**Функции-обёртки, декораторы**

JavaScript предоставляет удивительно гибкие возможности по работе с функциями: их можно передавать, в них можно записывать данные как в объекты, у них есть свои встроенные методы…

Конечно, этим нужно уметь пользоваться. В этой главе, чтобы более глубоко понимать работу с функциями, мы рассмотрим создание функций-обёрток или, иначе говоря, «декораторов».

[Декоратор](http://en.wikipedia.org/wiki/Decorator_pattern) – приём программирования, который позволяет взять существующую функцию и изменить/расширить ее поведение.

*Декоратор* получает функцию и возвращает обертку, которая делает что-то своё «вокруг» вызова основной функции.

**[bind – привязка контекста](http://learn.javascript.ru/decorators" \l "bind-privyazka-konteksta)**

Один простой декоратор вы уже видели ранее – это функция [bind](http://learn.javascript.ru/bind):

function bind(func, context) {

return function() {

return func.apply(context, arguments);

};

}

Вызов bind(func, context) возвращает обёртку, которая ставит this и передаёт основную работу функции func.

**[Декоратор-таймер](http://learn.javascript.ru/decorators" \l "dekorator-taymer)**

Создадим более сложный декоратор, замеряющий время выполнения функции.

Он будет называться timingDecorator и получать функцию вместе с «названием таймера», а возвращать – функцию-обёртку, которая измеряет время и прибавляет его в специальный объект timer по свойству-названию.

Использование:

function f(x) {} // любая функция

var timers = {}; // объект для таймеров

// отдекорировали

f = timingDecorator(f, "myFunc");

// запускаем

f(1);

f(2);

f(3); // функция работает как раньше, но время подсчитывается

alert( timers.myFunc ); // общее время выполнения всех вызовов f

При помощи декоратора timingDecorator мы сможем взять произвольную функцию и одним движением руки прикрутить к ней измеритель времени.

Его реализация:

var timers = {};

// прибавит время выполнения f к таймеру timers[timer]

function timingDecorator(f, timer) {

return function() {

var start = performance.now();

var result = f.apply(this, arguments); // (\*)

if (!timers[timer]) timers[timer] = 0;

timers[timer] += performance.now() - start;

return result;

}

}

// функция может быть произвольной, например такой:

var fibonacci = function f(n) {

return (n > 2) ? f(n - 1) + f(n - 2) : 1;

}

// использование: завернём fibonacci в декоратор

fibonacci = timingDecorator(fibonacci, "fibo");

// неоднократные вызовы...

alert( fibonacci(10) ); // 55

alert( fibonacci(20) ); // 6765

// ...

// в любой момент можно получить общее количество времени на вызовы

alert( timers.fibo + 'мс' );

Обратим внимание на строку (\*) внутри декоратора, которая и осуществляет передачу вызова:

var result = f.apply(this, arguments); // (\*)

Этот приём называется «форвардинг вызова» (от англ. forwarding): текущий контекст и аргументы через applyпередаются в функцию f, так что изнутри f всё выглядит так, как была вызвана она напрямую, а не декоратор.

**[Декоратор для проверки типа](http://learn.javascript.ru/decorators" \l "dekorator-dlya-proverki-tipa)**

В JavaScript, как правило, пренебрегают проверками типа. В функцию, которая должна получать число, может быть передана строка, булево значение или даже объект.

Например:

function sum(a, b) {

return a + b;

}

// передадим в функцию для сложения чисел нечисловые значения

alert( sum(true, { name: "Вася", age: 35 }) ); // true[Object object]

Функция «как-то» отработала, но в реальной жизни передача в sum подобных значений, скорее всего, будет следствием программной ошибки. Всё-таки sum предназначена для суммирования чисел, а не объектов.

Многие языки программирования позволяют прямо в объявлении функции указать, какие типы данных имеют параметры. И это удобно, поскольку повышает надёжность кода.

В JavaScript же проверку типов приходится делать дополнительным кодом в начале функции, который во-первых обычно лень писать, а во-вторых он увеличивает общий объем текста, тем самым ухудшая читаемость.

**Декораторы способны упростить рутинные, повторяющиеся задачи, вынести их из кода функции.**

Например, создадим декоратор, который принимает функцию и массив, который описывает для какого аргумента какую проверку типа применять:

// вспомогательная функция для проверки на число

function checkNumber(value) {

return typeof value == 'number';

}

// декоратор, проверяющий типы для f

// второй аргумент checks - массив с функциями для проверки

function typeCheck(f, checks) {

return function() {

for (var i = 0; i < arguments.length; i++) {

if (!checks[i](arguments[i])) {

alert( "Некорректный тип аргумента номер " + i );

return;

}

}

return f.apply(this, arguments);

}

}

function sum(a, b) {

return a + b;

}

// обернём декоратор для проверки

sum = typeCheck(sum, [checkNumber, checkNumber]); // оба аргумента - числа

// пользуемся функцией как обычно

alert( sum(1, 2) ); // 3, все хорошо

// а вот так - будет ошибка

sum(true, null); // некорректный аргумент номер 0

sum(1, ["array", "in", "sum?!?"]); // некорректный аргумент номер 1

Конечно, этот декоратор можно ещё расширять, улучшать, дописывать проверки, но… Вы уже поняли принцип, не правда ли?

**Один раз пишем декоратор и дальше просто применяем этот функционал везде, где нужно.**

**[Декоратор проверки доступа](http://learn.javascript.ru/decorators" \l "dekorator-proverki-dostupa)**

И наконец посмотрим ещё один, последний пример.

Предположим, у нас есть функция isAdmin(), которая возвращает true, если у посетителя есть права администратора.

Можно создать декоратор checkPermissionDecorator, который добавляет в любую функцию проверку прав:

Например, создадим декоратор checkPermissionDecorator(f). Он будет возвращать обертку, которая передает вызов f в том случае, если у посетителя достаточно прав:

function checkPermissionDecorator(f) {

return function() {

if (isAdmin()) {

return f.apply(this, arguments);

}

alert( 'Недостаточно прав' );

}

}

Использование декоратора:

function save() { ... }

save = checkPermissionDecorator(save);

// Теперь вызов функции save() проверяет права

**[Итого](http://learn.javascript.ru/decorators" \l "itogo)**

Декоратор – это обёртка над функцией, которая модифицирует её поведение. При этом основную работу по-прежнему выполняет функция.

**Декораторы можно не только повторно использовать, но и комбинировать!**

Это кардинально повышает их выразительную силу. Декораторы можно рассматривать как своего рода «фичи» или возможности, которые можно «нацепить» на любую функцию. Можно один, а можно несколько.

Скажем, используя декораторы, описанные выше, можно добавить к функции возможности по проверке типов данных, замеру времени и проверке доступа буквально одной строкой, не залезая при этом в её код, то есть (!) не увеличивая его сложность.

Предлагаю вашему вниманию задачи, которые помогут выяснить, насколько вы разобрались в декораторах. Далее в учебнике мы ещё встретимся с ними.