*Chapter 1*

***Values, Types, and Operators***

Any piece of **discrete information** can be reduced to a sequence of zeros and ones thus represented in **bits.**

Bits that make up the number 13:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

So that’s the binary number 00001101, or 8 + 4 + 1, which equals 13.

**Values**

There are six basic types of values in JavaScript:

* Numbers
* Strings
* Booleans
* Objects
* Functions
* Undefined values

**Numbers**

JavaScript numbers are always stored as double precision floating point numbers. This format stores numbers in 64 bits.

Given 64 binary digits, you can represent 264 different numbers, which is about 18 quintillion (18,000,000,000,000,000,000).

Just as π (pi) cannot be precisely expressed by a finite number of decimal digits, many numbers lose some precision when only 64 bits are available to store them.

**Arithmetic**

(100 + 4) \* 11

The + and \* symbols are called ***operators.***The % symbol is used to represent the *remainder* operation. X % Y is the remainder of dividing X by Y. Remainder’s precedence is the same as that of multiplication and division. % referred to as *modulo*, though technically *remainder* is more accurate*.*

**Strings**

Strings are used to represent text. They are written by enclosing their content in quotes.

“Patch my boat with chewing gum”

‘Monkeys wave goodbye’

*Newlines* (the characters you get when you press Enter) also can’t be put in quotes.

\n newline

\t tab character

“This is the first line\nAnd this is the second”

Becomes:

This is the first line

And this is the second

**Unary Operators**

Not all operators are symbols. Some are written as words. One examples if the *typeof* operator, which produces a string value naming the type of the value you give it.

console.log(typeof 4.5)

// number

console.log(typeof “x”)

// string

Operators that use **two values** are called ***binary* operators,** while those that take **one** are called ***unary*** operators.

**Boolean Values**

…

**Logical Operators**

There are some operations that can be applied to Boolean values themselves. JavaScript supports three logical operators: ***and, or,*** and ***not.***

The **&&** operator represents logical **and.** It is a binary operator, and its results is true only if both the values given to it are true.

The **||** operator denotes logical **or.** It produces true it either of the values given to it is true.

***Not*** is written as an exclamation mark (**!**). It is a unary operator that flips the value given to it **!true** produces **false** and **!false** gives **true.**

A **Ternary** operator operates on three values. It is written with a question mark and a colon:

console.log(true ? 1 : 2);

// 1

console.log(false ? 1 : 2);

// 2

The values on the left of the question mark “picks” which of the other two values will come out. **When it is true, the middle value is chosen, and when it is false, the value of the right comes out.**

**Automatic Type Conversion**

When an operator is applied to the “wrong” type of value, JavaScript will quietly convert that value to the type it wants, using a set of rules that often aren’t what you want or expect. This is called ***type coercion.***

console.log(8 \* null)

// → 0

console.log("5" - 1)

// → 4

console.log("5" + 1)

// → 51

console.log("five" \* 2)

// → NaN

console.log(false == 0)

// → true

When comparing values of the same type using ==, the outcome is easy to predict: **you should get true when both values are the same, except in the case of NaN.** But when it differs JavaScript uses a complicated and confusing set of rules to determine what to do. **In most cases, it just tries to convert one of the values to the other value’s type.** However, when ***null***  or ***underfined*** occurs on either side of the operator, it produces true only if both sides are one of ***null*** or ***undefined.***

console.log(null == undefined);

// → true

console.log(null == 0);

// → false

**Short-Circuiting of Logical Operators**

In the case of **true || X,** no matter what **X** is even if its never evaluated. The same goes for **false** **&& X,** which is false and will ignore X. This is called ***short-circuit evaluation.***

*Chapter 2*

***Program Structure***

**Expressions and Statements**

A fragment of code that produces a **value** is an **expression.** Every value that is written literally (such as 22 or “psychoanalysis”) is an expression. An expression between parentheses is also an expression, as is a binary operator applied to two expressions.

If an expression corresponds to a sentence fragment, a JavaScript *statement* corresponds to a full sentence in a human language. ***A program is simply a list of statements.***

**Variables**

When a variable points as a value, that does not mean it is tied to that value forever. The = operator can be used at any time on existing variables to disconnect them from their current value and have them point to a new one.

var mood = "light";

console.log(mood);

// → light

mood = "dark";

console.log(mood);

// → dark

You should imagine variables as **tentacles,** rather than boxes. They do not **contain values;** they **grasp** them – two variables can refer to the same value. ***A program can access only the values that it still has a hold on.*** *When you need to remember something, you grow a tentacle to hold on to it or you reattach one of your existing tentacles to it.*

A variable not assigned a value returns the value **undefined**

**Keywords and Reserved Words**

Words with special meaning, such as ***var***, are *keywords*, and they may not be used as variable names.

**The Environment**

The collection of variables and their values that exist at a given time is called the *environment.*

**Functions**

A function is a price of program wrapped in a value. Executing a function is called *invoking, calling, or applying it.* Values given to functions are called *arguments.* The console.log function isn’t a simple variable. **It is actually an expression that retrieves the *log* property from the values held by the console variable.**

**Return Values**

When a function produces a value, it is said to ***return* that value**. Anything that produces a value is an expression in JavaScript, which means function calls, can be used within larger expressions.

**Updating Variables Succinctly**

Especially when looping, a program often needs to “update” a variable to hold a value based on that variable’s previous value.

coutner = counter + 1

JS provides a shortcut

counter += 1;

For counter +=1 and counter -=1, there are even shorter equivalents: counter++ and counter--

**Summary**

JavaScript systems always put a number of useful standard variables into your environment. Functions are special values that **encapsulate** a program. You can invoke them by writing *functionName(argument1, argument2).* Such a function call is an *expression*, and may produce a value.

*Chapter 3*

***Functions***

Typical adult English speakers have some 20,000 words in their vocabulary. Few programming languages come with 20,000 commands built in. And the vocabulary that is available tends to be *more precisely defined, and thus less flexible than in human language*. Therefore, we usually have to add some of our own vocabulary to avoid repeating ourselves.

**Defining a Function**

A function is just a regular variable definition where the value given to the variable happens to be a **function.**

A function is created by an expression that starts with the keyword *function.* A *return* statement determines the value the function returns. When control comes across such a statement, it immediately jumps out of the current function and gives the returned value to the code that called the function. The *return* keyword without an expression after it will cause the function to return *undefined.*

**Parameters and Scopes**

The parameters to a function behave like **regular variables** but their initial values are given by the caller of the function not the code in the function itself.

An important property of functions is that the variables created inside of them, **including their parameters, are local to the function.** This “localness” of variables applies only to the parameters and to variables decalred with the *var* keyword inside the function body. **Variables declared outside of any function are called *global.***

Functions are the only thing that create a new scope.

**The Call Stack**

The place where the computer stores this context is the *call stack.* Every time a function is called, the current context is put on top of this “stack”. When the function returns, it removes the top context from the stack and uses it to continue execution. Creating an infinite loop the computer will run out of space or “blow the stack”

**Closure**

This feature being able to reference a specific instance of local variables in an enclosing function – is called a **closure.** ***A function that “closes over” some local variables is called a closure.***

function wrapValue(n) {

var localVariable = n;

return function() { return localVariable; };

}

var wrap1 = wrapValue(1);

var wrap2 = wrapValue(2);

console.log(wrap1());

// → 1

console.log(wrap2());

// → 2

…………. More notes form the section?!?!??

**Recursion**

It is perfectly okay for a function to call itself, *as long as it takes care not to overflow the stack.* **A function that calls itself is called *recursive.*** Recursion allows some functions to be written in a different style.

Running through a simple loop is a lot cheaper than calling a function multiple times.

**Functions and Side Effects**

Functions can be roughly divided into those that are called for their side effects and those that are called for their return value.

Functions that create values are easier to combine in new ways than functions that directly perform side effects.

A pure function has the pleasant property that, when called with the same arguments, it always produces the same value (and doesn’t do anything else).

**Summary**

A key aspect in understanding functions is understanding local scopes. Parameters and variables declared inside a function are local to the function, re-created every time the function is called, and not visible from the outside. Functions declared inside another function have access to the outer function’s local scope.