Точно так же, если потребуется сравнить два отдельных объекта или массива, необходимо будет сравнить значения их свойств или элементов. Ниже приводится определение функции, сравнивающей два массива:

```
function equalArrays(a,b) {
    if (a.length != b.length) return false;  // Массивы разной длины не равны
        for(var i = 0; i < a.length; i++)    // Цикл по всем элементам
            if (a[i] !== b[i]) return false; // Если хоть один элемент
                                             // отличается, массивы не равны
        return true;                         // Иначе они равны
}
```

3.8. Преобразование типов

JavaScript может гибко преобразовывать один тип в другой. Мы уже могли убедиться в этом на примере логических значений: везде, где интерпретатор JavaScript ожидает получить логическое значение, можно указать значение любого типа и JavaScript автоматически выполнит необходимое преобразование. Одни значения («истинные» значения) преобразуются в значение `true`, а другие («ложные») – в `false`. То же относится и к другим типам: если интерпретатор ожидает получить строку, он автоматически преобразует любое другое значение в строку. Если интерпретатор ожидает получить число, он попробует преобразовать имеющееся значение в число (в случае невозможности такого преобразования будет получено значение `NaN`). Например:

```
10 + " objects"  // => "10 objects". Число 10 преобразуется в строку
"7" * "4"        // => 28: обе строки преобразуются в числа
var n = 1 - "x"; // => NaN: строка "x" не может быть преобразована в число
n + " objects"   // => "NaN objects": NaN преобразуется в строку "NaN"
```

В табл. 3.2 описывается, как в JavaScript выполняется преобразование значений из одного типа в другой. Жирным шрифтом в таблице выделены значения, соответствующие преобразованиям, которые могут преподносить сюрпризы. Пустые ячейки соответствуют ситуациям, когда преобразование не требуется и не выполняется.

Преобразования одного простого типа в другой, показанные в табл. 3.2, выполняются относительно просто. Преобразование в логический тип уже обсуждалось в разделе 3.3. Преобразование всех простых типов в строку четко определено. Преобразование в число выполняется немного сложнее. Строки, которые могут быть преобразованы в числа, преобразуются в числа. В строке допускается наличие пробельных символов в начале и в конце, но присутствие других непробельных символов, которые не могут быть частью числа, при преобразовании строки в число приводят к возврату значения `NaN`. Некоторые особенности преобразования значений в числа могут показаться странными: значение `true` преобразуется в число 1, а значение `false` и пустая строка `""` преобразуются в `0`.

Преобразование простых типов в объекты также выполняется достаточно просто: значения простых типов преобразуются в соответствующие объекты-обертки (раздел 3.6), как если бы вызывался конструктор `String()`, `Number()` или `Boolean()`.

Исключение составляют значения `null` и `undefined`: любая попытка использовать их в контексте, где требуется объект, вместо преобразования будет приводить к возбуждению исключения `TypeError`.

Преобразование объектов в простые типы выполняется значительно сложнее и является темой обсуждения раздела 3.8.3.

Таблица 3.2. Преобразование типов в JavaScript

Значение	Преобразование в:			
	Строку	**Число**	**Логическое значение**	**Объект**
`undefined`	`"undefined"`	`NaN`	`false`	*возбуждается ошибка TypeError*
`null`	`"null"`	`0`	`false`	*возбуждается ошибка TypeError*
`true`	`"true"`	`1`		`new Boolean(true)`
`false`	`"false"`	`0`		`new Boolean(false)`
`""` (пустая строка)		`0`	`false`	`new String("")`
`"1.2"` (непустая строка, число)		`1.2`	`true`	`new String("1.2")`
`"one"` (не пустая строка, не число)		`NaN`	`true`	`new String("one")`
`0`	`"0"`		`false`	`new Number(0)`
`-0`	`"0"`		`false`	`new Number(-0)`
`NaN`	`"NaN"`		`false`	`new Number(NaN)`
`Infinity`	`"Infinity"`		`true`	`new Number(Infinity)`
`-Infinity`	`"-Infinity"`		`true`	`new Number(-Infinity)`
`1` (конечное, ненулевое)	`"1"`		`true`	`new Number(1)`
`{}` (любой объект)	*см. разд. 3.8.3*	*см. раздел 3.8.3*	`true`	
`[]` (пустой массив)	`""`	`0`	`true`	
`[9]` (1 числовой элемент)	`"9"`	`9`	`true`	
`['a']` (любой другой массив)	*используется метод* `join()`	`NaN`	`true`	
`function(){}` (любая функция)	*см. разд. 3.8.3*	`NaN`	`true`	

3.8.1. Преобразования и равенство

Благодаря гибкости преобразований типов в JavaScript оператор равенства `==` также гибко определяет равенство значений. Например, все следующие сравнения возвращают `true`:

```
null == undefined // Эти два значения считаются равными.
"0" == 0          // Перед сравнением строка преобразуется в число.
0 == false        // Перед сравнением логич. значение преобразуется в число.
"0" == false      // Перед сравнением оба операнда преобразуются в числа.
```

В разделе 4.9.1 четко описывается, какие преобразования выполняет оператор `==`, чтобы определить, являются ли два значения равными, и в этом же разделе описывается оператор идентичности `===`, который не выполняет никаких преобразований перед сравнением.

Имейте в виду, что возможность преобразования одного значения в другое не означает равенства этих двух значений. Если, например, в логическом контексте используется значение undefined, оно будет преобразовано в значение false. Но это не означает, что undefined == false. Операторы и инструкции JavaScript ожидают получить значения определенных типов и выполняют преобразования в эти типы. Инструкция if преобразует значение undefined в false, но оператор == никогда не пытается преобразовать свои операнды в логические значения.

3.8.2. Явные преобразования

Несмотря на то что многие преобразования типов JavaScript выполняет автоматически, иногда может оказаться необходимым выполнить преобразование явно или окажется предпочтительным выполнить явное преобразование, чтобы обеспечить ясность программного кода.

Простейший способ выполнить преобразование типа явно заключается в использовании функций Boolean(), Number(), String() и Object(). Мы уже видели, как эти функции используются в роли конструкторов объектов-оберток (раздел 3.6). При вызове без оператора new они действуют как функции преобразования и выполняют преобразования, перечисленные в табл. 3.2:

```
Number("3")    // => 3
String(false)  // => "false" или можно использовать false.toString()
Boolean([])    // => true
Object(3)      // => new Number(3)
```

Обратите внимание, что все значения, кроме null или undefined, имеют метод toString(), результатом которого обычно является то же значение, которое возвращается функцией String(). Кроме того, обратите внимание, что в табл. 3.2 отмечается, что при попытке преобразовать значение null или undefined в объект возбуждается ошибка TypeError. Функция Object() в этом случае не возбуждает исключение, вместо этого она просто возвращает новый пустой объект.

Определенные операторы в языке JavaScript неявно выполняют преобразования и иногда могут использоваться для преобразования типов. Если один из операндов оператора + является строкой, то другой операнд также преобразуется в строку. Унарный оператор + преобразует свой операнд в число. А унарный оператор ! преобразует операнд в логическое значение и инвертирует его. Все это стало причиной появления следующих своеобразных способов преобразования типов, которые можно встретить на практике:

```
x + ""  // То же, что и String(x)
+x      // То же, что и Number(x). Можно также встретить x-0
!!x     // То же, что и Boolean(x). Обратите внимание на два знака !
```

Форматирование и парсинг чисел являются наиболее типичными задачами, решаемыми компьютерными программами, и потому в JavaScript имеются специализированные функции и методы, обеспечивающие более полный контроль над преобразованиями чисел в строки и строк в числа.

Метод toString() класса Number принимает необязательный аргумент, определяющий основание системы счисления для преобразования. Если этот аргумент не определен, преобразование выполняется в десятичной системе счисления. Но вы