Камешки, банки, рамки... Ну и ну!

Одна из метафор, которые, по моему опыту, хорошо помогают понять смысл областей видимости, — цветные камешки, которые раскладываются по банкам соответствующих цветов.

Представьте, что у вас есть куча красных, синих и зеленых камешков. Вы хотите разложить все камешки по банкам: красные кладутся в красную банку, зеленые — в зеленую, а синие — в синюю. Если после сортировки вам понадобится зеленый камешек, вы уже знаете, что его нужно искать в зеленой банке.

В этой метафоре камешки представляют переменные в нашей программе. Банки соответствуют областям видимости (функциям и блокам), которым мы назначили разные цвета просто для целей обсуждения. Таким образом, цвет каждого камешка определяется *цветом* области видимости, в которой этот камешек был изначально создан.

Разметим пример программы из главы 1 цветами:

```
// Внешняя/глобальная область видимости: КРАСНЫЙ

var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    // Функциональная область видимости: СИНИЙ

    for (let student of students) {
        // Область видимости цикла: ЗЕЛЕНЫЙ

        if (student.id == studentID) {
            return student.name;
        }
    }
}
```

```
var nextStudent = getStudentName(73);
console.log(nextStudent); // Suzy
```

Мы обозначили три цвета области видимости комментариями: КРАСНЫЙ (внешняя/глобальная область видимости), СИНИЙ (область видимости функции `getStudentName(..)`) и ЗЕЛЕНЫЙ (область видимости цикла `for`). Но по листингу может быть трудно распознать границы областей видимости. Чтобы вам было проще наглядно представить области видимости, на рис. 2 они обозначены рамками:

```
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    for (let student of students) {
        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// "Suzy"
```

1 2 3

Рис. 2. Границы областей видимости

1. Рамка 1 (КРАСНЫЙ) охватывает глобальную область видимости, которая содержит три идентификатора/переменные:

`students` (строка 1), `getStudentName` (строка 8) и `nextStudent` (строка 16).

2. Рамка 2 (СИНИЙ) охватывает область видимости функции `getStudentName(..)` (строка 8), которая содержит всего один идентификатор/переменную: параметр `studentID` (строка 8).
3. Рамка 3 (ЗЕЛЕНЫЙ) охватывает область видимости цикла `for` (строка 9), которая содержит всего один идентификатор/переменную: `student` (строка 9).

Формально параметр `studentID` не совсем принадлежит области видимости СИНИЙ (2). Эта неоднозначность будет рассмотрена в разделе «Предполагаемые области видимости» приложения А. А пока будет достаточно отнести `studentID` к области видимости СИНИЙ (2).

Границы областей видимости определяются во время компиляции на основании того, где находятся функции/блоки видимости, как они вложены друг в друга и т. д. Каждая область видимости полностью содержится в родительской области видимости — она никогда не принадлежит двум внешним областям видимости.

Цвет каждого камешка (переменной/идентификатора) определяется цветом банки, в которой он находится (объявляется), а не цветом области видимости, из которой к нему может происходить обращение (например, `students` в строке 9 и `studentID` в строке 10).

Вспомните: в главе 1 было сказано, что `id`, `name` и `log` являются свойствами, а не переменными; иначе говоря, это не камешки, разложенные по банкам, поэтому им не назначаются цвета по правилам, описанным в книге. О том, как обрабатываются обращения к свойствам, рассказано в третьей книге серии, «Объекты и классы».

Когда движок JS обрабатывает программу (во время компиляции) и обнаруживает объявление переменной, он фактически спрашивает: «В каком *цвете* (рамке, банке) я сейчас нахожусь?» Пере-

менная обозначается тем же *цветом*, что означает, что она принадлежит этой рамке/банке.

Область видимости ЗЕЛЕНЫЙ (3) полностью вложена в область СИНИЙ (2); аналогичным образом область СИНИЙ (2) полностью вложена в область КРАСНЫЙ (1). Области видимости могут вкладываться друг в друга так, как показано, до произвольной глубины, необходимой вашей программе.

Ссылки (не объявления) на переменные/идентификаторы считаются допустимыми в том случае, если подходящее объявление существует либо в текущей области видимости, либо в любой области видимости выше текущей, но не в области видимости более низкого уровня.

Выражение в области видимости КРАСНЫЙ (1) может обращаться только к переменным из КРАСНЫЙ (1), но не СИНИЙ (2) или ЗЕЛЕНЫЙ (3). Выражение в области видимости СИНИЙ (2) может обращаться к переменным из СИНИЙ (2) или КРАСНЫЙ (1), но не ЗЕЛЕНЫЙ (3). Наконец, выражение в области видимости ЗЕЛЕНЫЙ (3) может обращаться к переменным из КРАСНЫЙ (1), СИНИЙ (2) и ЗЕЛЕНЫЙ (3).

Процесс определения этих цветов во время выполнения можно на концептуальном уровне представить себе как поиск. Так как ссылка на переменную `students` в цикле `for` в строке 9 не является объявлением, цвет у нее отсутствует. Соответственно, мы спрашиваем у текущей области видимости СИНИЙ (2), присутствует ли в ней камешек с заданным именем. Так как его нет, поиск продолжается во внешней/вмещающей области видимости: КРАСНЫЙ (1). В банке КРАСНЫЙ (1) лежит камешек с именем `students`, так что ссылка на переменную `students` в цикле идентифицируется как КРАСНЫЙ (1).

Команда `if (student.id == studentID)` в строке 10 аналогичным образом содержит ссылки на переменную ЗЕЛЕНЫЙ (3) с именем `student` и переменную СИНИЙ (2) с именем `studentID`.

Движок JS на самом деле не определяет эти «цвета» во время выполнения; использованный в тексте термин «поиск» — всего лишь риторический прием, который позволяет понять суть концепций. Во время компиляции большинство ссылок на переменные соответствует уже известным областям видимости, так что их «цвет» уже определен и он хранится с каждой ссылкой для предотвращения лишнего поиска во время выполнения программы. Подробнее об этом нюансе рассказано в главе 3.

Ключевые выводы из аналогии с камешками и банками (и рамками):

- Переменные объявляются в конкретных областях видимости, что можно рассматривать как цветные камешки, разложенные по банкам соответствующих цветов.
- Любая ссылка на переменную, находящаяся в области видимости, в которой она была объявлена, или в одной из областей видимости более глубокой вложенности, будет помечена как относящаяся к тому же цвету — если только промежуточная область видимости не «заместит» объявление переменной; см. раздел «Затенение» главы 3.
- Определение «цветов» областей видимости и находящихся в них переменных происходит во время компиляции. Эта информация используется для «поиска» переменных (определения цвета камешков) во время выполнения кода.

Дружеское общение

Другая полезная метафора для процесса анализа переменных и областей видимости, из которых они происходят, — «диалоги», происходящие внутри движка в ходе обработки и последующего выполнения кода. Мы можем «прислушаться» к этим диалогам, чтобы лучше понять на концептуальном уровне, как работают области видимости.

Послушаем, что говорят участники процесса в ходе обработки нашей программы:

- *Движок*: отвечает за компиляцию и выполнение ваших программ JavaScript.
- *Компилятор*: один из друзей *Движка*; выполняет всю тяжелую работу по разбору и генерированию кода (см. предыдущий раздел).
- *Менеджер области видимости*: еще один друг *Движка*; собирает и ведет список поиска всех объявленных переменных/идентификаторов и поддерживает набор правил, определяющих их доступность для текущего выполняемого кода.

Чтобы вы в полной мере поняли, как работает JavaScript, необходимо начать думать так, как думает Движок (и его друзья), задавать те вопросы, которые задают они, и давать такие же ответы на вопросы. Чтобы проанализировать это общение, вернемся к нашему постоянному примеру:

```
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    for (let student of students) {
        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// Suzy
```

Посмотрим, как JS будет обрабатывать эту программу, начиная с первой команды. Массив и его содержимое — обычные литералы-значения JS (а следовательно, проблемы области видимости их вообще не касаются), поэтому нас здесь будет интересовать

только объявление `var students = [ .. ]` и инициализация/присваивание.

Обычно мы рассматриваем эту команду как единое целое, но с точки зрения *Движка* это не так. Для JS это две разные операции: одна выполняется *Компилятором* во время компиляции, а другая выполняется *Движком* во время выполнения.

Обработка этой программы *Компилятором* начинается с лексического разбора и разделения ее на лексемы, которые затем преобразуются в дерево (AST).

После того как *Компилятор* доберется до генерирования кода, ему приходится учитывать многие неочевидные подробности. Разумно предположить, что *Компилятор* сгенерирует для первой команды код, который означает примерно следующее: выделить память для переменной, связать ее с именем `students` и сохранить в этой переменной ссылку на массив. Однако это не все.

Ниже перечислены операции, которые будут выполнены *Компилятором* при обработке этой команды:

1. Встречая в программе конструкцию `var students`, *Компилятор* приказывает *Менеджеру области видимости* проверить, существует ли в этой конкретной области видимости переменная с именем `students`. Если она существует, то *Компилятор* игнорирует это объявление и двигается дальше. В противном случае *Компилятор* генерирует код, который (во время выполнения) прикажет *Менеджеру области видимости* создать новую переменную с именем `students` в этой области видимости.
2. Затем *Компилятор* генерирует код, который будет позднее выполнен *Движком*, для обработки присваивания `students = []`. Код, выполняемый *Движком*, сначала прикажет *Менеджеру области видимости* проверить, доступна ли в текущей области видимости переменная с именем `students`. Если она недоступна, *Движок* продолжает поиск в других местах (см. ниже раздел «Вложенные области видимости»). Когда *Движок* найдет переменную, оно присваивает ей ссылку на массив `[ .. ]`.

В форме диалога между *Компилятором* и *Движком* первая фаза компиляции программы выглядит примерно так:

Компилятор: Эй, *Менеджер области видимости* (для глобальной области видимости), я нашел формальное объявление идентификатора с именем `students`. Когда-нибудь слышал о таком?

Менеджер области видимости (глобальный): Нет, впервые слышу. Вот создаю специально для тебя.

Компилятор: Эй, *Менеджер области видимости*, я нашел формальное объявление идентификатора с именем `getStudentName`. Когда-нибудь слышал о таком?

Менеджер области видимости (глобальный): Тоже нет, но создам для тебя.

Компилятор: Эй, *Менеджер области видимости*, `getStudentName` указывает на функцию, так что нам понадобится новая область видимости.

Менеджер области видимости (для функции): Понял, получай новую область видимости.

Компилятор: Эй, *Менеджер области видимости* (для функции), я нашел объявление формального параметра для `studentID`. Слышал о таком?

Менеджер области видимости (для функции): Нет, но создал его в этой области видимости.

Компилятор: Эй, *Менеджер области видимости* (для функции), я нашел цикл `for`, которому нужна своя область видимости.

...

Диалог состоит из вопросов и ответов. *Компилятор* спрашивает текущий *Менеджер области видимости*, встречалось ли ранее объявление обнаруженного им идентификатора. Если оно не встречалось, то *Менеджер области видимости* создает эту переменную в своей области видимости. Если встречалось, то оно

фактически игнорируется, потому что *Менеджеру области видимости* делать ничего не нужно.

Компилятор также сигнализирует, когда он переходит между областями видимости функций или блоков, чтобы были созданы новые экземпляры области видимости и *Менеджера области видимости*.

Позднее, когда начнется выполнение программы, начнется диалог *Движка* с *Менеджером области видимости*, и этот диалог может выглядеть примерно так:

Движок: Эй, *Менеджер области видимости* (глобальный), ты можешь найти идентификатор `getStudentName`, чтобы я мог присвоить ему эту функцию?

Менеджер области видимости (глобальный): Да, вот переменная.

Движок: Эй, *Менеджер области видимости*, я нашел ссылку-приемник для `students`, ты когда-нибудь слышал о таком?

Менеджер области видимости (глобальный): Да, он был формально объявлен для этой области видимости. Вот, держи.

Движок: Спасибо. Я инициализирую переменную `students` значением `undefined`, так что она готова к использованию. Эй, *Менеджер области видимости* (глобальный), я нашел ссылку-приемник для переменной `nextStudent`, знаешь о ней?

Менеджер области видимости (глобальный): Да, она была формально объявлена для этой области видимости. Держи.

Движок: Спасибо. Я инициализирую переменную `nextStudents` значением `undefined`, так что она готова к использованию. Эй, *Менеджер области видимости* (глобальный), я нашел ссылку-источник для `getStudentName`, знаешь о ней?

Менеджер области видимости (глобальный): Да, она была формально объявлена для этой области видимости. Держи.

Движок: Прекрасно, значение в `getStudentName` является функцией, поэтому я ее выполню.

Движок: Эй, *Менеджер области видимости*, теперь нам нужно создать экземпляр области видимости функции.

...

Этот диалог — еще один обмен вопросами и ответами, в котором *Движок* сначала приказывает *Менеджеру области видимости* провести поиск поднятого (hoisted) идентификатора `getStudentName`, чтобы связать с ним функцию. Затем *Движок* продолжает спрашивать *Менеджера области видимости* о ссылке-приемнике для `students` и т. д.

Ниже приводится краткая сводка обработки команд вида `var students = [ .. ]` за два этапа:

1. *Компилятор* создает объявление переменной в области видимости (так как она еще не была ранее объявлена в текущей области видимости).
2. Во время работы *Движка* для обработки присваивания в этой команде *Движок* дает команду *Менеджеру области видимости* провести поиск переменной и инициализировать ее `undefined`, чтобы она была готова к использованию, после чего присваивает ему значение-массив.

Вложенная область видимости

Когда наступает время выполнить функцию `getStudentName()`, *Движок* запрашивает у *Менеджера области видимости* экземпляр области видимости для этой функции, затем переходит к поиску параметра (`studentID`), чтобы присвоить ему значение аргумента 73, и т. д.

Область видимости функции `getStudentName(..)` вложена в глобальную область видимости. Блоковая область видимости для цикла `for` аналогичным образом вложена в область видимости этой функции. Области видимости могут вкладываться друг в друга на произвольную глубину так, как определяет программа.

Каждая область видимости получает собственный экземпляр *Менеджера области видимости* при каждом ее выполнении (один или несколько раз). Каждая область видимости автоматически регистрирует все свои идентификаторы в начале выполнения (это называется поднятием переменных, см. главу 5).

В начале области видимости, если какой-либо идентификатор поступил из объявления функции, эта переменная автоматически инициализируется ссылкой на ассоциированную функцию. А для любого идентификатора, поступающего из объявления `var` (в отличие от `let`/`const`), переменная автоматически инициализируется `undefined`, чтобы она могла использоваться; в противном случае переменная остается неинициализированной (т. е. в состоянии TDZ, см. главу 5) и не может использоваться до выполнения ее полного объявления и инициализации.

В команде `for (let student of students) {` идентификатор `students` является ссылкой-источником, для которого необходимо провести поиск. Но как выполнить такой поиск, ведь область видимости функции не найдет такой идентификатор?

Чтобы понять это, представим, что происходит такой разговор:

Движок: Эй, *Менеджер области видимости* (*для функции*), я нашел ссылку-источник для `students`, слышал о такой?

Менеджер области видимости (для функции): Нет, впервые слышу. Попробуй следующую внешнюю область видимости.

Движок: Эй, *Менеджер области видимости* (*для глобальной области видимости*), я нашел ссылку-источник для `students`, слышал о такой?

Менеджер области видимости (для глобальной области видимости): Да, было такое формальное объявление. Вот оно.

...

Один из ключевых аспектов лексической области видимости заключается в том, что если в любой момент ссылку на идентификатор не удается найти в текущей области видимости, происходит

обращение к следующей внешней области видимости; процесс повторяется, пока не будет обнаружен ответ или пока не будут проверены все возможные области видимости.

Неудача при поиске

Когда *Движок* завершает перебор всех *лексически доступных* областей видимости (двигаясь наружу), но найти идентификатор так и не удается, возникает ситуация ошибки. Но в зависимости от режима программы (действует строгий режим или нет) и роли переменной (т. е. *приемник* или *источник*; см. главу 1) эта ситуация ошибки будет решена иначе.

Путаница с неопределенностью

Если переменная является источником, безрезультатный поиск идентификатора считается необъявленной (неизвестной, отсутствующей) переменной, что всегда приводит к выдаче ошибки `ReferenceError`. Кроме того, если переменная и код являются приемником, и код выполняется в строгом режиме, переменная считается необъявленной, и в этом случае также выдается ошибка `ReferenceError`.

Сообщение об ошибке для ситуации с необъявленной переменной в большинстве сред JS выглядит так: «Reference Error: XYZ не определен». Слова «не определен» (not defined) в английском языке почти не отличаются от «неопределенный» (undefined). Тем не менее в JS это два совершенно разных понятия, и, к сожалению, это сообщение об ошибке постоянно создает путаницу.

«Не определен» в данном случае означает «не объявлен», т. е. у переменной нет подходящего формального объявления в любой *лексически доступной* области видимости. С другой стороны, «неопределенный» в действительности означает, что переменная была найдена (объявлена), но на данный момент она не содержит другого значения, поэтому по умолчанию в ней хранится значение `undefined`.