What is a program

- Program - a sequence of **statements** that are executed in a certain order (by default sequentially, from top to bottom).
In synchronous programs only **one statement is executed at a time**.
During the execution of these statements **input data** is (optionally) received from the outside of program, then through the use of **expressions** somehow transformed and/or used to create new data, and then the resulting data is **output** to the outside of program.
Program also operates on the data that is created during the program execution.
Data is passed throughout the program by associating it with a **variable** and later referencing variable name to access data created during the previous steps of the program.
Some programs start their execution, do their task and output the result. After which they terminate (finish). Such programs are sometimes called scripts. They are usually relatively simple.
Other programs run indefinetely, until an external command is given to terminate a program (e.g. 'close' button is pressed).

Values, data types

- Value - **a piece of data** that a program works with. Always belongs to one of the **types**.
- Literal value - value that is created during program execution. For instance, if you need to add 3 to some value, you would write literal value `3` in your code in order to perform this operation.
- (Data) Types - different categories of data that are defined by a programming language. Different languages have different types.
JavaScript has 8 types, of them **7 primitive** types and an **object** type.
Types determine what kind of **operations** can be performed on values of that type. For instance a string of text can be converted to lower case, while two numbers can be multipled by each other.
Performing an **invalid** (illegal) operation on a value (e.g. trying to multiply two strings) is a **mistake** & usually (but in JS not always) results in an explicit **program error**.
Values can be **converted** between types, becoming values of **another type**. However not all type conversions are possible or make sense. Number 512 can become a string '512'. But string 'hello' can't become any meaningful number.
Additionally dividing data between types allows programming language to store data and operate on it more efficiently. For instance arrays are optimized to allow very fast iteration of their elements.
N.B. In JS arrays aren't its own type, but rather a special variety of objects.

Boolean literals, null, undefined

```
true
false
// Can be only this two values
null // lack of real value, " not hin g"
undefined // lack real of value, " not hin g"
```

null and undefined are functionally similar, but two different types, duplication is due to historical reasons.

Language itself uses undefined most of the times (e.g. a function without a return, returns undefined). Because of that the convention is to use null when you (as opposed to the program) need to use "-nothing" as a value

Operators, expressions, variables

- Operator - special symbol or symbols (e.g. +) that takes values (operands) and results in a new value
- Expression - one or many operators with their operands. Always results in (resolves to) a single value
- Variable - a container (label, binding) with a programmer-defined name that is associated with (holds) a single value.
Once a value associated with (assigned to, bound to) a varibale, it can be used in the following code by referencing variable name.
In JS any variable can hold any type (JS is a dynamically typed language). That is, a variable doesn't have a type, but its value does.
- Assignment operaror (`=`) - takes value on its right side and puts it in its left-side operand (variable, object property, array index)
It has a very low precedence, so whatever is being assigned almost always resolves to value and then is assigned.

Expressions 1

```
/* Literals */
// Literal of any type is an expression that
resolves to itself
5 // literal
" foo " // literal
/* Operators with operands */
5 + (7 * 13) // math expression
```


Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
http://crosswordcheats.com

Expressions 1 (cont)

```
> true && (false || !false) // logical expression
+"562" // unary operator
```

Expression is something that resolves to a value (is evaluated) during program execution

Complex expression resolution example

```
/* Preparation */
let myVarA = 3
const myVarB = 2
const arr = [5, 6, 7]
function square d(n umber) {
    return number * number
}
/* Example */
55 / (12 - arr[2]) - +(true && !false) / square -
d(m yVarB) - ++myVarA
// 55 / (12 - 7) - +(true && !false) / square -
d(m yVarB) - ++myVarA
// 55 / 5 - +(true && !false) / square d(m yVarB)
- ++myVarA
// 55 / 5 - +(true &&  true) / square d(m yVarB) -
++myVarA
// 55 / 5 - + true / square d(m yVarB) - ++myVarA
// 55 / 5 - +true /  4 - ++myVarA
// 55 / 5 -  1 / 4 - ++myVarA
// 55 / 5 - 1 / 4 -  4
// 11 - 1 / 4 - 4
// 11 - 0.25 - 4
// 10.75 - 4
// 6.75
```

See operation precedence table for order in which subexpressions resolve

Non-primitive (composite) types

Types that have internal structure and contain primitive types or other non-primitive types as its components. They are alternatively called composite, compound or aggregate data types.

Non-primitive (composite) types (cont)

Is JS there is only one non-primitive data type - object. Its internal structure is a collection of key-value pairs, where each value is stored and accessed by its key (a string that programmer chooses similar to a variable name).

Such data types is useful when we need to store heterogeneous, but related values. For instance, different information about a user (his name, age, date of birth etc.).

However there are special variates of objects that behave differently. Two main subtypes are arrays and functions.

Arrays store many values in themselves, each values is stored at and accessed by an integer index. Indices in array are contiguous, that is after 0 goes 1, then 2, then 3 and so on.

Arrays are useful when we need to store a list of similar values. For instance, a series of numerical measurements.

N.B. In JS it's technically possible skip indices and after storing value at an index 0, for example, store next one at an index 10. However that breaks internal optimizations of arrays and is a fundamentally wrong way to use them.

Functions are special syntactic constructs (but in JS they are also values). They contain a series of statements in themselves and can be called (invoked, executed) in different places throughout the program. When called they (optionally) take some input data, execute its contained statements and (optionally) return result back where they were called.

Arrays

```
[1, 2, 3] // literal
[1, " foo ", true, null, undefined] // can hold
values of multiple types
[[1, 2], [3, 4]] // can hold other arrays as
elements
[{amount: 6}, {amount: 16}] // can hold objects as
elements
const arr = [11, 22, 33]
arr[1] // get array element value, resolves to 22
arr[1] = 0 // set a element value, now arr is [11,
0, 33], overwrites existing value
arr.length // special property, it contains number
of elements (is this case 3)
```

Each array element is stored at an index. Indices start at *0*, not 1.

For iteration over arrays see block on iteration

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
http://crosswordcheats.com

Iteration (loops)

```
let counterA = 0
while (counterA < 10) {
   // at the start of every iteration checks if
condition is true
   // If true, runs body, then checks again
   // If false, then the loop is finished
    con sol e.l og( 'co unterA equals ' + counterA)
    cou nterA++
}
for (let i = 0; i < 10; i++) {
      con sol e.l og('i equals ' + i)
}
// let i = 0 - runs one time when loop starts
// i < 10 - checks condition at the start of every
iteration,
// same logic as in while loop
// i++ - runs at the end of every iteration
// two loops above are functi onally identical,
// but for loop encaps ulates counter (i) declar -
ation within its syntax
// and separates main action of the loop in its
body
// from changing counter value
const arr = ['a', 'b', 'c']
for (let i = 0; i < arr.le ngth; i++) {
      con sol e.l og( 'el ement of arr at index ' +
i + ' equals ' + arr[i])
}
let counterB = 0
while (counterB < 10) {
   if (counterB === 3) {
        cou nterB++
        con tinue // forces immediate exit from
current iteration
       // goes to next iteration
       // can be used in 'for' loops as well
   }
    con sol e.l og( 'co unterB equals ' + counterB)
    cou nterB++
```

Iteration (loops) (cont)

```
> }
let counterC = 0
while (true) { // condition will never be false
  console.log('counterC equals ' + counterC)
  counterC++
  if (counterC === 5) {
    break // forces immediate loop termination
    // program execution goes further
    // can be used in 'for' loops as well
  }
}
```

Loop body runs repeatedly (iterates) for as long as the loop condition is true. When the loop condition becomes false, the loop terminates, and program execution contiues further.

Loop use cases

Usually loops are used to do the same action, but with a different value that changes between iterations.
That value is stored in a variable (often called counter, index, i) outside of loop body and is changed inside loop body (always in case of 'while' loops and occasionally in 'for' loops) or, in case of 'for' loops, in a special expression inside parentheses (last of three expressions).
Most often that repeated action is about doing something with an array element and the value of 'i' variable is used as an index to access an array element at that index.
'for' loops are good for that case, because number of iterations are known at the start of the loop (for instance array length equals the number of iterations for iterate over an entire array).
Alternatively loops are used to repeat some action until something happens (for instance network request is successful).
'while' loops are good for that case, because the number of iterations are unknown at the start of a loop (we don't know how many times we have to repeat the request until it succeeds).

Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
http://crosswordcheats.com

Statements

A statement is a command to a computer to do something.
Programming languages (including JS) have a number of special words (**keywords**) that, when used in the code, indicate to the computer that a statement is issued and needs to be executed when program is run.
In JavaScript statements can be on single line (for instance variable declaration), or on multiple (for instance 'if' statement).
- Block statement (`{}`) - a special statement that contains other statements within it. Rarely used on its own, usually it is used as a part of another statement ('if', 'while', 'for', 'function' etc.). When used as a part of another statement it's called "**body**" (e.g. function body).
In JS statements without blocks are (optionally) terminated by a **semicolon** (`;`).
N.B. More than one statement can be on a single line, in which case they **must** be separated (terminated) by a semicolon. Example:

```
let a; let b = 5; let c; // last semicolon is not mandatory
```

There are also a few other edge cases when semicolons must be used, because without them it's impossible to unabmiguously divide code into statements. But there cases are very rare.
Even though semicolons are optional begginners are often encouraged to still use them, because that way it's easier for programmer to see where one statement ends & another begins.

Primitive types

Types with no internal structure (e.g. a single string of text, a single number).
Primitive types are immutable, that is their value can't be changed. Examples:

```
5 + 10 // two values are used to create new value
let myVar = 7
myVar = myVar + 4 // existing myVar value is used (along side 4) to create new value
// That new value overwrites existing myVar value
```

N.B. strings are considered primitives, but technically have internal structure, since it's possible to access (but not change) its individual characters.

Number literals

```
12 // integer
3.45 // float
-512 // negative
0
Infinity // also -Infinity
```

Number literals (cont)

```
> NaN // "not a number", special value
// results from illegal operations such as 5 / "foo"
```

String literals

```
'foo',
" foo " // same as single quotes
/* Escaping */
'I don\'t know' // I don't know
"Jack \"Ow l\" Smith" // Jack " Owl " Smith
" Don't need to escape "
"Use this to escape \\ in string s" // Use this to escape \ in strings
/* Special characters */
" First line. \nSecond line"
/*
First line.
Second line
*/
```

Variable declaration

```
/* Declare & assign (initialize) */
var myVarA = 2 // outdated keyword, don't use
let myVarB = 4
const myVarC = 8 // can't be reassigned later
// recomm ended to be used by default
/* Only declare */
let myVarD // has undefined as value
/* Reassign */
myVarB = " foo "
myVarD = [1, 2, 3]
```


Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
http://crosswordcheats.com

Expressions 2

```
/* Variables */
const myVar = " Hello world"
myVar // resolves to a value stored in the variable
/* Object properties */
const human = {
    name: " Joh n",
   age: 20,
}
human.age // resolves to a value stored in the
property
/* Array elements */
const arr = [10, 20, 30]
arr[1] // resolves to a value stored at that index
/* Function calls */
// Resolves to whatever is returned by a funtion
function sum(a, b) {
    return a + b
}
sum(5, 15)
```

Places to use values

```
const value = "I'm just a value"
// As a part of a larger expression
value + ". Or am I?"
// Assigned to variables
const newValue = value
// Assigned to object properties
const obj = {}
obj.me ssage = value
// Put in an array at a certain index
const arr = [1, 2, 3]
arr[0] = value
// As function parameters
function isBool ean (to Test) {
    return toTest === false || toTest === true
```

Places to use values (cont)

```
> }
isBoolean(value)
// As a function returned value
function getTen() {
  const value = 10
  return value
}
```

Expressions resolve to values & values can used in these places

Objects

```
{
   key: " val ue",
    key2: 5,
} // literal
// can hold arrays & other objects
const post = {
    text: "Come and join me!",
    cat ego ries: ["fu n", " use r-f rie ndl y",
" pay wal led "],
    isV isible: true,
    cre atedAt: " 202 2-0 6-1 2T1 8:5 8:1 3.0 -
59Z ",
    eng age ment: {
        likes: 5,
        com ments: 0,
        shares: 0,
   },
}
post.i sVi sible // get value stored in property
" isV isi ble "
post.text = "Best time of your life" // set value
for property " tex t",
// overwrites existing value
post.e nga gem ent s.like // get and set deeply
nested properties
// by chaining property names
post.foo // getting non-ex isting property
evaluates to undefined,
// but doesn't result in explicit error
post.f oo.bar // this results in an error,
```


Sponsored by CrosswordCheats.com
Learn to solve cryptic crosswords!
http://crosswordcheats.com

Objects (cont)

> // because it's impossible to access property of *undefined*

Objects are containers for values, where each values is associated with a string key (a property). Key-value pairs are separated by a comma.

They are used to store values of different types that are related in some way (for instance to represent real world entities: people, cars, bank accounts etc.)

Conditionals

```
let conditionA = true
if (condi tionA) {
    con sol e.l og( "I will run")
}
conditionA = false
if (condi tionA) {
    con sol e.l og( "I won't run")
}
// any condition must resolve to a boolean value.
// If an expression in condtion isn't boolean,
it's converted to boolean
conditionA = 5
if (condi tionA) {
    con sol e.l og( "I, too, will run")
}
// logical operators can be used to create complex
conditions
const conditionB = false
if (condi tionA && !condi tionB) {
    con sol e.l og( " Complex condition is true!")
}
// use esle keyword to do something if condition
is false
if (condi tionB) {
    con sol e.l og( " Either I will run")
} else {
    con sol e.l og( "Or me")
}
// use else if keyword to check for multiple cases.
```

Conditionals (cont)

```
> // It checks conditions until the first true condition is met,
// that branch runs, following branches are ignored
const age = 15
if (age < 5) {
  console.log("Baby")
} else if (age < 16) {
  console.log("Child") // this will run
} else if (age < 30) {
  console.log("Young adult") // this and following branches won't run
  // even though condition is true,
  // because previous branch was executed already
} else if (age < 60) {
  console.log("Adult")
} else { // this is optional, to run if all above condtions are false
  console.log("Old man")
}
```


Sponsored by **CrosswordCheats.com**
Learn to solve cryptic crosswords!
http://crosswordcheats.com