Часть 1 (Предисловие)

Предисловие

О строгом режиме

В томе JavaScript I был рассмотрен Pure JavaScript (ECMAScript 5th Edition). С появлением этого стандарта у программиста появилась возможность указать реализации языка, что сценарий необходимо интерпретировать в строгом соответствии со спецификацией языка. Или, как говорится, в строгом режиме.
Это делается при помощи следующей директивы интерпретатору:

```
«use strict»;
```

Эта директива прописывается в коде сценария в начале той области видимости, в которой сценарий следует интерпретировать в строгом режиме (будь то глобальная или локальная область видимости). Будучи примененной в целевой области видимости, эта директива включает строгий режим на время исполнения кода в этой области видимости. Необходимо помнить, что директивы обратного действия нет.
В строгом режиме:
нельзя опускать точку с запятой в записи оператора;
переменные перед любым использованием должны объявляться;
нельзя пользоваться восьмеричкой в литералах;
доступ к членам объектов по запрещенным для них операциям ведет к ошибке;
нет оператора `with`;
функции, вызванные как функции, получают в `this` ссылку на `undefinied`;

Часть 2 (ECMAScript 6)

О стандарте

Этот стандарт успешно внедрен, и должен быть рассмотрен.
Стандарт `ECMAScript6` предусматривает следующие нововведения.

О строгом режиме

Строгий и нестрогий режимы по прежнему поддерживаются различными интерпретаторами. Разумеется, логика работы в строгом режиме в стандарте `ECMAScript6` обнаруживает некоторые различия с тем, что подразумевает строгий режим в стандарте `ECMAScript5`. Эти различия будут описаны.

Объявление переменных

Переменные могут объявляться как обычно (записью `var VarName;`), и тогда их область видимости регулируется как обычно; или же они могут быть объявлены новыми средствами (в соответствующих формах записи), и тогда становится доступна блочная область видимости.
При объявлении переменной (записью `let VarName;`) она будет доступна в своем блоке кода. При объявлении переменной (записью `const VarName;`) она также будет доступна в своем блоке кода, но при этом она не может быть реинициализирована (при этом, если инициализация не прописана при объявлении, то это приведет к выбросу (синтаксической) ошибки); если эта запись используется для объявления объектов, то ссылка на объект не может быть изменена, но сам объект может быть изменен (соответствующими средствами языка).
При использовании различных средств объявления переменных необходимо помнить следующее — повторное объявление записью `var` допустимо (если речь идет только о записи `var`); однако, повторное объявление прочими записями не допустимо (даже если для первого объявления задействовалась запись `var`). Также следует помнить, что переменные, объявленные записями `let` и `const`, не могут быть использованы (ни по чтению, ни по записи) до их объявления; попытка их использования (до записи объявления) приведет к выбросу ошибки (ошибки ссылки). Поэтому область кода до их объявления принято называть «временной мертвой зоной» (это мертвая зона в плане использования этих переменных).

Строки и регулярные выражения

О регулярных выражениях

Флаг `u` объекта `RegExp` позволяет интерпретировать суррогатную пару как один символ (если этот флаг установлен);

Функции работы со строками

`codePointAt()`

используется для возврата значения кодового пункта (по указанной позиции символа в строке); принимает параметр — значение (указанной позиции символа в строке); возвращает значение кодового пункта (по указанной позиции символа в строке); возвращает значение (это численное значение);
этот метод поддерживает работу с вспомогательными многоязыковыми плоскостями (в таблице кодов символов);

`String.fromCodePoint()`

используется для возврата строки, состоящей из символа (указанного кодовым пунктом); принимает параметр — значение (указанного кодового пункта); возвращает строку, состоящую из символа (указанного кодовым пунктом); возвращает строку;
этот метод поддерживает работу с вспомогательными многоязыковыми плоскостями (в таблице кодов символов);

`normalize()`

используется для нормализации строк; принимает параметры — ряд значений (см. др. источники); осуществляет нормализацию своей строки (см. др. источники); возвращает значение (см. др. источники);

Функции

Об объявлении функций

Стандарт `ECMAScript6` позволяет объявлять функции и в пределах блоков кода — будучи объявленной в блоке, функция локализуется в нем (это при работе в строгом режиме, в нестрогом режиме функции просто будут объявлены в инкапсуллирующей функции или глобальной области видимости).

Параметры по умолчанию

Стандарт ECMAScript6 позволяет производить передачу параметров по умолчанию специально предназначенными для этого средствами. Запись объявления функции при применении этих средств имеет вид:

```
function FuncName(VarName0, … , VarNameN=Value) { … }
```

Параметры, передаваемые по умолчанию, объявляются обычной для этого записью (по крайней мере, в C- образных языках). Таких параметров может быть и несколько, но они должны быть в конце списка параметров. Сам язык допускает и невыполнение этого правила — параметры по умолчанию могут быть и не в конце списка параметров. Тогда, если идущие за ними параметры передаются, то передача параметров по умолчанию (со значениями по умолчанию) обеспечивается передачей им значений undefinied.
Параметры, передаваемые по умолчанию, как обычно, могут быть инициализированы и пользовательскими значениями — тогда эти значения просто передаются явно.

О параметрах функции вообще

Параметры функции — это локальные переменные функции. Их область действия — это область действия функции (параметры создаются при вызове функции, и прекращают существовать по ее завершении). Параметры, которые не были переданы (явно или по умолчанию (в новом или старом стиле)) не создаются, и поэтому не могут быть использованы в теле функции.
В языке JavaScript функция может принимать произвольное количество параметров. В связи с этим целесообразно помнить о соответствующей терминологии — параметры (прописанные в объявлении функции) называются «именованными», а параметры (не прописанные в объявлении функции, и передаваемые избытком) называются «неименованными».

Список параметров переменной длины

Можно задать список параметров переменной длины. Тогда запись области параметров имеет вид:

```
(VarName0, ...VarNameN)
/* в этой записи ...VarNameN прописывается в конце,
таким образом, ...VarNameN можно прописать лишь единожды */
```

В результате будет создан массив VarNameN, который будет использоваться для хранения параметров (кроме тех параметров, что объявлены обычным образом — поэтому параметры, хранящиеся в этом массиве, принято называть остаточными). Остаточные параметры не учитываются счетчиком параметров (FuncName.lenght), но учитываются объектом argumunts.

Объявление списка параметров переменной длины не может применяться при объявлении методов доступа (по записи); это касается методов объектов, объявленных в форме литерала. Это ограничение введено в язык по той причине, что объекты (объявленные литералом) должны обрабатываться интерпретатором достаточно просто (а объявление списка параметров произвольной длины идет вразрез с этим требованием).

Передача массивов функциям

Стандарт ECMAScript6 предусматривает средства, позволяющие передать функции массив, даже если функция не принимает массив.
Это делается при помощи следующей записи:

```
FuncName (...VarName);
// VarName – это объект Array
```

При использовании такой записи массив VarName будет подан функции в виде секвенции его элементов (это делается автоматически). При использовании такой записи необходимо помнить, что массив должен подаваться функции первым из аргументов — остальные аргументы функции, если они передаются, должны следовать после (именно после) него.

О вызове функций

Функции, являющиеся конструкторами, должны вызываться как конструкторы. Стандарт ECMAScript6 не позволяет вызывать функции- конструкторы как обычные функции.

Стрелочные функции

Так называемые «стрелочные функции» это функции, объявляемые специфической записью (с использованием символа =>).
Стрелочные функции обладают следующей спецификой:
они заведомо не являются функциями- конструкторами (и, соответственно, не могут быть вызваны как функции- конструкторы);
они не имеют ссылки на пототип (prototype);
не имеют ссылки на массив аргументов (arguments), и, соответственно, не могут принимать аргументы избытком (кроме остаточных аргументов);

Эта специфика позволяет стрелочным функциям быть относительно простыми (и просто оптимизируемыми интерпретатором).
Стрелочные функции объявляются записью, представляющей из себя альтернативу следующей записи:

```
let VarName0 = function(VarName1, … ) { … };
```

Запись объявления стрелочной функции:

```
let VarName0 = (VarName1, … ) => … ;
// после => прописывается тело функции
```

В этой записи область параметров, как обычно, заключается в скобки. Когда у функции один единственный параметр, скобки можно и опустить (но это единственный случай, когда их можно опустить — даже когда у функции нет параметров скобки обязательны).
Что касается тела функции, то оно может прописываться по разному. Если оно тривиально (его полезная нагрузка составляет одно выражение), то оно может прописываться не как тело функции, а просто как выражение — его значение и будет возвращаемым стрелочной функцией значением. Если же оно не тривиально, то оно должно прописываться как тело функции (включая скобки, и возврат значения). Тело стрелочной функции может быть и пустым.
Стрелочные функции, как и любые функции, могут быть вызваны непосредственно при объявлении, соответствующая запись имеет вид:

```
let VarName0 = ((VarName1, … ) => … )(Value0, … );
```

Объекты

Возможности синтаксиса литералов объектов

Проверка дубликатов

В стандарте `ECMAScript6` в литералах объектов не производится проверка на наличие в них дуплицированных членов. В случае дупликации происходит переопределение одноименных членов — они переопределяются членами, прописанными ниже.

Сокращенный синтаксис инициализации членов

Стандарт `ECMAScript6` поддерживает сокращенный синтаксис инициализации членов объектов. Речь идет об инициализации члена объекта значением одноименной переменной. Разумеется, имена не должны повторяться; но, при инициализации членов объектов это — злостно устоявшаяся практика, и интерпретатор к ней лоялен. Поэтому в язык и введен сокращенный синтаксис инициализации членов объектов. Он заключается в следующем — при инициализации члена объекта значением одноименной переменной достаточно прописать имя члена объекта, а саму инициализацию допустимо не прописывать.
В записи это выглядит так:

```
{VarName0, … , VarNameN}
```

Необходимо помнить, что это допустимо только для литералов объектов.

Сокращенный синтаксис объявления методов

Стандарт ECMAScript6 поддерживает сокращенный синтаксис объявления методов объектов. Он заключается в следующем — при объявлении метода нет необходимости прибегать к классической записи объявления члена — метода объекта — вместо этого достаточно прописать сам этот метод.
В записи это выглядит так:
```
{VarName( … ) { … }}
```

Необходимо помнить, что это допустимо только для литералов объектов.

Сокращенная запись имен членов объектов

Стандарт ECMAScript6 поддерживает сокращенный синтаксис имен членов объектов. Речь идет о записи имен объектов с индексацией. Этот синтаксис заключается в следующем — если имя члена объекта предполагается прописать с индексацией (`ObjectName["VarName"]`), то можно прибегнуть к сокращенной записи (`"VarName"`).
Кроме того, что допустима сокращенная запись, в стандарте ECMAScript6 допустимо прописывать имена членов объектов в кавычках (`"VarName"`) - даже когда речь не идет об индексации.
Необходимо помнить, что это допустимо только для литералов объектов.

Super

Ключевое слово `super` может быть использовано методом для ссылки на непосредственный прототип своего объекта.
Это ключевое слово может применяться только в методе, объявленном в сокращенном синтаксисе.

Методы объектов

Стандарт ECMAScript6 вводит ряд дополнительных методов объектов.
Это следующие методы.

`Object.is()`

используется для проверки на идентичность; принимает параметры — значение, и значение; проверяет свои параметры на идентичность;возвращает логическое значение (`true` – идентичны, `false` – нет);

этот метод исполняет проверку более строго, чем оператор идентичности (`+0` и `-0` этот метод считает разными значениями, значения `NaN` считает одинаковыми значениями);

`Object.assign()`

используется для работы с примесями; принимает параметры — ряд объектов; приписывает первому принятому объекту собственные члены последующих принятых объектов (однако, он не приписывает члены — методы доступа); возвращает объект (первый принятый объект, претерпевший изменения);

`Object.setPrototypeOf()`

используется для замены прототипа объекта; принимает параметры — объект (прототип которого заменяется), и объект (который станет прототипом); заменяет прототип объекта указанным образом; возвращает объект (прототип которого заменен);

Деструктуризация ссылочных типов

Речь идет о деструктуризации собственно объектов и массивов.
Само понятие деструктуризация означает представление экземпляров вышеупомянутых объектов в виде секвенции переменных. Члены деструктурируемых объектов присваиваются переменным из обозначенной секвенции переменных.
Деструктуризация применима только к экземплярам вышеупомянутых объектов, причем, заданным в литеральной форме. Это, видимо, потому, что разработчик языка предполагает, что для деструктуризации должны использоваться объекты, созданные из литерала, а объекты, созданные обычным образом должны использоваться обычным же образом.
Деструктуризация будет распознана синтаксически некорректной, если использование в этой записи деструктуризируемого объекта будет возвращать значение `null` или `undefinied`. По сути, это лишь перефразирование вышеизложенного правила — действительно, если деструктуризируемый объект будет создан из литерала, то его использование не будет возвращать значение `null` или `undefinied`.

Деструктуризация объектов

Запись деструктуризации объекта:

```
let ObjectName = {VarName0: … , … };
// создан объект из литерала

let {VarName0, … } = ObjectName;
```

```
// использована деструктуризация
```

здесь запись `let {VarName0, … } = ObjectName;` и есть полезная нагрузка записи деструктуризации. В результате применения этой записи переменным из секвенции переменных будут присвоены значения одноименных членов объекта. Если в секвенции переменных использовано меньше переменных, чем имеется членов у объекта, то члены объекта (которым не сопоставлены переменные) просто не будут извлекаться ни в какие переменные.
Альтернативная запись:

```
let ObjectName = {VarName0: … , … };
// создан объект из литерала

let VarName0 = … , … ;
// объявлены переменные для использования в деструктуризации

({VarName0, … } = ObjectName);
// использована деструктуризация
```

эта альтернативная запись объявляет секвенцию переменных (используемых в деструктуризации) иначе. Это делает применение `let` непосредственно в деструктуризации избыточным.
Поэтому (чтобы запись была распознана синтаксически корректной) сама деструктуризация должна прописываться именно так — в круглых скобках (иначе же запись деструктуризации будет трактоваться как присваивание литералу объекта, и поэтому будет распознана синтаксически некорректной).
Будучи использованной в каких- бы то ни было формах записи, деструктуризация возвращает значение деструктурируемого объекта (речь идет о том выражении, в котором непосредственно производится деструктуризация).
В секвенции переменных (используемых для деструктуризации) может быть прописано больше переменных, чем имеется в составе деструктуризируемого объекта. Эти переменные получат в результате деструктуризации значение `undefinied`.
Таким переменным можно присвоить значение по умолчаню, это делается непосредственно в выражении деструктуризации:

```
let {VarName0, … , VarNameN=Value} = ObjectName;
```

значение по умолчанию используется также когда в объекте присутствует соответствующий член, но его значение — `undefinied`.
При деструктуризации возможно присваивание и секвенции неодноименных переменных. Такая запись имеет вид:

```
let {VarName0: NewVarName0, … } = ObjectName;
```

но, как видно из записи, имена одноименных переменных все же задействуются. В результате применения этой записи объект будет представлен в виде секвенции переменных `NewVarName0, …` .

При использовании этой возможности присваивание значений по умолчанию по прежнему доступно (запись принимает вид: `VarName0: NewVarName0 = Value, …`).
деструктуризация позволяет извлекать данные и из вложенных объектов:
```
let {VarName0: {NestedVarName0}, … } = ObjectName;
```

при использовании этой возможности по прежнему доступно присваивание неодноименным переменным:
```
let {VarName0: {NestedVarName: NewNestedVarName0}, … } = ObjectName;
```

Деструктуризация массивов

Деструктуризация массивов во многом аналогична деструктуризации объектов (разве что, имеетместо своя специфика).
Деструктуризация массивов затрагивает только элементы массива.
Форма записи:
```
let ArrayName = [ … ];
//создан массив из литерала

let [VarName0, … ] = ArrayName;
// использована деструктуризация
```

если же определенные элементы массива не требуется извлекать, то они просто опускаются в секвенции переменных (но их позиции по прежнему указываются):
```
let [VarName0, , … ] = ArrayName;
```

опущены они могут быть в любой позиции на усмотрение программиста.
Альтернативная запись:
```
let ArrayName = [ … ];
// создан объект (массив) из литерала

let VarName0 = … , … ;
// объявлены переменные для использования в деструктуризации

[VarName0, … ] = ArrayName;
// использована деструктуризация
```

эта запись выглядит именно так, и не требует круглых скобок.
При применении деструктуризации к массивам (так же, как и в случае объектов) доступны значения по умолчанию:
```
let ArrayName = [ … ];
// создан объект (массив) из литерала

let [VarName0=Value, … ] = ArrayName;
// использована деструктуризация
```

При применении деструктуризации к массивам (так же, как и в случае объектов) возможна деструктуризация вложенных массивов:

```
let ArrayName = [ … ];
// создан объект (массив) из литерала
// он содержит вложенный массив

let [VarName0, [ … ], … ] = ArrayName;
// использована деструктуризация
```

При применении деструктуризации к массивам доступна запись:

```
let [VarName0, … , ...VarNameN] = ArrayName;
```

в этой записи переменная `...VarNameN` используется для сохранения элементов массива, для сохранения которых не хватило прописанных до `...VarNameN` переменных. Переменная `...VarNameN` – это массив (создается неявно, самим интерпретатором), и для доступа к сохраненным в нем переменным используется индексация. Сами сохраненные в нем переменные принято называть остаточными элементами.

Переменная для остаточных элементов может быть и единственной в секвенции переменных, но если наряду с ней используются и другие переменные, то переменная для остаточных элементов должна быть последней в их списке.

Всвязи с появлением в языке деструктуризации массивов стало доступно специфическое средство — запись, сходная с записью деструктуризации массивов (она позволяет переупорядочивать массив):

```
let VarName0 = Value0, … , VarNameN = ValueN;
// объявлены переменные, используемые в этой записи

[VarName0, … , VarNameN] = [VarNameN, … , VarName0];
// произведено упорядочивание
```

эта запись весьма необычна, но она называется записью деструктуризации.

Ее действие следующее — выражение, непосредственно производящее деструктуризацию, приводит к упорядочиванию секвенции переменных, и возвращает деструктуризируемый объект (массив).

Смешанная деструктуризация

Речь идет о деструктуризации вложенных друг в друга объектов и массивов.

Для деструктуризации объекта используется запись `… = { … }`, для деструктуризации массива используется запись `… = [ … ]`.

Для доступа к вложенному объекту используется запись `VarName: { … }`, для доступа к вложенному массиву используется запись `… , [ … ]`.

Деструктуризация и функции

Деструктуризация может применяться совместно с функциями как в составе тела функции, так и в следующих специфических средствах.

Деструктуризация и функции. Область параметров

Деструктуризация может быть задействована в области параметров:
```
function FuncName({VarName0, … } = ObjectName) … ;
// стоит обратить внимание на область параметров
```

в области параметров прописано выражение, непосредственно производящее деструктуризацию. Таким образом, в функцию будет передан объект.
Запись будет распознана синтаксически корректной (очевидно, для этого случая и предусмотрена возможность использовать деструктуризацию в такой форме записи при заключении ее в круглые скобки).
Есть и следующий способ использования деструктуризации в области параметров. Программист прибегает к следующей записи:
```
function FuncName(VarName0, … , {VarNameN, … , VarNameZ}) … ;
// здесь область параметров выглядит итого более необычно
```

и при вызове передавать параметры в записи:
```
FuncName(Value0, … , {VarNameN: ValueN, … , VarNameZ: ValueZ});
// и тогда параметры передаются так
```

такие параметры (`{VarNameN, … , VarNameZ}`) принято называть деструктурированными параметрами. Деструктурированные параметры, как обычно, могут быть переданы целиком, либо частично (тогда неинициализированные переменные получат значение `undefinied`). Саму запись вида `{VarNameN, … , VarNameZ}` принято называть шаблоном деструктуризации, так как она определяет результат деструктуризации. Впрочем, запись, определяющая результат деструктуризации, называется шаблоном деструктуризации как при деструктуризации объектов, так и массивов.
Если в объявлении функции прописаны деструктуризованные параметры, то при передаче параметров их нельзя опускать (точнее, ряд переменных в шаблоне деструктуризации может отсутствовать, но саму запись шаблона деструктуризации опускать нельзя). Это потому, что деструктуризованные параметры представляет собой упрощенную запись выражения, непосредственно задействующего деструктуризацию, и если эту запись опустить, то этому пункту в списке параметров будет присвоено значение `undefinied` (а это в данном случае считается синтаксической ошибкой). Сам шаблон деструктуризации, являясь пунктом в списке параметров, может иметь значение по умолчанию (оно задается обычной для этого записью, в применении к шаблону деструктуризации она имеет вид: `{ … } = Value`).
Шаблону деструктуризации в данном случае доступны все возможности шаблонов деструктуризации — они были рассмотрены выше.

Тип данных Symbol

Тип данных `Symbol` это элементарный тип данных. Этот тип данных называется так потому, что предназначен для хранения символов (имен, используемых в сценарии). Переменные типа `Symbol` создаются специфической записью:
```
let VarName = Symbol();
```

это единственный способ объявить переменную типа `Symbol` (литеральной формы у них нет). Значение типа `Symbol` возвращается функцией `Symbol()`; причем, необходимо помнить, что это — просто функция, не предназначенная для использования как конструктор (действительно, ведь речь идет о создании переменной элементарного типа данных, а не объекта). Хотя `Symbol()` - это просто функция, она (как и любая функция) является объектом, и ей доступны ее члены (специфические члены будут рассмотрены далее). Саму функцию `Symbol()` стоит своевременно описать.

`Symbol()`

используется для создания переменных типа `Symbol`; параметров либо не принимает, либо принимает строку (ее содержимое в дальнейшем трактуется как описание символа); создает переменную типа `Symbol` (с учетом принятых араметров); возвращает переменную типа `Symbol` (значение этой переменной формируется самим интерпретатором, и он обеспечивает его уникальность);

Также следует своевременно описать специфические члены этой функции.

`Symbol.for()`

используется для создания символа по указанному описанию; принимает параметр — строку (содержит описание); проверяет, есть ли уже символ с указанным описанием, если есть — то возвращает этот символ, если нет — то создает его и затем возвращает; возвращает значение типа `Symbol`;
этот метод в своей работе задействует системную структуру данных — глобальный реестр символов, по реестру он проверяет наличие символа, и вносит в реестр новые символы;
сама же функция `Symbol()` вообще не работает с этим реестром — нипроверяет наличие символа, ни вносит в него созданные символы;

`Symbol.keyFor()`

используется для возврата описания переменной- символа; принимает параметр — имя переменной- символа; возвращает описание указанной переменной-

символа; возвращает строку (содержит описание; если же символ не внесен в реестр, то возвращается значение `undefinied`);

`etc.`

о прочих членах — см. другие источники;

Использование переменных типа `Symbol` приосходит следующим образом. Создается переменная этого типа (при создании она надляется уникальным значением). Затем она используется для приписывания какому- либо объекту нового члена (запись `ObjectName[VarName] = Value;`). В дальнейшем, для доступа к нему используется запись `ObjectName[VarName]`. Обычная же запись (вида `ObjectName.VarName`, или `ObjectName["VarName"]`) оказывается недоступна. Такие члены объектов не перечисляются усовершенствованным циклом `for`, и не возвращаются методом `Object.getOwnPropertyNames()`. Для возврата таких членов используется другой метод — это метод `Object.getOwnPropetySymbols()` – этот метод принимает параметр — целевой объект; и возвращает его члены, указывемые переменной- символом; возвращает массив (он состоит из запрошенных членов указанного объекта).
В таком использовании и заключается единственное назначение переменных этого типа. То есть, для доступа к члену объекта используется не то, что является именем (форма записи для доступа по имени не доступна), а то, что является его альтрнативой — символ (что и видно в записи); переменная типа `Symbol`, как и говорилось выше, получает значение при создании, и в дальнейшем не может быть реинициализирован. Таким образом, переменная типа `Symbol` позволяет приписать объекту член, защищенный вышеперечисленными мерами от действий в коде сценария.

О приведении типов

Тип данных `Symbol` целесообразно считать не приводимым к другим типам данных. Разработчик языка внедрил этот тип данных для специфического применения (рассмотренного выше), поэтому переменные- символы и не предполагается использовать наряду с переменными прочих типов данных.
Что касается имеющихся возможностей приведения типов — автоматическое приведение типов между типом `Symbol` и прочими типами не выполняется. Хотя, явное приведение в ряде случаев все же возможно. Явное приведение к строке выполняется вызовом `String()` от переменной- символа. С тем же успехом можно выполнить вызов `Object()` от переменной- символа (в результате методом `toString()` будет произведено приведение к строке). Так же, переменная- символ может использоваться в условии операторов управления ходом вычислений — и тогда будет интерпретироваться как логическое значение (любое наличествующее значение трактуется как истина);

причем, такое использование переменной типа `Symbol` в коде сценария — это последний специфический способ явного приведения типа `Symbol`.

Наборы и отображения

Наборы и отображения — это специфические объекты, предоставляемые языком. Они действительно специфические, и их можно даже считать не объектами, а просто синтаксически сходными с ними средствами.

Наборы

Наборы — это объекты `Set`. Эти объекты представляют собой секвенцию неповторяющихся значений (повторяющиеся значения они не хранят; но, это могут быть значения разных типов, приводимые к одному значению). Данные сохраняются в наборах в порядке их добавления. Добавление данных и прочие операции производятся методами наборов.
Создание наборов производится следующей записью:

```
let VarName = new Set();
```

это обычная запись для создания экземпляров ссылочных типов. Следует рассмотреть члены этих объектов.

Конструктор наборов

`Set()`

конструктор наборов; параметров либо не принимает, либо принимает массив (трактуется как секвенция элементов набора); создает набор (руководствуясь переданными параметрами); возвращает набор;
если переданный в параметрах массив содержит повторяющиеся значения, то повторы значений будут отброшены, и длина секвенции элементов набора будет меньше размера массива (это потому, что набор может хранить только уникальные значения);

Методы наборов

`add()`

используется для добавления элементов набора; принимает параметр — значение (это значение (какого- либо типа), добавляемое в набор); добавляет в свой набор указанное значение; возвращает значение `Set.size` (обновленное);

возвращаемое значение следует уточнить для конкретного интерпретатора;
если добавляется уже присутствующее в наборе значение, то такой вызов этого метода просто игнорируется;

`has()`

используется для проверки наличия в наборе указанного значения; принимает параметр — значение (какого- либо типа, это и есть указанное значение); проверяет наличие указанного значения в своем наборе; возвращает логическое значение (`true` – есть, `false` – нет);

`delete()`

используется для удаления указанного значения из набора; принимает параметр — значение (какого- либо типа, это и есть указанное значение); удаляет указанное значение из своего набора; возвращает значение `Set.size` (обновленное);
возвращаемое значение следует уточнить для конкретного интерпретатора;

`clear()`

используется для очистки набора от его элементов; параметров не принимает; очищает свой набор от всех его элементов; возвращает значение `Set.size` (обновленное);
возвращаемое значение следует уточнить для конкретного интерпретатора;

`forEach()`

используется для вызова указанной пользовательской функции от каждого элемента набора; принимает параметры — функцию (это пользовательская функция), и объект (в его контексте и будет исполняться пользовательская функция; этот параметр может быть и не передан); последовательно вызывает пользовательскую функцию от каждого из элементов набора (передаваемая в параметрах пользовательская функция должна быть прописана соответствующим образом — принимать параметр, осуществлять прописанные программистом операции, само тело пользовательской функции может быть тривиальным — оно должно лишь быть прописанным по этим правилам — а корректное применение этой функции по своему набору остается на откуп интерпретатору); возвращает ничтоже;

Переменные наборов

`size`

используется для хранения длины секвенции элементов набора (в количестве элементов секвенции набора);

Приведение набора к массиву

Как видно из вышеописанного, наборы во многом аналогичны массивам. Поэтому предусмотрена возможность приведения набора к массиву. Эта возможность доступна в следующей записи:

```
let VarName0 = new Set( … );
// создан экземпляр набора

VarName1 = [...VarName0];
// создан массив из набора
```

эта возможность часто используется для создания массива уникальных значений.

Наборы со слабыми ссылками на объекты

Наборы со слабыми ссылками используют слабые ссылки на сохраняемые в них объекты (если в них сохраняются объекты). Слабые ссылки на объект работают несколько иначе, чем переменные (используемые для ссылки на объект). При использовании слабых ссылок на объект наблюдается следующее — когда переменная (используемая для ссылки на объект) более не используется для ссылки на объект, то и слабая ссылка (задействуемая набором со слабыми ссылками) более не будет использоваться для ссылки на этот объект. Разумеется, переменных (используемых для ссылки на объект) в коде сценария может быть и несколько — слабая ссылка на объект перестанет работать, когда ни одна переменная (использовавшаяся для ссылки на объект) не будет больше на него указывать.
Наборы со слабыми ссылками создаются записью:

```
let VarName = new WeakSet( … );
```

и используются аналогично обычным наборам, за тем исключением, что они могут хранить только экземпляры ссылочных типов. Кроме того, они не имеют следующих члено `forEach()`, и `size`.

Отображения

Отображения — это объекты `Map`. Эти объекты представляют собой секвенцию пар ключ/значение (повторяющиеся значения они не хранят; но, это могут быть значения разных типов, приводимые к одному значению; ключи тоже могут быть разных типов). Данные сохряняются в отображениях в порядке их добавления. Добавление данных и прочие операции производятся методами отображений.
Создание наборов производится следующей записью:

```
let VarName = new Map();
```

это обычная запись для создания экземпляров ссылочных типов. Следует рассмотреть члены этих объектов.

Конструктор отображений

`Map()`

конструктор отображений; параметров либо не принимает, либо принимает массив (элементы которого — массивы, содержащие два элемента (которые будут трактоваться как ключ и значение соответственно)); создает отображение (руководствуясь переданными параметрами); возвращает нитоже;

Методы отображений

`set()`

используется для добавления элемента в отображение; принимает параметры — значение (ключ элемента отображения), и значение (собственно значение элемента отображения); добавляет элемент в отображение (руководствуясь переданными параметрами); возвращает значение `size` (обновленное);

`get()`

используется для возврата значения элемента отображения по его ключу; принимает параметр — значение (ключа); возвращает значение элемента отображения по указанному ключу; возвращает значение (собственно значение элемента отображения);
если значение по указанному ключу отсутствует, то возвращается специальное значение `undefinied`;

`has()`

используется для проверки наличия в отображении элемента с указанным ключем; принимает параметр — значение (ключа); проверяет наличие в своем отображении элемента с указанным ключем; возвращает логическое значение (`true` – есть, `false` – нет);

`delete()`

используется для удаления элемента отображения с указанным ключем; принимает параметр — значение (ключа); удаляет элемент своего отображения с указанным ключем; возвращает значение `size` (обновленное);
возвращаемое значение следует уточнить для конкретного интерпретатора;

`clear()`

используется для очистки отображения от его элементов; параметров не принимает; удаляет все элементы своего отображения; возвращает значение `size` (обновленное);
возвращаемое значение следует уточнить для конкретного интерпретатора;

`forEach()`

используется для вызова указанной пользовательской функции от каждого элемента отображения; принимает параметры — функцию (это пользовательская функция), и объект (в его контексте и будет исполняться пользовательская функция; этот параметр может быть и не передан); последовательно вызывает пользовательскую функцию от каждого из элементов отображения (передаваемая в параметрах пользовательская функция должна быть прописана соответствующим образом — принимать параметры (для ключа, значения, и ссылки на отображение соответственно), осуществлять прописанные программистом операции, само тело пользовательской функции может быть тривиальным — оно должно лишь быть прописанным по этим правилам — а корректное применение этой функции по своему набору остается на откуп интерпретатору); возвращает ничтоже;
поведение этого метода (взависимости от переданных параметров) следует уточнять для конкретного интерпретатора;

Переменные отображений

`size`

используется дляхранения значения размера отображения в количестве элементов (пар ключ/значение);

Отображения со слабыми ссылками на объекты

Отображения со слабыми ссылками аналогичны наборам со слабыми ссылками.
Такие отображения хранят слабые ссылки на объекты в своих ключах; и, когда приходит пора удалить слабую ссылку, то она удаляется вместе с

сопоставленным ключу отображения значением (то есть, происходит удаление всего элемента отображения).
Отображения со слабыми ссылками создаются записью:
```
let VarName = new WeakMap( … );
```

и используются аналогично обычным отображениям, за тем исключением, что они могут хранить в своих ключах только экземпляры ссылочных типов. Кроме того, они не имеют следующих членов `clear()`, `forEach()`, и `size`.

Итераторы, генераторы

Итераторы и генераторы — это специфические объекты. Они используются в соответствии со своей спецификой (подробности — далее).

Итераторы

Итератор — это специфический объект. Эти объекты используются при произведении итераций по коллекциям в коде сценария. Итератором считается объект, который удовлетворяет требованиям к объектам- итераторам (создается функцией- генератором итераторов, и содержит характерные члены).
Итераторы содержат следующие члены.

Конструктор итераторов

```
FuncName()
```

конструктор итераторов; принимает параметры — ссылку на коллекцию; создает итератор; возвращает ничтоже;
строго говоря, это аппроксимация — такого конструктора нет, а объекты-итераторы создаются не конструктором, а функцией- генератором итераторов (она должна возвращать объект, удовлетворяющий требованиям к итераторам);

Методы итераторов

```
ObjectName.next()
```

используется для возврата объекта результатов (объекта, содержащего значение следующего элемента коллекции, и специфические члены); параметров не принимает; возвращает объект результатов; возвращает объект (это и есть объект результатов);

есть специфическая форма записи этого метода — с принимаемым параметром (этот параметр используется как значение, реинициализирующее значение, прописанное непосредственно за оператором `yield`);

`ObjectName.throw()`

используется для выброса прерывания; параметров не принимает; производит выброс прерывания; возвращает ничтоже;
выброс прерывания осуществляется непосредственно после последнего исполнившегося оператора `yield` функции- генератора (подробности о генераторах — далее), и перехват этого прерывания должен производиться в соответствующем участке кода сценария (включающем этот оператор `yield`);
есть специфическая форма записи этого метода — с принимаемым параметром (этот параметр — объект прерывания, которое и будет выброшено);

Переменные итераторов

Их нет.

Переменные объекта результатов

`ResultsObjectName.value`

используется для ссылки на следующее значение (то есть, значение, возвращаемое методом `next()`, вернувшим данный объект результатов);
если все итерации совершены, то эта переменная содержит специальное значение `undefinied`;
вызов метода `ObjectName.next()` изменяет значение счетчика итераций объекта `ObjectName`, поэтому объект результатов каждый раз содержит актуальное значение переменной `ResultsObjectName.value`;

`ResultsObjectName.done`

используется для индикации — произведены ли итерации по всем элементам коллекции; содержит логическое значение (`true` – произведены);

Прочее

Подробнее об итераторах — далее/в тематических источниках.

Генераторы

Генератор — это специфический объект. Генераторы используются для работы с итераторами (они являются генераторами итераторов).
Генераторы — это объекты- функции с возможностью выхода и повторного входа при сохранении контекста исполнения тела этой функции.
При вызове функции- генератора она возвращает объект- итератор. Вызов из этого итератора метода `next()` приводит к исполнению тела функции-генератора до первого оператора `yield`. Оператор `yeald` либо возвращает значение (запись `yield Value`), либо приводит к вызову другогой функции-генератора (запись `yield* EtcFuncName()`); вызовдругой функции-генератора принято называть «делегированием генераторов». Сам же метод `next()` возвращает объект результатов (в данном случае переменная `value` содержит значение возвращаемое оператором `yield`, переменная `done` обозначает завершение всех итераций по всем элементам коллекции (`true` – завершено)). Последующие вызовы этого метода (метода `next()`) будут приводить к возобновлению выполнения функции- генератора.
Функции- генераторы объявляются следующей записью:

```
function* FuncName( … )
{ …
 yield … ;
 … }
```

как и говорилось выше, операторов `yield` может быть и несколько; этот оператор может быть непосредственно в теле функции- генератора (или же во вложенной в нее структуре управления ходом вычислений), но не может быть во вложенной в нее функции. Оператор `*` в этой записи позволяет произвольное употребление пробелов с ним. Оператор `yield` может использоваться и просто как оператор с его операндом, и как составляющая более сложного выражения — тогда это выражение и будет записью этого оператора.
Помимо оператора `yield` в генераторах может использоваться и оператор `return`; в случае генераторов этот оператор опционален (но, как обычно, если он употреблен, то обозначает конец тела функции). В случае генераторов оператор `return` возвращает не возвращаемое функцией значение, а значение `ResultsObjectName.value` (при использовании этого оператора значение `ResultsObjectName.done` равно `true`). С точки зрения метода `next()` объекта- итератора оператор `return` аналогичен оператору `yield`.
Функция- генератор может быть объявлена как в классическом определении функции, так и в функциональном выражении:

```
let VarName = function* ( … )
{ …
 yield … ;
 … }
```

Функции- генераторы не могут быть объявлены как стрелочные функции.

Функции- генераторы могут быть и методами объектов (если используется сокращенная запись объявления метода, то она имеет вид `*FuncName … ;`).

Генераторы. Цикл for – of

Этот цикл используется для работы с итерируемыми объектами (подробности о них — в следующем параграфе). Форма записи:

```
for (VarName of CollectionName) { … }
```

этот цикл применим только к итерируемым объектам; он вызывает из итерируемого объекта метод `[Symbol.iterator]()` и таким образом создает объект- итератор; затем, этот цикл в каждой итерации производит вызов метода `next()` объекта- итератора, и сохраняет значение его члена `value` в переменной цикла (`VarName`). Этот цикл завершается непосредственно по факту завершения всех итераций, поэтому переменной цикла не присваивается в конце специальное значение `undefinied`. Для этого цикла строка оператора `return` функции- генератора считается завершением всех итераций, поэтому переменной цикла не присваивается возвращаемое им значение.

Об итерируемых объектах

Итерируемые объекты — это наборы (но не со слабыми ссылками), отображения (но не со слабыми ссылками), массивы, обертки строк, и прочие объекты, удовлетворяющие требованиям к итерируемым объектам. Строго говоря, это одно требование, и оно заключается в следующем — итерируемым объектам поставляется объект- итератор по умолчанию.
Итак, итерируемым объектам поставляется итератор по умолчанию. Итератор по умолчанию доступен по языковой переменной- символу, доступ к итератору по умолчанию поизводится следующей записью:

```
let VarName = ObjectName[Symbol.iterator]();
// ObjectName – это итерируемый объект
```

в результате в переменную будет возвращена ссылка на итератор по умолчанию для данного объекта.
Пользовательский объект может быть объявлен итерируемым, если в его объявлении прописать итератор по умолчанию. Так как язык не предоставляет итератор по умолчанию для пользовательских объектов, то итератор для них должен быть определен пользователем.
Для этого прибегают к специфической записи:

```
let ObjectName = { … , *[Symbol.iterator]() { … }, … };
```

этой записью в объекте и прописывается итератор по умолчанию. Эта запись трактуется следующим образом: объявить `ObjectName[Symbol.iterator]` как функцию- генератор (в данной записи используется краткий синтаксис объявления методов, и это делает запись выглядящей нелогично). Впрочем, итератор по умолчанию может быть приписан объекту и после его создания:

```
ObjectName[Symbol.iterator] = *function() { … };
```

эта запись имеет вполне логичный вид.

Некоторые языковые генераторы

Массивам, наборам, и отображениям доступны следующие языковые генераторы.

`entries()`

это метод вышеназванных объектов; параметров не принимает; возвращает объект- итератор (при вызове из него метода `next()` будет возвращен соответствующий объект результатов (его член `value` содержит массив из двух элементов, которые трактуются как ключ и значение соответственно; ключем для массивов является индекс, для наборов — значение, для отображений — собственно ключ)); возвращает объект- итератор;
этот генератор используется для возврата итератора по умолчанию для отображений;

`values()`

это метод вышеназванных объектов; параметров не принимает; возвращает объект- итератор (при вызове из него метода `next()` будет возвращен соответствующий объект результатов (его член `value` содержит значение, которое трактуется как собственно значение элемента массива/ набора/ отображения)); возвращает объект- итератор;
этот генератор используется для возврата итератора по умолчанию для массивов и наборов;

`keys()`

это метод вышеназванных объектов; параметров не принимает; возвращает объект- итератор (при вызове из него метода `next()` будет возвращен соответствующий объект результатов (его член `value` содержит значение, которое трактуется как ключ элементов массива/ набора/ отображения; ключем для массивов является индекс, для наборов — значение, для отображений — собственно ключ)); возвращает объект- итератор;

Итерируемые объекты и деструктуризация

Строго говоря, деструктуризация (как и приведение к массиву (запись `...VarName`)) доступна не только объектам и массивам, но и всем итерируемым объектам.

При использовании этих операций интерпретатором задействуется итератор по умолчанию. С помощью него интерпретатор и определяет, что возвращать в результате этих операций. Для массивов и наборов — один итератор по умолчанию, для отображений — другой (так что при приведении отображения к массиву получится массив двухэлементных массивов). Обертки строк, как и любые итерируемые объекты, имеют итератор по умолчанию (так что при приведении строки к массиву получится массив односимвольных строк (при этом поддержка Unicode осуществляется полностью, и «широкие» символы трактуются корректно)).

Классы

Классы, как таковые, так и не были внедрены в язык `JavaScript`. В этом языке по прежнему нет классов — есть лишь собственно объекты и прототипы объектов. Классы объявляются следующей записью:

```
class ClassName { … }
```

методы класса объявляются в сокращенной форме (`FuncName( … ) { … }`). Конструктор класса объявляется в данном случае специфической записью (записью `constructor( … ) { … }`). Так же могут быть объявлены данные с методами доступа (обычной для этого записью `get VarName() { … }`, и `set VarName( … ) { … }` соответственно) — с точки зрения объявления класса эти объявления представляют собой объявления методов. Что касается объявления данных в обычной форме записи — их в принципе не должно быть в объявлении класса. Имена методов (обычных методов, и методов доступа — но не конструктора) могут быть объявлены как записью `FuncName`, так и записью `["FuncName"]` (причем в этой записи может использоваться как `"FuncName"`, так и выражение, возвращающее строку).

Составляющие объявления класса (в области `{ … }` в самом объявлении класса) разделяются просто пробелами (пробельными символами).

Составляющие объявления класса хранятся в общей для класса информации — то есть, в прототипе; при этом необходимо помнить, что члены объектов этого класса, инициализируемые конструктором, будут приписаны самим объектам класса (в результате их инициализации).

Необходимо помнить, что объявления класса — это всего лишь так называемый синтаксический сахар, а наследование в языке `JavaScript` по прежнему основано на прототипах. Использование этого синтаксического сахара просто приводит к тому, что создаются прототип и конструктор.

Также необходимо помнить о следующем: код сценария внутри объявления класса исполняется в строгом режиме; классы доступны только после их объявления в коде сценария; методы внутри объявления класса являются неперечислимыми.

Создание объектов производится записью:

```
let ObjectName = new ClassName();
```

Классы могут быть объявлены как записью объявления класса, так и просто записью классового выражения. Эта запись имеет следующий вид:
```
let VarName = class { … };
```

И тогда объекты создаются так:
```
let ObjectName = new VarName();
```

В классовом выражении имя может и присутствовать, и тогда запись примет вид:
```
let VarName = class ClassName { … };
```

но тогда имя класса будет доступно только внутри класса, и для создания объектов будет применяться соответствующая запись:
```
let ObjectName = new VarName();
```

Статические члены класса

Члены класса (то есть, то, что может быть прописано в объявлении класса — за исключением конструктора) могут быть объявлены как статические. Это делается записью ключевого слова `static` в начале их объявлений. Это запрещает обращаться к ним как к членам объектов, и вынуждает обращаться через имя класса (именно через имя класса, записью `ClassName.FunctionName()`).

Наследование классов

В язык `JavaScript` внедрена возможность имитировать наследование классов. Запись имитации наследования имеет следующий вид:
```
class DerivedClassName extends BasicClassName
{ … } // члены производного класса
```

по факту этой имитации становится доступен конструктор базового класса, он может быть вызван записью `super()`. Более того, конструктор производного класса и должен вызывать конструктор базового (если конструктор производного класса вообще определен). При определении конструктора производного класса необходимо помнить следующее:
вызов конструктора базового класса доступен только в конструкторе производного класса;
до вызова конструктора базового класса использование `this` в конструкторе производного класса не доступно;

При определении производного класса необходимо помнить следующее:
методы производного класса затеняют (одноименные) методы базового класса (для вызова метода базового класса следует использовать ссылку на базовый класс — запись `super.FuncName()` вызовет метод базового класса);

статические методы базового класса наследуются производным классом (и становятся статическими методами производного класса — они становятся доступны по записи `DerivedClassName.FuncName()`);
при определении производного класса вместо записи определенного ранее базового класса может использоваться запись какой- либо функции-конструктора (то есть, какой- либо функции, удовлетворяющей требованиям к конструкторам), тогда запись определения производного класса примет соответствующий вид (`DerivedClassName extends FuncName { … }`);

Массивы

Массивы дополнены следующими средствами.

Методы создания

`Array.of()`

используется для создания массивов; принимает параметры — ряд значений (трактуются как значения элементов создаваемого массива); создает массив (руководствуясь переданными параметрами); возвращает массив;

`Array.from()`

используется для создания массивов; принимает параметр — итерируемый/ подобный массиву объект (из его элементов и будет создан массив); создает массив (руководствуясь переданными параметрами); возвращает массив;
есть альтернативная форма записи с двумя параметрами — первый параметр трактуется как ссылка на итерируемый/ подобный массиву объект, второй параметр трактуется как пользовательская функция преобразования (эта функция осуществляет преобразование элементов, включаемых в массив; эта функция автоматически вызывается интерпретатором от включаемых в массив элементов — программисту лишь требуется ее прописать определенным образом (она должна принимать параметр — ему будут присваиваться значения включаемых в массив элементов, тело же полностью пользовательское));

Прочие методы

`find()`

используется для поиска целевого значения в массиве; принимает параметры — функцию (пользовательская функция), и объект (в его контексте и будет исполняться пользовательская функция, этот аргумент может быть опущен); выполняет поиск целевого значения в своем массиве, и возвращает первое

найденное значение, признанное целевым (за признание значения элемента целевым отвечает пользовательская функция; эта функция вызывается самим интерпретатором от каждого элемента массива, программисту следует лишь прописать ее тело определенным образом — она должна принимать параметр (значение элемента массива), исполнять прописанное на усмотрение программиста тело, и возвращать логическое значение (`true` – если значение элемента признано целевым, `false` – если нет)); возвращает значение (это найденное значение, если же целевое значение не найдено — то это значение `undefinied`);

`findIndex()`

используется для поиска индекса элемента (хранящего целевое значение) в массиве; принимает параметры — функцию (пользовательская функция), и объект (в его контексте и будет исполняться пользовательская функция, этот аргумент может быть опущен); выполняет поиск индекса элемента (хранящего целевое значение) в своем массиве, и возвращает первый найденный индекс элемента, хранящего значение, признанное целевым (за признание значения элемента целевым отвечает пользовательская функция; эта функция вызывается самим интерпретатором от каждого элемента массива, программисту следует лишь прописать ее тело определенным образом — она должна принимать параметр (значение элемента массива), исполнять прописанное на усмотрение программиста тело, и возвращать логическое значение (`true` – если значение элемента признано целевым, `false` – если нет)); возвращает значение (это индекс элемента, если же целевое значение не найдено — то это значение `-1`);

`fill()`

используется для заполнения массива; принимает параметры — значение (им и будет проводиться заполнение), значение (стартовой позиции, с нее включительно и будет проводиться заполнение; этот параметр может быть и опущен — тогда стартовой позицией считается 0я), и значение (финальной позиции, по нее не включительно и будет проводиться заполнение; этот параметр может быть и опущен — тогда заполнение будет проводиться до конца массива); заполняет свой массив (руководствуясь переданными параметрами); возвращает массив (это свой массив, претерпевший изменения);

`copyWithin()`

используется для копирования элементов в своем массиве; принимает параметры — значение (стартовая позиция, с нее включительно и будет проводиться затирание массива копируемыми элементами), значение (стартовая позиция, с нее включительно и будет проводиться копирование элементов), и значение (финальная позиция, по нее не включительно и будет проводиться копирование элементов; этот параметр может быть и опущен — тогда

копирование элементов будет проводиться до конца массива); производит копирование элементов в своем массиве (руководствуясь переданными параметрами, производит затирание своих элементов копируемыми (из самого себя) элементами); возвращает массив (это свой массив, претерпевший изменения);

Типизированные массивы

Типизированные массивы — это специфический вид массивов, поддерживаемый для использования их в математических вычислениях. Типизированные массивы могут хранить только числа.
Определенные виды типизированных массивов хранят определенные виды численных значений:

`Int8Array`	number (Integer, BYTE)
`Uint8Array`	number (Unsigned Integer, BYTE)
`Uint8ClampedArray`	number (Unsigned Integer, BYTE)
`Int16Array`	number (Integer, WORD)
`Uint16Array`	number (Unsigned Integer, WORD)
`Int32Array`	number (Integer, DWORD)
`Uint32Array`	number (Unsigned Integer, DWORD)
`Float32Array`	number (Floating- point, Single- precesion)
`Float64Array`	number (Floating- point, Double- precesion)

они хранят эти типы значений непосредственно в таком виде (а не в том виде, в котором числа представляются в интерпретаторе (то есть, Double- precesion), исключение составляют типизированные массивы `Float64Array` – в них числа хранятся в том же виде, в каком и представляются в интерпретаторе).
Применение типизированных массивов дает следующую выгоду: числа хранятся в соответствующем типу массива представлении (и не требуют дальнейшего приведения к целевому представлению; кроме того, при хранении чисел (меньше по емкости чем Double- precesion) достигается экономия памяти).
При применении типизированных массивов совместно с ними требуется применять специфический тип массива — буфер массива.
Буфер массива создается следующей записью:

```
let VarName = ArrayBuffer( … );
```

в этой записи конструктору нужно указать аргумент — число (количество байт, выделяемых для буфера). Работа с буфером массива ведется посредством так называемых представлений данных (собственно объектов — представлений данных, или же объектов — конкретных типизированных массивов).
Объекты — представления данных создаются следующей записью:

```
ViewVarName = new DataView(VarName);
// VarName – ArrayBuffer
```

представления данных служат интерфейсом к буферам массивов. При работе с представлениями можно и не прибегать (строго говоря, и не прибегают) к использованию конкретных типов типизированных массивов — представления и так служат интерфейсами к буферам массивов, и решают свою задачу достаточно полно. При использовании конкретных типов типизированных массивов, они просто используются вместо представлений данных (вместо объекта — представления данных создается объект — конкретный типизированный массив).
Представление может быть создано как для всего буфера, так и для его части, для этого используется запись:

```
ViewVarName = new DataView(VarName, Value0, Value1);
```

здесь `Value0` – смещение с которого создается представление, `Value1` – длина создаваемого представления в байтах (этот аргумент можно опустить, и тогда представление будет создано до конца буфера массива).
Доступ к данным (как и прочие операции) осуществляется вызовом соответствующих методов объекта- отображения.
Следует рассмотреть члены объектов `ArrayBuffer`, `DataView`, и объектов — конкретных типизированных массивов.

Члены объектов ArrayBuffer

Конструктор объектов ArrayBuffer

`ArrayBuffer()`

конструктор объектов `ArrayBuffer`; принимает параметр — значение (число выделяемых под буфер байт); создает объект `ArrayBuffer` (руководствуясь переданными параметрами); возвращает ничтоже;

Методы объектов ArrayBuffer

`slice()`

этот метод аналогичен одноименному методу массивов (объектов `Array`); за тем исключением, что возвращает объект — буфер массива;

Переменные объектов ArrayBuffer

`length`

аналогично одноименному члену массивов (объектов `Array`); однако, это константа — буфер массива не может варьировать размер;

Члены объектов DataView

Конструктор объектов DataView

DataView()

конструктор объектов `DataView`; принимает параметры — либо объект (это объект `ArrayBuffer`, представление которого создается), либо объект (это объект `ArrayBuffer`, представление которого создается), значение (число, трактуется как смещение в указанном первым аргументом буфере массива, с которого и создается представление) и значение (число, трактуется как длина создаваемого представления в байтах; этот параметр может быть опущен); создает объект `DataView` (руководствуясь переданными параметрами); возвращает ничтоже;

Методы объектов DataView

Методы для считывания значений

getInt8()

используется для возврата значения элемента из своего типизированного массива; принимает параметр — значение (трактуется как значение смещения, по которому и запрашивается элемент (смещение в байтах)); возвращает значение элемента своего типизированного массива (руководствуясь переданными параметрами); возвращает значение (элемента);
методы представлений могут быть применены независимо от того, какое представление чисел использовалось для инициализации этого представления;

getUint8()

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

getInt16()

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

getUint16()

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`getInt32()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`getUint32()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`getFloat32()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`getFloat64()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

Методы для записи значений

`setInt8()`

используется для записи значения элемента в свой типизированный массив; принимает параметры — значение (трактуется как значение смещения, по которому и записывается значение в элемент (смещение в байтах)), и значение (трактуется как значение, которое и будет записано); запсывает значение в элемент своего типизированного массива (руководствуясь переданными параметрами); возвращает ничтоже;
методы представлений могут быть применены независимо от того, какое представление чисел использовалось для инициализации этого представления;

`setUint8()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`setInt16()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`setUint16()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`setInt32()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`setUint32()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`setFloat32()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

`setFloat64()`

аналогичен вышеописанному методу; за тем исключением, что используется для доступа к данным в соответствующем представлении;

Переменные объектов DataView

`buffer`

ссылка на используемый буфер массива;

`bufferOffset`

смещение, с которого создавалось представление (если соответствующий аргумент при создании представления не задействован, то равно `0`);

`byteLenght`

длина педставления в байтах (если соответствующий аргумент при создании представления не задействован, то равно длине буфера массива);

Члены конкретных видов типизированных массивов

Конкретные виды типизированных массивов — `Int8Array`, `Uint8Array`, `Uint8ClampedArray`, `Int16Array`, `Uint16Array`, `Int32Array`, `Uint32Array`, `Float32Array`, `Float64Array` – по сути представляют собой специфические отображения, работающие с конкретным представлением численных значений. То есть, конкретные типизированные массивы допускают операции только со своим конкретным представлением чисел. Для операций с хранящимися в этих массивах данными доступна обычная для этого запись индексации (`VarName[ … ]`).

Конструктор конкретных типизированных массивов

Как обычно, конструктор имеет имя типа создаваемых объектов. С точки зрения принимаемых параметров, конструктор этих объектов идентичен `DataView()`. Разве что, имеется следующая специфика — конструктор так же может принять специфический аргумент (когда используется с единственным аргументом), и он может быть следующим: число (трактуется как размер этого массива в байтах), ссылочный тип (либо конкретный типизированный массив — тогда будет создана его копия, массив — тогда будет создана его копия (но это будет конкретный типизированный массив), либо объект подобный массиву — тогда будет создана его копия (но это будет конкретный типизированный массив), либо итерируемый объект — и тогда будет создан конкретный типизированный массив из элементов, полученных итерированием (это действие остается на откуп интерпретатору, и поэтому не требуется от программиста)). Однако, возможность задействования тех или иных параметров должна уточняться для каждого конкретного интерпретатора.

Методы конкретных типизированных массивов

Конкретные типизированные массивы располагают несколько иным набором методов, нежели обычные массивы. У них имеются следующие методы.

Методы обычных массивов

Эти методы аналогичны одноименным методам массивов (но необходимо помнить — при работе с конкретными типизированными массивами можно работать только с их конкретными типами данных — при использовании недопустимого значения оно заменяется значением `0`).
Типизированные массивы имеют следующие методы обычных массивов.

```
indexOf()
lastIndexOf()
reverse()
sort()
```

```
reduce()
reduceRight()
filter()
some()
forEach()
map()
join()
slice()

of()
from()

find()
findIndex()
fill()
copyWithin()
```

Методы — языковые генераторы

Типизированные массивы также имеют методы — языковые генераторы.

```
entries()
value()
keys()
```

Специфические методы

Это следующие методы.

```
set()
```

используется для копирования другого массива; принимает параметры — обычный или типизированный массив (это указанный массив), и значение (это смещение в своем массиве, с которого и происходит копирование; этот аргумент можно и опустить); копирует элементы указанного массива в свой типизированный массив (руководствуясь переданными параметрами); возвращает ничтоже;

```
subarray()
```

используется для извлечения подмассива; принимает параметры — значение (это стартовая позиция, с нее включительно; этот аргумент можно опустить), и

значение (это конечная позиция, по нее не включительно происходит работа метода; этот аргумент можно опустить); извлекает элементы из своего типизированного массива (руководствуясь переданными параметрами); возвращает типизированный массив (это новый типизированный массив);

Переменные конкретных типизированных массивов

Это те же переменные, что у объектов `DataView()`. Разве что, имеет место своя специфика. Переменная `length` не доступна для реинициализации — предполагается, что размер такого массива не должно изменять. Так же имеется своя специфическая переменная — `BYTES_PER_ELEMENT`.

`BYTES_PER_ELEMENT`

как и следует из названия, эта переменная содержит значение — емкость элемента своего типа массива в байтах; константа;
доступность этой переменной следует уточнять для конкретного интерпретатора;

Объекты Promise

Описание этих объектов требует некоторых вводных сведений.
Эти объекты рассматриваются в связи с так называемым циклом событий.
Цикл событий — это информационный процесс в двигателе `JavaScript`. Цикл событий в этом языке однопоточный — код на этом языке исполняется в один поток. Сам этот информационный процесс управляет очередью заданий. Несмотря на отсутствие многопоточности в языке `JavaScript` об очереди заданий все же приходится говорить — это потому, что исполнение разных составляющих кода на `JavaScript` может мультиплексироваться во времени Это происходит, например, при наличии событий веб- документа. Наступающие события пополняют очередь заданий (очередь событий — на то и очередь, что пополняется в порядке очереди).
Использование объектов `Promise` позволяет программисту влиять на очередность исполняемых заданий. Заданиями в данном случае будут составляющие кода сценария, оформленные в функции, используемые в связи с объектом `Promise` (в рамках предоставляемых в связи с ним механизмов).

Состояния объектов Promise

Объекты `Promise` могут принимать следующие состояния:
состояние ожидания/ неустановившийся объект `Promise`/ с т.з. интерпретатора это состояние «`pending`» — задание еще не завершено;

выполнено/ установившийся объект Promise/ с т.з. интерпретатора это состояние «fulfilled» — задание выполнено успешно;
отклонено/ установившийся объект Promise/ с т.з. интерпретатора это состояние «rejected» — задание завершено интерпретатором (при этом выбрасывается ошибка), а не выполнено успешно;

при достижении объектом Promise определенного состояния программист может запросить выполнение определенного (прописанного им) кода. Для выполнения целевого кода служит метод then() объекта Promise. Этот метод используется для выполнения целевого кода (по достижению объектом Promise целевого состояния). Принимает параметры — функцию (предназначена для выполнения в состоянии «fulfilled»), и функцию (предназначена для выполнения в состоянии «rejected»). Обе эти функции передаются в форме безымянного литерала. Вызов метода then() приводит к (условному) исполнению одной из принимаемых им функций. Обе функции, принимаемые методом then(), опциональны (могут быть опущены). Сам метод then() возвращает объект Promise. Так как он возвращает объект, это позволяет вызывать из него методы (возможен в т. ч. пресловутый вызов по цепочке). Когда по цепочке вызывается метод then(), это называется «соединением» - соединение происходит по цепочке. При соединении необходимо помнить, какой результат и на каком этапе соединения будет получен — без этого не возможно корректное программирование с использованием соединений.
Любой объект, в котором прописан метод then(), принято называть «thenable-object». Все объекты Promise являются «thenable», но не все «thenable» являются объектами Promise.
Для исполнения целевого кода при достижении объектом Promise состояния «rejected» может быть применен и специализированный метод — метод catch(), в отличии от метода then() он используется только в состоянии «rejected» объекта Promise. Этот метод принимает параметр — функцию (предназначена для выполнения в состоянии «rejected»), тоже в форме безымянного литерала.
Оба вышеописанных метода — then() и catch(), являются заданиями (с точки зрения очереди заданий). Они пополняют очередь заданий условно (при вызове, и только по достижению объектом Promise соответствующего состояния).

Создание объектов Promise

Объекты Promise создаются их конструктором:
```
let VarName = Promise( … );
```

конструктор принимает функцию - «исполнитель», инициализирующую объект; кроме того, что она инициализирует объект, ей передаются две функции — resolve() и reject(). Эти две функции автоматически вызываются по

достижении объектом `Promise` соответствующих состояний — выполнено/ отклонено; даже если объект достигает состояния отклонено (и тогда выбрасывается ошибка) программисту не требуется перехватывать ошибки, а соответствующая функция будет вызвана автоматически. Остальная же часть функции исполнителя предназначена для инициализации объекта в его исходном состоянии, и поэтому исполняется безусловно — сразу при вызове конструктора объекта `Promise`. Тело функции «исполнителя» — пользовательское, однако оно должно содержать две функции — `resolve()` и `reject()`, тело самих этих функций — пользовательское. Во всем же остальном тело функции-«исполнителя» пользовательское; и возвращаемое им значение — тоже пользовательское (однако, возвращаемое «исполнителем» значение просто отбрасывается). Функция- «исполнитель» передается конструктору в виде безымянного литерала.

Члены объектов Promise

Конструктор объектов Promise

`Promise()`

конструктор объектов `Promise`; принимает параметр - функцию («исполнитель»); создает объект `Promise`; возвращает ничтоже;
функция- «исполнитель» должна принимать два параметра — с именами `resolve` и `reject`; тело функции «исполнителя» — пользовательское, однако оно должно содержать две функции — `resolve()` и `reject()`, тело самих этих функций — пользовательское;

Методы объектов Promise

`then()`

используется для выполнения целевого кода (по достижению объектом `Promise` целевого состояния); принимает параметры — функцию (предназначена для выполнения в состоянии «`fulfilled`»), и функцию (предназначена для выполнения в состоянии «`rejected`»); вызов этого метода приводит к (условному) исполнению одной из принимаемых им функций; возвращает объект `Promise` (это свой объект метода);
обе функции, принимаемые этим методом, опциональны (могут быть опущены);
обе эти функции задаются безымянным литералом;
их тела — пользовательские (но они должны быть прописаны принимающими один параметр), возвращаемое ими значение будет автоматически обернуто в объект `Promise`;

`catch()`

используется для выполнения целевого кода (по достижению объектом `Promise` целевого состояния); принимает параметр — функцию (предназначена для выполнения в состоянии «`rejected`»); вызов этого метода приводит к (условному) исполнению принимаемой функции; возвращает объект `Promise` (это свой объект метода);
эта функция задается безымянным литералом;
ее тело — пользовательское (но она должна быть прописана принимающей один параметр), возвращаемое ей значение будет автоматически обернуто в объект `Promise`;

`resolve()`

используется для возврата объекта `Promise` в состоянии «`fulfilled`»; принимает параметр — значение; возвращает объект `Promise` в состоянии «`fulfilled`» (вызов этого метода приводит к исполнению соответствующей функции в теле функции- «исполнителя»); возвращает объект `Promise`;
этому методу можно передать и объект `Promise` – тогда будет возвращен он сам, без изменений;
этому методу можно передать и какой- либо thenable- объект — тогда будет возвращен объект `Promise` в соответствующем состоянии, получаемый исполнением метода `then()` переданного объекта;

`reject()`

используется для возврата объекта `Promise` в состоянии «`rejected`»; принимает параметр — значение; возвращает объект `Promise` в состоянии «`rejected`»; возвращает объект `Promise`;
этому методу можно передать и объект `Promise` – тогда будет возвращен он сам, без изменений;
этому методу можно передать и какой- либо thenable- объект — тогда будет возвращен объект `Promise` в соответствующем состоянии, получаемый исполнением метода `then()` переданного объекта;

`all()`

используется для работы с рядом объектов `Promise`; принимает параметр — ссылочного типа (обязательно итерируемый объект), этот параметр инкапсуллирует ряд объектов `Promise`, с которыми работает этот метод; возвращает объект `Promise`, который установлен по факту установки всех указанных объектов `Promise` (он возвращается установленным по факту установки указанных объектов, но он будет «`fulfilled`» только когда все

указанные объекты будут «fulfilled»; если же факт установки указанных объектов не произошел, то объект возвращается «rejected»); возвращает объект Promise;

`race()`

используется для работы с рядом объектов Promise; принимает параметр — ссылочного типа (обязательно итерируемый объект), этот параметр инкапсуллирует ряд объектов Promise, с которыми работает этот метод; возвращает объект Promise, который установлен по факту установки первого установившегося из указанных объектов Promise (он возвращается установленным по факту установки одного из указанных объектов, и его конкретное состояние соответствует состоянию этого объекта); возвращает объект Promise;

Переменные объектов Promise

Их нет.

Резюме

Вызов конструктора объектов Promise приводит к добавлению функции-«исполнителя» в очередь заданий. Поэтому и возможно получение еще не установившегося объекта Promise.
Метод then() объектов Promise также приводит к добавлению (этого метода) в очередь заданий. Это же касается и метода catch(). Использование этих методов эквивалентно использованию конструкций перехвата ошибок. Если же эти методы не задействуются, то ошибки просто отбрасываются.
Добавление этих функций в очередь заданий (вместо немедленного их исполнения) необходимо для реализации концепции асинхронных вычислений (имеется ввиду асинхронность с потоком операторов сенария).
Функция- «исполнитель» конструктора объектов Promise и метод then() объектов Promise принимают параметры — функции, сходные по назначению. Необходимо помнить, что это — разные функции, которые вызываются в разных случаях.
Использование объектов Promise происходит во многом при помощи их методов. Соответствующие методы приводят к установке объекта в соответствующее состояние. Это, в свою очередь, приводит к возврату соответствующих значений — по сути, эти значения и представляют результат работы с объектом Promise.
Объекты Promise могут быть реализованы специфически на разных интерпретаторах; о реализации объектов Promise следует уточнять для конкретного интерпретатора.

Прокси- объекты

Прокси- объекты используются для работы с ними вместо целевого объекта (при этом, прокси- объект представляется функционально идентичным целевому). Таким образом, они используются для виртуализации целевого объекта.
Прокси- объекты создаются соответствующим конструктором:
```
let VarName = new Proxy( … );
```

этот конструктор принимает параметры — объект (это целевой объект прокси-объекта), и объект (это объект- обработчик (содержит ряд функций- ловушек, этот ряд может быть и пустым — тогда этот объект- обработчик прописывается пустым), задается литералом); создает (руководствуясь переданными параметрами) прокси- объект; возвращает объект `Proxy`.
После того, как прокси- объект создан, он может использоваься вместо своего целевого объекта. После того, как прокси- объект создан, все операции над прокси- объектом приводят к операциям над целевым объектом прокси- объекта. Кроме того, все операции над целевым объектом отображаются и на прокси-объекте. Сам же прокси- объект не хранит значений в своих членах — вместо этого он хранит в них ссылки на одноименные члены своего целевого объекта.
После создания прокси- объекта определенные операции над целевым объектом могут быть перехвачены интерпретатором и обработаны по логике, прописанной программистом. Поэтому о создании прокси- объекта принято говорить, что «прокси- объект подключен к целевому объекту».
Необходимо помнить, что только определенные операции могут быть перехвачены; остальные операции не могут быть перехвачены, и исполняются только по логике, предусмотренной самим языком. Могут быть перехвачены только те операции, для которых языком предусмотрены функции- ловушки. Функции- ловушки используются в конструкторе прокси- объектов (они прописываются во втором его аргументе). Каждая функция- ловушка имеет определенное имя, и соответствует определенной операции. Эта функция прописывается под определенным языком именем, с определенными для нее языком параметрами, и с пользовательским телом — это тело и будет исполняться при перехвате соответствующей операции. Если же требуется реализовать эту операцию так, как это предусмотрено в самом языке, то соответствующую ей ловушку можно просто не использовать. Для этого можно также использовать ловушку, и в ней вызвать метод, реализующий поведение соответствующей операции по умолчанию (хотя, это избыточно). Этот метод, реализующий поведение по умолчанию, хранится в объекте `Reflect`. Объект `Reflect` – это служебный объект, предназначенный для хранения этих методов. Этот объект «статический» (и все его методы тоже). Его методы одноименны соответствующим ловушкам, и принимают такие же параметры.
Возможности, предоставляемые объектом `Reflect` принято называть `Reflection API`.
Языком `JavaScript` предусмотрены следующие функции- ловушки.

`get()`

используется для перехвата операции (обращение к члену объекта по чтению), поведение операции по умолчанию реализовано в методе `Reflect.get()`; соответствующий метод принимает параметры — объект (целевой для прокси-объекта), строку или переменную- символ (трактуется как ключ для доступа к целевому члену объекта), и объект (трактуется как целевой объект прокси-объекта); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвращает значение (целевого члена объекта);

`set()`

используется для перехвата операции (обращение к члену объекта по записи), поведение операции по умолчанию реализовано в методе `Reflect.set()`; соответствующий метод принимает параметры — объект (целевой для прокси-объекта), строку или переменную- символ (трактуется как ключ целевого члена объекта), значение (трактуется как значение целевого члена объекта), и объект (трактуется как целевой объект прокси- объекта); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвращает логическое значение (`true` – оперция выполнена, `false` – операция не выполнена);
сигнатура ловушки должна полностью соответствовать сигнатуре метода;

`has()`

используется для перехвата операции (проверка наличия члена объекта оператором `in`), поведение операции по умолч. реализ- но в `Reflect.has()`; принимает параметры — объект (целевой для прокси- объекта), и строку или переменную- символ (трактуется как ключ целевого члена объекта); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвращает логическое значение (возвращаемое оператором `in`);
сигнатура ловушки должна полностью соответствовать сигнатуре метода;

`deleteProperty`

используется для перехвата операции (удаление члена объекта оператором `delete`), поведение операции по умолчанию реализовано в методе `Reflect.deleteProperty()`; принимает параметры — объект (целевой для прокси- объекта), и строку или переменную- символ (трактуется как ключ целевого члена объекта); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвращает логическое значение (возвращаемое оператором `delete`); сигнатура ловушки должна полностью соответствовать;

`getPrototypeOf()`

используется для перехвата операции (вызов `Object.getPrototypeOf()`), поведение операции по умолч. реализовано в `Reflect.getPrototypeOf()`; принимает параметр — объект (целевой для прокси- объекта); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвращает (см. описание);
сигнатура ловушки должна полностью соответствовать сигнатуре метода;

`setPrototypeOf()`

используется для перехвата операции (вызов `Object.setPrototypeOf()`), поведение операции по умолч. реализовано в `Reflect.setPrototypeOf()`; принимает параметры — объект (целевой для прокси- объекта), и объект (который будет использоваться как прототип); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвр. (см. описание);
этот метод различает прокси- объект и его целевой объект;

`isExtensible()`

используется для перехвата операции (вызов `Object.isExtensible()`), поведение операции по умолчанию реализовано в `Reflect.isExtensible()`; принимает параметр — объект (целевой для прокси- объекта); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвращает (см. описание);

`preventExtensions()`

используется для перехвата операции (операции — вызова метода `Object.preventExtensions()`), поведение операции по умолчанию реализовано в `Reflect.PreventExtensions()`; принимает параметр — объект (целевой для прокси- объекта); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвращает (см. описание);

`getOwnPropertyDescriptor()`

используется для перехвата операции (вызов метода `Object.getOwnPropertyDescriptor()`), поведение операции по умолчанию реализовано в методе `Reflect.getOwnPropertyDescriptor()`; принимает

параметры — объект (целевой для прокси- объекта), строку или переменную-символ (ключ); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвращает (см. описание);

`defineProperty()`

используется для перехвата операции (вызов метода `Object.defineProperty()`), поведение операции по умолчанию реализовано в методе `Reflect.defineProperty()`; принимает параметры — объект (целевой дляпрокси- объекта), строку или переменную- символ (ключ), объект (объект- дескриптор); соответствующий метод производит данную операцию в реализации по умолчанию (руководствуясь переданными параметрами); соответствующий метод возвращает (см. описание);

`ownKeys()`

используется для перехвата операции (возврата ключей собственных членов, к исполнению этой внутренней операции приводит ряд методов: `Object.keys()`, `Object. GetOwnPropertyNames()`, `Object.getOwnPropertySymbols()`, `Object.Assign()`), поведение операции по умолчанию реализовано в `Reflect.ownKeys()`; принимает параметр — объект (это целевой объект прокси- объекта); соответствующий метод реализует операцию (возвращает ключи собственных членов объекта в виде массива; это внутренняя операция, исполняемая в интересах вышеупомянутого ряда методов); возвращает массив; сигнатура ловушки должна полностью соответствовать сигнатуре метода;

`apply()`

используется для вызова функции, поведение операции по умолчанию реализовано в методе `Reflect.apply()`;
подробнее — см. справочник;

`construct()`

используется для вызова функции- конструктора, поведение операции по умолчанию реализовано в методе `Reflect.construct()`;
подробнее — см. справочник;