последнее, но не менее значимое: вызов .finally(f) удобнее, чем .then(f, f) – не надо дублировать функции f.

**На завершённых промисах обработчики запускаются сразу**

Если промис в состоянии ожидания, обработчики в .then/catch/finally будут ждать его. Однако, если промис уже завершён, то обработчики выполнятся сразу:

// при создании промиса он сразу переводится в состояние "успешно завершён"

let promise = new Promise(resolve => resolve("готово!"));

promise.then(alert); // готово! (выведется сразу)

Теперь рассмотрим несколько практических примеров того, как промисы могут облегчить нам написание асинхронного кода.

**[Пример: loadScript](https://learn.javascript.ru/promise-basics" \l "loadscript)**

У нас есть функция loadScript для загрузки скрипта из предыдущей главы.

Давайте вспомним, как выглядел вариант с колбэками:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Ошибка загрузки скрипта ${src}`));

document.head.append(script);

}

Теперь перепишем её, используя Promise.

Новой функции loadScript более не нужен аргумент callback. Вместо этого она будет создавать и возвращать объект Promise, который перейдет в состояние «успешно завершён», когда загрузка закончится. Внешний код может добавлять обработчики («подписчиков»), используя .then:

function loadScript(src) {

return new Promise(function(resolve, reject) {

let script = document.createElement('script');

script.src = src;

script.onload = () => resolve(script);

script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));

document.head.append(script);

});

}

Применение:

let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(

script => alert(`${script.src} загружен!`),

error => alert(`Ошибка: ${error.message}`)

);

promise.then(script => alert('Ещё один обработчик...'));

Сразу заметно несколько преимуществ перед подходом с использованием колбэков:

**Цепочка промисов**

Давайте вернёмся к ситуации из главы [Введение: колбэки](https://learn.javascript.ru/callbacks): у нас есть последовательность асинхронных задач, которые должны быть выполнены одна за другой. Например, речь может идти о загрузке скриптов. Как же грамотно реализовать это в коде?

Промисы предоставляют несколько способов решения подобной задачи.

В этой главе мы разберём цепочку промисов.

Она выглядит вот так:

new Promise(function(resolve, reject) {

setTimeout(() => resolve(1), 1000); // (\*)

}).then(function(result) { // (\*\*)

alert(result); // 1

return result \* 2;

}).then(function(result) { // (\*\*\*)

alert(result); // 2

return result \* 2;

}).then(function(result) {

alert(result); // 4

return result \* 2;

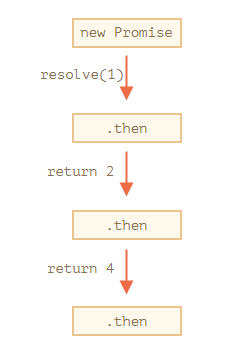
});

Идея состоит в том, что результат первого промиса передаётся по цепочке обработчиков .then.

Поток выполнения такой:

1. Начальный промис успешно выполняется через 1 секунду (\*),
2. Затем вызывается обработчик в .then (\*\*).
3. Возвращаемое им значение передаётся дальше в следующий обработчик .then (\*\*\*)
4. …и так далее.

В итоге результат передаётся по цепочке обработчиков, и мы видим несколько alert подряд, которые выводят: 1 → 2 → 4.



Всё это работает, потому что вызов promise.then тоже возвращает промис, так что мы можем вызвать на нём следующий .then.

Когда обработчик возвращает какое-то значение, то оно становится результатом выполнения соответствующего промиса и передаётся в следующий .then.

**Классическая ошибка новичков: технически возможно добавить много обработчиков .then к единственному промису. Но это не цепочка.**

Например:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve(1), 1000);

});

promise.then(function(result) {

alert(result); // 1

return result \* 2;

});

promise.then(function(result) {

alert(result); // 1

return result \* 2;

});

promise.then(function(result) {

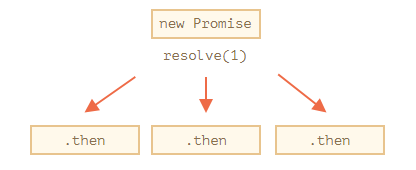
alert(result); // 1

return result \* 2;

});

Мы добавили несколько обработчиков к одному промису. Они не передают друг другу результаты своего выполнения, а действуют независимо.

Вот картина происходящего (сравните это с изображением цепочки промисов выше):



Все обработчики .then на одном и том же промисе получают одно и то же значение – результат выполнения того же самого промиса. Таким образом, в коде выше все alert показывают одно и то же: 1.

На практике весьма редко требуется назначать несколько обработчиков одному промису. А вот цепочка промисов используется куда чаще.

**[Возвращаем промисы](https://learn.javascript.ru/promise-chaining" \l "vozvraschaem-promisy)**

Обработчик handler, переданный в .then(handler), может вернуть промис.

В этом случае дальнейшие обработчики ожидают, пока он выполнится, и затем получают его результат.

Например:

new Promise(function(resolve, reject) {

setTimeout(() => resolve(1), 1000);

}).then(function(result) {

alert(result); // 1

return new Promise((resolve, reject) => { // (\*)

setTimeout(() => resolve(result \* 2), 1000);

});

}).then(function(result) { // (\*\*)

alert(result); // 2

return new Promise((resolve, reject) => {

setTimeout(() => resolve(result \* 2), 1000);

});

}).then(function(result) {

alert(result); // 4

});

Здесь первый .then показывает 1 и возвращает новый промис new Promise(…) в строке (\*). Через одну секунду этот промис успешно выполняется, и его результат (аргумент в resolve, то есть result \* 2) передаётся обработчику в следующем .then. Он находится в строке (\*\*), показывает2 и делает то же самое.

Таким образом, как и в предыдущем примере, выводятся 1 → 2 → 4, но сейчас между вызовами alert существует пауза в 1 секунду.

Возвращая промисы, мы можем строить цепочки из асинхронных действий.

**[Пример: loadScript](https://learn.javascript.ru/promise-chaining" \l "primer-loadscript)**

Давайте используем эту возможность вместе с промисифицированной функцией loadScript, созданной нами в [предыдущей главе](https://learn.javascript.ru/promise-basics" \l "loadscript), чтобы загружать скрипты по очереди, последовательно:

loadScript("/article/promise-chaining/one.js")

.then(function(script) {

return loadScript("/article/promise-chaining/two.js");

})

.then(function(script) {

return loadScript("/article/promise-chaining/three.js");

})

.then(function(script) {

// вызовем функции, объявленные в загружаемых скриптах,

// чтобы показать, что они действительно загрузились

one();

two();

three();

});

Этот же код можно переписать немного компактнее, используя стрелочные функции:

loadScript("/article/promise-chaining/one.js")

.then(script => loadScript("/article/promise-chaining/two.js"))

.then(script => loadScript("/article/promise-chaining/three.js"))

.then(script => {

// скрипты загружены, мы можем использовать объявленные в них функции

one();

two();

three();

});

Здесь каждый вызов loadScript возвращает промис, и следующий обработчик в .then срабатывает, только когда этот промис завершается. Затем инициируется загрузка следующего скрипта и так далее. Таким образом, скрипты загружаются один за другим.

Мы можем добавить и другие асинхронные действия в цепочку. Обратите внимание, что наш код всё ещё «плоский», он «растёт» вниз, а не вправо. Нет никаких признаков «адской пирамиды вызовов».

Технически мы бы могли добавлять .then напрямую к каждому вызову loadScript, вот так:

loadScript("/article/promise-chaining/one.js").then(script1 => {

loadScript("/article/promise-chaining/two.js").then(script2 => {

loadScript("/article/promise-chaining/three.js").then(script3 => {

// эта функция имеет доступ к переменным script1, script2 и script3

one();

two();

three();

});

});

});

Этот код делает то же самое: последовательно загружает 3 скрипта. Но он «растёт вправо», так что возникает такая же проблема, как и с колбэками.

Разработчики, которые не так давно начали использовать промисы, иногда не знают про цепочки и пишут код именно так, как показано выше. В целом, использование цепочек промисов предпочтительнее.

Иногда всё же приемлемо добавлять .then напрямую, чтобы вложенная в него функция имела доступ к внешней области видимости. В примере выше самая глубоко вложенная функция обратного вызова имеет доступ ко всем переменным script1, script2, script3. Но это скорее исключение, чем правило.

**Thenable**

Если быть более точными, обработчик может возвращать не именно промис, а любой объект, содержащий метод .then, такие объекты называют «thenable», и этот объект будет обработан как промис.

Смысл в том, что сторонние библиотеки могут создавать свои собственные совместимые с промисами объекты. Они могут иметь свои наборы методов и при этом быть совместимыми со встроенными промисами, так как реализуют метод .then.

Вот пример такого объекта:

class Thenable {

constructor(num) {

this.num = num;

}

then(resolve, reject) {

alert(resolve); // function() { native code }

// будет успешно выполнено с аргументом this.num\*2 через 1 секунду

setTimeout(() => resolve(this.num \* 2), 1000); // (\*\*)

}

}

new Promise(resolve => resolve(1))

.then(result => {

return new Thenable(result); // (\*)

})

.then(alert); // показывает 2 через 1000мс

JavaScript проверяет объект, возвращаемый из обработчика .then в строке (\*): если у него имеется метод then, который можно вызвать, то этот метод вызывается, и в него передаются как аргументы встроенные функции resolve и reject, вызов одной из которых потом ожидается. В примере выше происходит вызов resolve(2) через 1 секунду (\*\*). Затем результат передаётся дальше по цепочке.

Это позволяет добавлять в цепочки промисов пользовательские объекты, не заставляя их наследовать от Promise.

**Порядок выполнения работы**

Задача 1

Встроенная функция setTimeout использует колбэк-функции. Создайте альтернативу, использующую промисы.

Функция delay(ms) должна возвращать промис, который перейдёт в состояние «выполнен» через ms миллисекунд, так чтобы мы могли добавить к нему .then:

function delay(ms) {

// ваш код

}

delay(3000).then(() => alert('выполнилось через 3 секунды'));