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

можете производить преобразование в любой системе счисления (с основанием от 2 до 36). Например:

```
var n = 17;
binary_string = n.toString(2);       // Вернет "10001"
octal_string = "0" + n.toString(8); // Вернет "021"
hex_string = "0x" + n.toString(16); // Вернет "0x11"
```

При выполнении финансовых или научных расчетов может потребоваться обеспечить преобразование чисел в строки с точностью до определенного числа десятичных знаков или до определенного количества значащих разрядов или получать представление чисел в экспоненциальной форме. Для подобных преобразований чисел в строки класс `Number` определяет три метода. Метод `toFixed()` преобразует число в строку, позволяя указывать количество десятичных цифр после запятой. Он никогда не возвращает строки с экспоненциальным представлением чисел. Метод `toExponential()` преобразует число в строку в экспоненциальном представлении, когда перед запятой находится единственный знак, а после запятой следует указанное количество цифр (т. е. количество значащих цифр в строке получается на одну больше, чем было указано при вызове метода). Метод `toPrecision()` преобразует число в строку, учитывая количество заданных значащих разрядов. Если заданное количество значащих разрядов оказывается недостаточным для отображения всей целой части числа, преобразование выполняется в экспоненциальной форме. Обратите внимание, что все три метода округляют последние цифры или добавляют нули, если это необходимо. Взгляните на следующие примеры:

```
var n = 123456.789;
n.toFixed(0);          // "123457"
n.toFixed(2);          // "123456.79"
n.toFixed(5);          // "123456.78900"
n.toExponential(1);    // "1.2e+5"
n.toExponential(3);    // "1.235e+5"
n.toPrecision(4);      // "1.235e+5"
n.toPrecision(7);      // "123456.8"
n.toPrecision(10);     // "123456.7890"
```

Если передать строку функции преобразования `Number()`, она попытается разобрать эту строку как литерал целого или вещественного числа. Эта функция работает только с десятичными целыми числами и не допускает наличие в строке завершающих символов, не являющихся частью литерала числа. Функции `parseInt()` и `parseFloat()` (это глобальные функции, а не методы какого-либо класса) являются более гибкими. Функция `parseInt()` анализирует только целые числа, тогда как функция `parseFloat()` позволяет анализировать строки, представляющие и целые, и вещественные числа. Если строка начинается с последовательности «0x» или «0X», функция `parseInt()` интерпретирует ее как представление шестнадцатеричного числа.[1] Обе функции, `parseInt()` и `parseFloat()`, пропускают начальные

[1] Согласно стандарту ECMAScript 3 функция `parseInt()` может выполнять преобразование строки, начинающейся с символа «0» (но не «0x» или «0X»), в восьмеричное или десятичное число. Поскольку поведение функции четко не определено, следует избегать использования функции `parseInt()` для интерпретации строк, начинающихся с «0», или явно указывать основание системы счисления! В ECMAScript 5 функция `parseInt()` будет интерпретировать строки как восьмеричные числа, только если ей во втором аргументе явно указать основание 8 системы счисления.