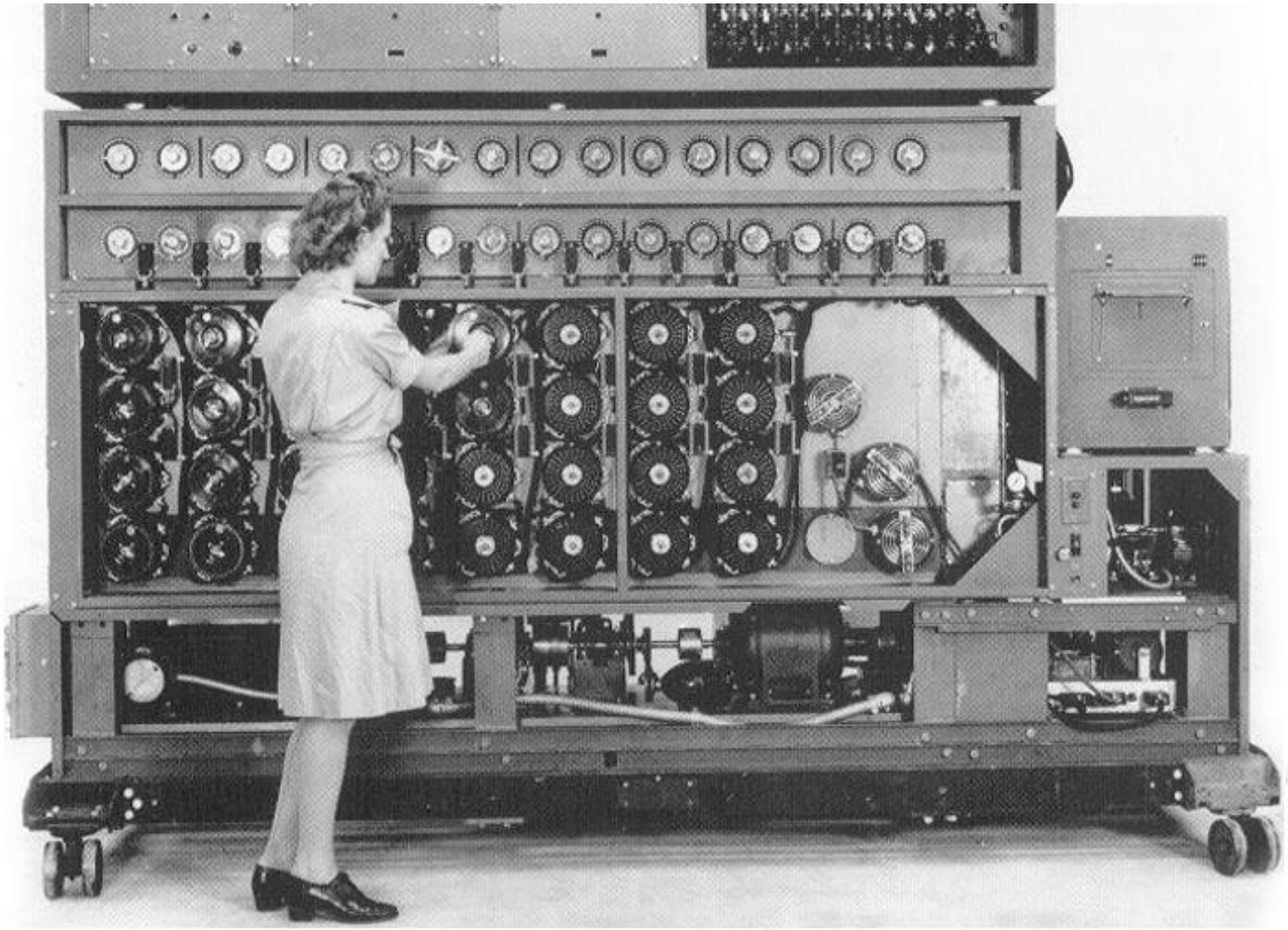
Introduction to Computer Science using JavaScript

Ritwik Dutta

Table of content

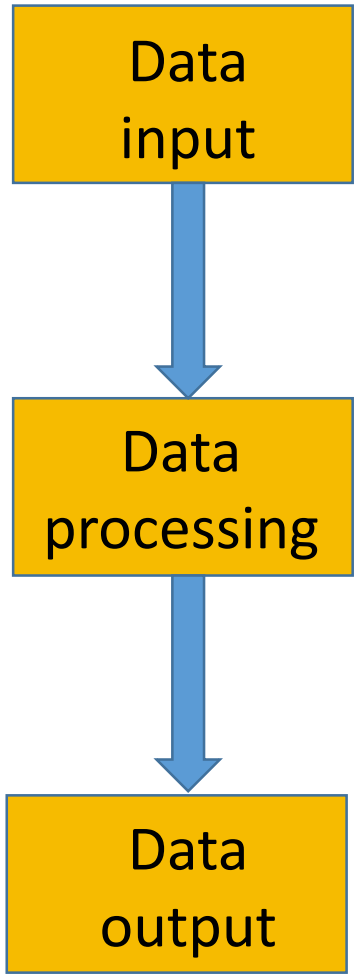


Computers & programming

Computer definition

- Machine that executes **tasks** according to **instructions**
- Tasks are basic operations
- Instructions are commands in a machine-readable format
- Tasks operate on input data and generate output data

Structure and flow



Input devices: Mouse, keyboard, mic, camera, scanner

Processing devices: Central Processing Unit (CPU), Arithmetic Logic Unit (ALU)

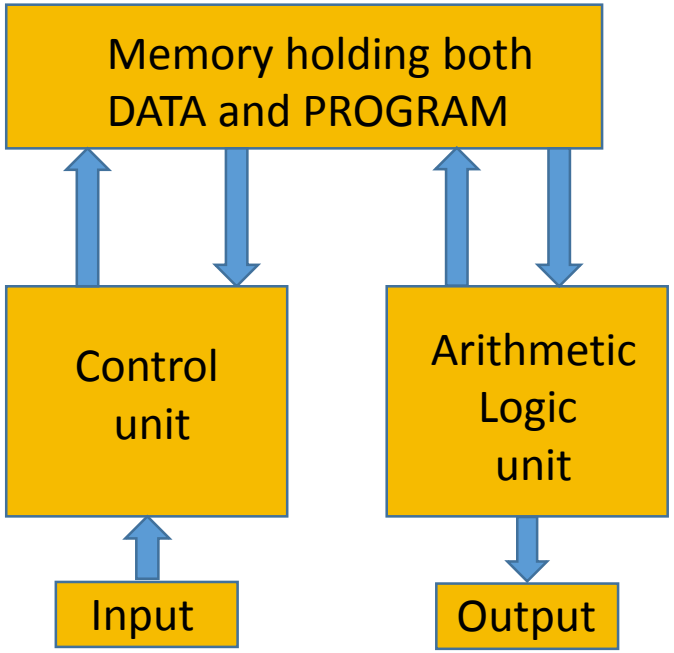
Output devices: monitor, printer, speaker, modem

Structure of a modern computer

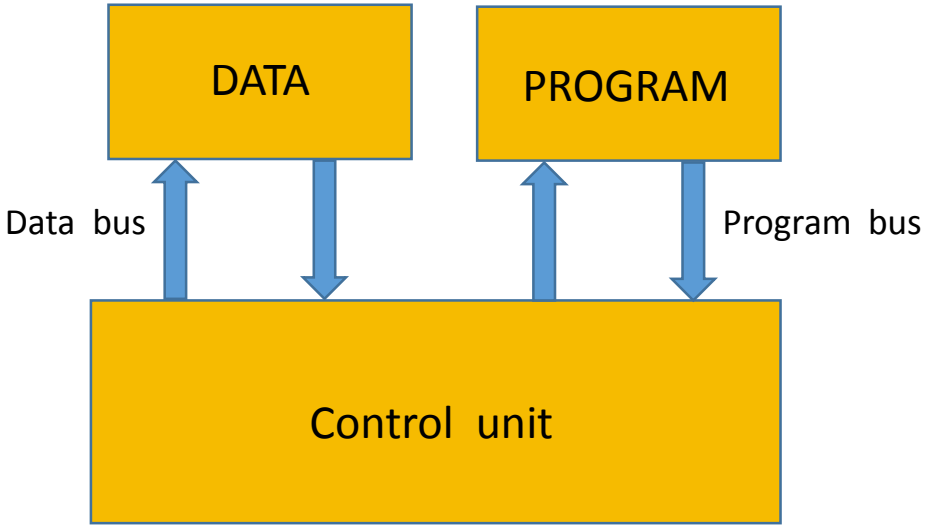
- Basic components
  - **Sequencer** determines order of instructions to be fetched
  - **Fetch unit** gets data and instructions stored in memory
  - **Control unit** determines type of operations to be performed
  - **CPU** executes different arithmetic and logic operations on different data according to instructions in the **Arithmetic Logic Unit (ALU)**
  - **Store unit** stores processed data back to memory
  - **Peripherals** allow information to be stored or retrieved from an external source (*e.g.*, CDROM drives, USB flash drivers, *etc*.)
- Modern digital computers based on integrated circuits are many times smaller and more capable than their mechanical and analog predecessors

Where are instructions and data stored?

- Instructions and data are stored in and fetched from memory
- Both can be in the same memory - Von Neumann architecture
- Both can be in different memories - Harvard architecture



Von Neumann architecture



Harvard architecture

Introducing registers

- Regular memory in a computer is often big and slow
- To speed up operations, there is a small amount of very fast storage using registers often implemented inside the CPU
- Register are used to quickly accept and store data that are being used immediately by the CPU
- Registers are usually denoted by the alphabet letters A, B, C, … *etc*.
- There may be multiple registers of each type designated by A0, A1, A2, … , *etc*.

Computer hardware

- Hardware refers to physical components that make up a computer system
- These components are physically connected to the computer and can be physically touched
- There are many different kinds of hardware that can be installed inside or connected to the outside of a computer
- Examples of internal hardware are the CPU, the computer memory, the CDROM drive, *etc*.
- Typical examples of external hardware are keyboard, mouse, computer monitor, printer, *etc*.

Computer algorithm

- An algorithm in general is a step by step procedure or formula to do a task or operation
- In order to tell a computer what calculation(s) to do, one needs to also specify how exactly to do it
- This “how-to-do-it” is nothing but a computer algorithm
- A computer algorithm is specified by a sequence of well-defined machine-readable instructions
- Algorithms are used for calculation, data processing, and sometimes reasoning
- An efficient algorithm is one that executes (produces the results) quickly and/or uses fewer resources … we refer to these metrics as the **time** and **space complexity** of an algorithm respectively

Practical algorithm

Different algorithms to get to Tom's house from the airport

ALGORITHM 1

1. Collect luggage
2. Go to the taxi stand
3. Get into a cab
4. Give the driver Tom's address
5. Arrive at Tom's house

Algorithm 2 takes more time (buy ticket, slow bus)
Algorithm 2 uses more resources (bus, Tom's car)

ALGORITHM 2

1. Collect luggage
2. Go to the bus stand
3. Get on to bus no. 8
4. Buy ticket for zone 2
5. Get down at 6th street
6. Call Tom's cell phone
7. Tom picks you up in his car
8. Arrive at Tom's house

Computer program

- A program is a sequence of instructions written to perform a certain task
- The program implements a formula or an algorithm to do a certain task, but is itself not the formula or the algorithm
- A programmer decides on a suitable algorithm and writes the computer program, often in a human readable form, called the **source program**
- The source program is first loaded or stored in the computer memory
- Next, the program is converted to a machine readable form
- The program instructions are then fetched from memory and executed by the CPU inside the computer
- It is highly desired that the instructions implement an efficient algorithm

Computer software

- Software (SW) is a generic term used to distinguish it from hardware (HW)
- SW comprises a set of instructions and their associated data stored on a media such as flash, disk, or tape drives
- These instructions constitute a computer program
- Can think of software as a computer program stored on a media
- A computer program directs a computer's processor to perform specific operations
- Software programs are written in specific languages
- There are two classes of languages: **low-level languages** and **high-level languages**

Low-level programming (1)

- A low-level programming language is what a computer or a machine understands
- This low level programming language is called the **machine language**
- Machine language is the native binary language in terms of 1's and 0's that a machine understands
- Takes specialized knowledge to program in machine code

Machine instruction	Machine operation
0000 0000	Restart
0000 0010	Stop
0001 0000	Skip next instruction

Low-level programming (2)

- A machine language instruction explicitly tells the CPU what operation it is supposed to do, where in memory to get the operands from, how exactly to do the computation with those operands, and where in computer memory to store the results of the calculation
- Each type of CPU has its own unique machine language
- Humans are not able to think in terms of or interpret a machine language easily
- Programmers therefore commonly use a more English-like high-level language (for programming) to develop their source code that ultimately gets translated into the machine language

High-level programming

- High-level languages (such as Basic, C, JavaScript, *etc*.) are very English-like, hence easier to conceive of and interpret
- Programmers therefore prefer to program in high-level languages
- A computer does not understand any high-level language
- The computer needs all its instructions to be given in terms of a low-level machine language that it can understand
- Programs written by a programmer need to be translated from a high-level language into a machine language understood by the target computer

Compiler(1)

- One level of abstraction up from machine code, a **compiler** is a software program that translates a source code written in a high-level programming language to a lower level **assembly language** for a target machine
- The most common reason for this translation is to create an **executable program** that the computer understands and can execute
- In the process of translation, a good compiler also gives warnings and flags errors in programming

Assembly instruction	Machine operation
MOVE A0, 2000	Copy contents of register A0 to memory location 2000
MOVE (B0), D2	Copy contents of location pointed to by register B0 to register D2
ADD #15, C3	Add 15 to register C3 and put sum in C3

Compiler(2)

- Compilers can also optimize and generate more efficient target code

- Example

```
for i = 1 to 1000 do
    A = B/C;
```

inefficient

```
A = B/C;
for i = 1 to 1000 do
    skip; // do nothing
```

optimized

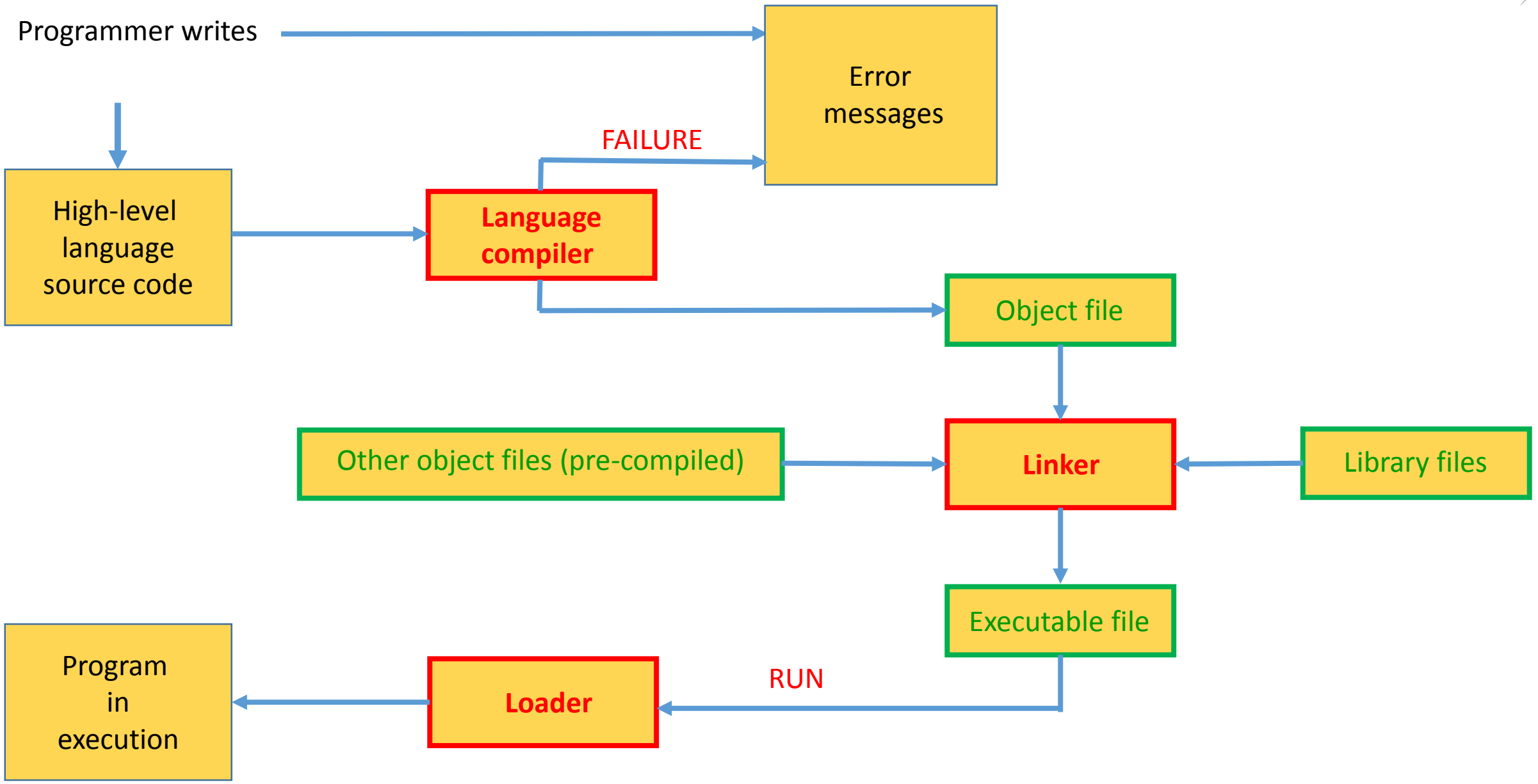
Assembler, linker, loader

- Assembly language is less sophisticated than high-level language but more sophisticated than low-level machine language
- Assembly language code is translated into machine language by an **assembler**
- The assembler converts assembly language programs into **object files**
- Object files contain machine instructions, data, and information needed to place instructions properly in the computer memory
- A **linker** merges different object files produced by an assembler, and creates common **executable file**
- The executable file is loaded into the computer's memory by a **loader**
- Executable instructions are then fetched from memory and executed by the CPU

Compiled program simple flow

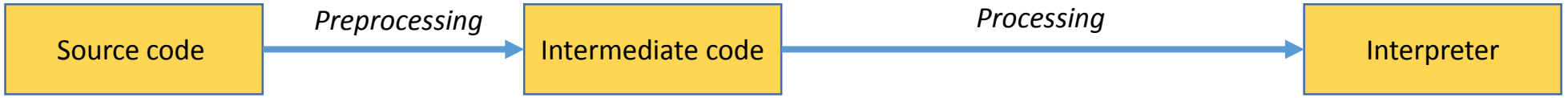
- A high level language program is written by a programmer
- The program is compiled by a compiler into the assembly language of the target CPU
- The assembly program is assembled by the assembler into machine language
- The assembled object files are linked by the linker into a single executable object file
- The object file is loaded by the loader into the computer memory for execution
- The instructions are fetched from the memory by the fetch unit
- The fetched instructions and data are executed by the CPU
- The results are stored in the memory by the store unit

Putting it in a picture

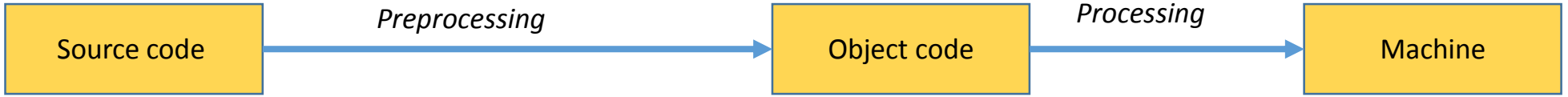


Interpreters *versus* compilers

- Both read input source code and analyze it
- Interpreters
  - Execute single instructions without object code generation, flag errors if any
  - Good for debugging & interaction

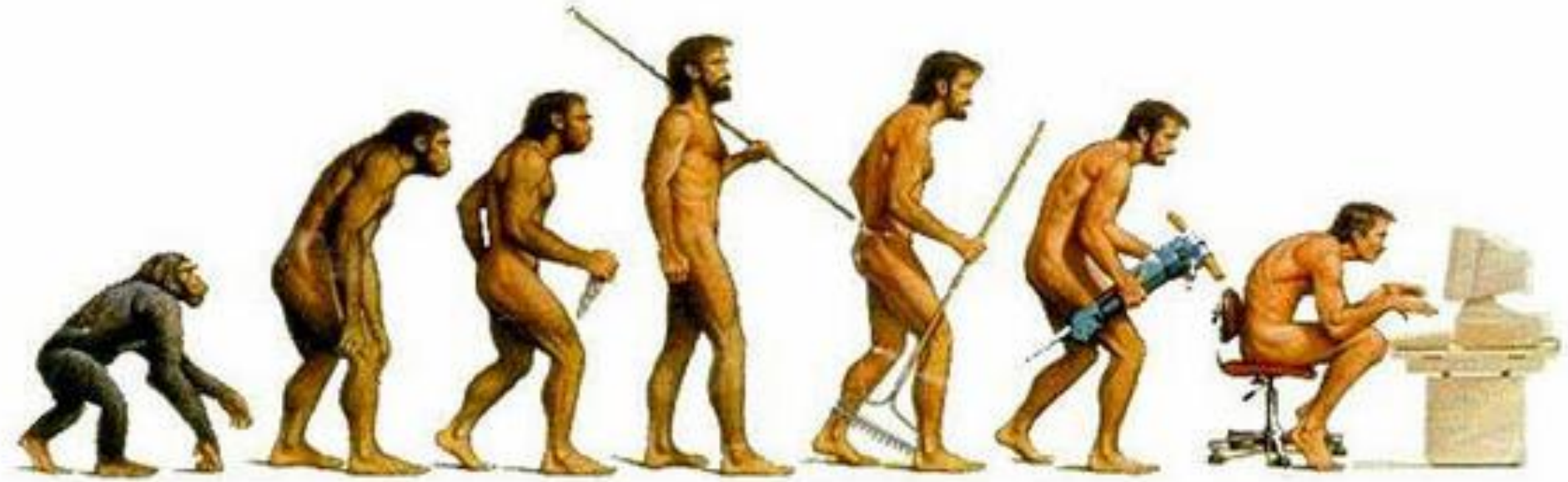


- Compilers
  - Read and analyze entire program, flag errors if any
  - “Improve” the program by making it more efficient
  - Generate object code executed on machine



Typical implementations

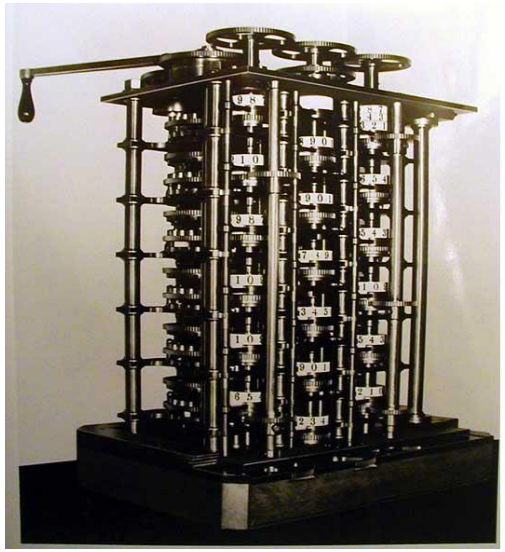
- Compilers
  - FORTRAN, C, C++, Java, COBOL, *etc*.
  - Strong need for optimization in many cases
- Interpreters
  - Python, PERL, Ruby, Java VM
  - Particularly effective if interpreter overhead is low relative to execution cost of individual statements
- Hybrid
  - Compile Java source to byte codes – Java Virtual Machine language
  - Interpret byte codes directly, or compile some or all byte codes to native code

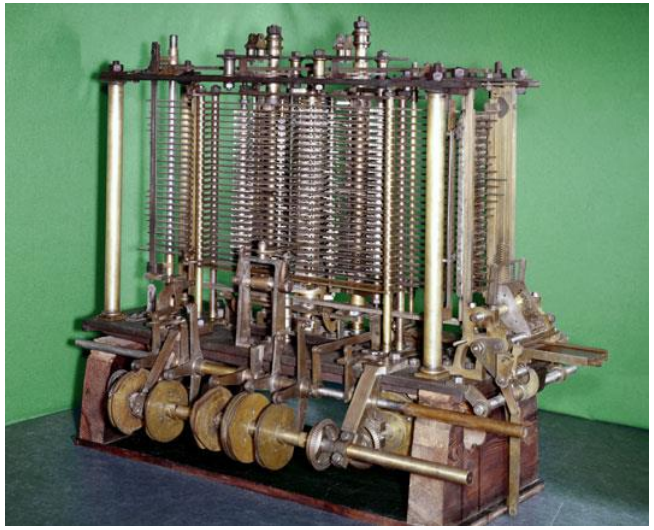


The evolution

Mechanical devices

- Pascal calculator (1642)
  - Performed addition using gears
- Leibniz stepped reckoner (1672)
  - Addition, multiplication, division, square roots using cylindrical wheels with movable carriage
- Babbage difference engine (1822)
  - Produced table of numbers
- Babbage analytical Engine (1837)
  - Performed calculations based on punched card instructions

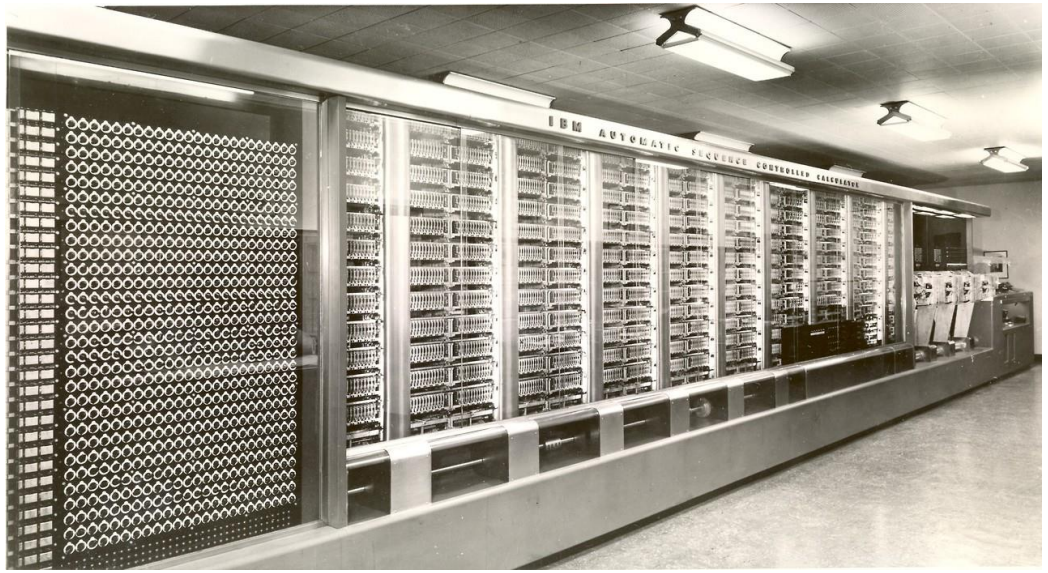




Electro-mechanical devices

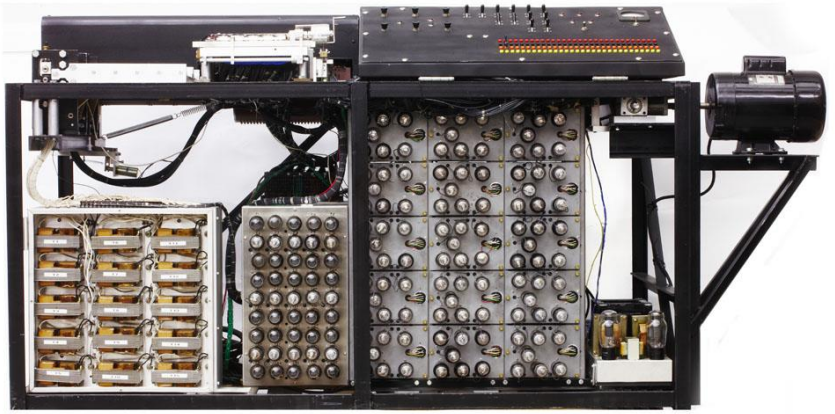
- Hollerith's tabulating machine (1889)
  - Holes punched in cards represented information to be tabulated … used for tabulation in US census

- Mark I (1944)
  - Sophisticated calculator from Harvard & IBM that used mechanical telephone replay switches to store information and accepted data on punch cards

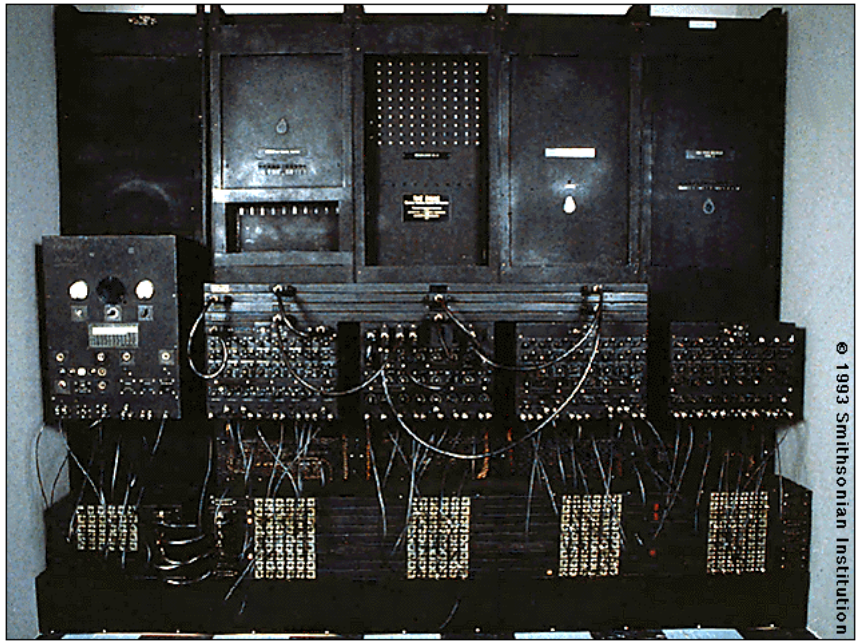


First generation computers

- Atanasoff-Berry computer (1939)
  - Used vacuum tubes doing binary arithmetic
  - Stored info by electronically burning holes in sheets of paper

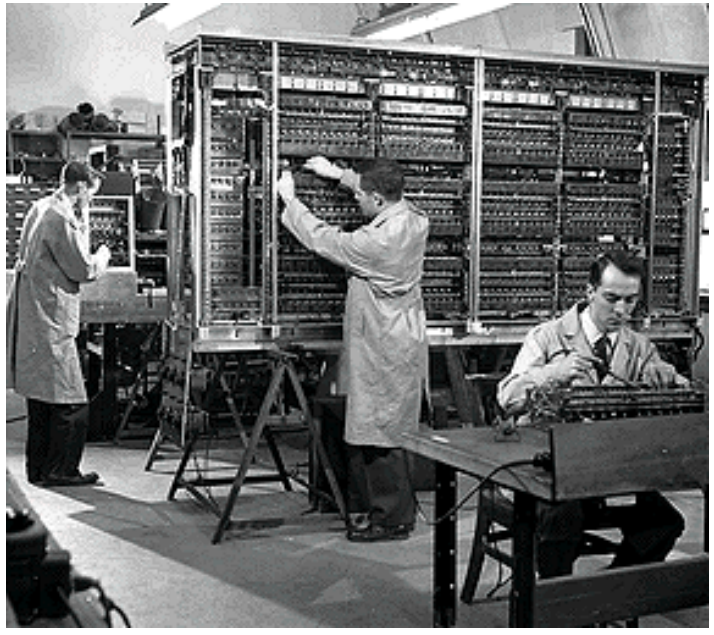


- ENIAC (1943)
  - Eckert and Mauchly
  - Electronic Numerical Integration and Calculator
  - 30 tons, 1500 sq ft., 17,000+ vacuum tubes
  - Solved a problem in 20 min that otherwise took three days to solve



Stored program computer(1)

- Alan Turing
  - Developed idea of “universal machine”
  - Perform many different tasks by changing a program (list of instructions)
- Von Neumann
  - Presented idea of stored program concept
  - The stored program computer would store computer instructions in a memory and execute on a CPU
- EDVAC and EDSAC (1944)
  - Von Neumann, Mauchly and Eckert
  - Binary serial computers that recognized machine language
  - Solved many problems by simply entering new instructions stored on paper tape

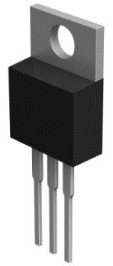


Stored program computer(2)

- UNIVAC (1951)
  - Mauchly and Eckert
  - First computer language C-10 developed by Betty Holberton who also developed first keyboard and numeric keypad
  - The UNIVAC I delivered to the U.S. Census Bureau was the first general purpose computer for commercial use

Second generation computers

- Bell Labs transistor (1947)
  - Shockley, Bardeen, Brittain invented transistor that replaced vacuum tubes since less expensive and increased calculating speeds



- IBM model 650 (1960)
  - Medium sized computer
  - Magnetic tape and high speed reel-to-reel tape machines replaced punched cards
  - Gave computers ability to read (access) and write(store) data quickly and reliably

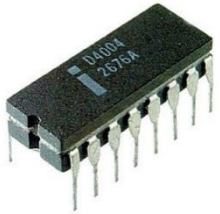
Third generation computers

- Integrated circuits (ICs)
  - Kilby and Noyce – working independently developed the IC (chip)
  - Allowed intricate circuits based on planar transistors to be printed and etched on silicon surface

- IBM 360 (1964)
  - One of the first mainframe computers using ICs
  - Communication with the mainframe via terminals
  - Covered range of applications, from small to large, both commercial and scientific

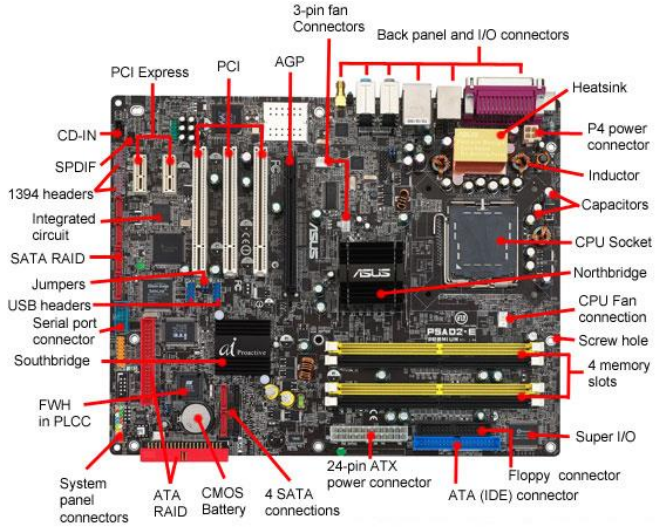
Fourth generation computers

- Microprocessor (1970)
  - Intel's invention of entire CPU on a chip made it possible to build the microcomputer (PC)
  - Altair 8800 was one of first PCs in 1975
  - Wozniak and Jobs designed and build first Apple Computer in 1976
  - IBM introduced IBM-PC in 1981



Today's personal computer (PC)

- Components are plugged into a motherboard featuring:
  - **BUS**: interconnection network that connects all components and allow data and control signals to travel from one to the other
  - **RAM slot**: where system memory chips are plugged into for the computer load the OS and drivers, thus making instruction processing faster



  - **CHIPSET**: *northbridge* IC responsible for communications between CPU and memory, the *southbridge* IC is responsible for the hard drive controller, I/O controller and integrated hardware such as sound card, video card, USB, PCI, ISA, IDE, BIOS, and Ethernet
  - **CPU socket(s)**: socket(s) where CPUs are connected; CPU contains the Arithmetic Logic Unit (ALU) that processes data and controls the flow of data between various units

Programming language evolution

- The first computer program was written by Ada Lovelace- in 1842-43
- The first “modern” language Plankalkül, designed by Konrad Zuse, was described in 1943, but implemented in 1998
- Languages invented in 1950-1960 are still used today
  - Fortran, LISP, COBOL, BASIC
- Most languages used today were invented in 1960-1970
  - C, Pascal, C, ML, SQL
- Many internet languages were developed in early/mid 1990
  - Python, Java, JavaScript, PHP

Inside programming languages

Philosophy of a programming language

- "… a language intended for use by a person to express a process by which a computer can solve a problem" - Hope and Jipping
- "… a set of conventions for communicating an algorithm" - E. Horowitz
- "… the art of programming is the art of organizing complexity" - Djikstra

Reality of a programming language

- Comprises vocabulary and rules designed to communicate instructions to a computer
- Contains abstractions for defining and manipulating data structures and controlling execution flow
- Usually split into two components: syntax (form) and semantics (meaning)
- High-level programming languages are portable across machine architectures and get compiled into low-level assembly language by compiler
- Assembly language gets assembled into executable machine code by assembler

Desires of a programming language

- Readable and maintainable
  - Comments, names, syntax
- Simple to learn and use
  - Small number of concepts that can be combined to do complex stuff
- Portable
  - Language standardization
- Offers abstraction
  - Control and data structures that hide detail, easy to express idea
- Compact
  - Express ideas concisely
- Efficient
  - Efficiently translated into machine code, efficiently executed, acquire as little space in the memory as possible

Example languages

- AI symbolic computation (Haskell, Lisp, Prolog)
- Scientific computing (Fortran)
- Business report generation (COBOL)
- Systems programming low level (C)
- Scripting (Perl, Python, TCL)
- Distributed systems mobile computation (Java)
- Web (PHP, JavaScript)

Why so many languages?

- Allows choosing the right language for a given problem

- If all you have is a hammer, every problem looks like a nail

Object-oriented programming

- A programming paradigm
- Main feature: communication between abstract objects
- Characteristics
  - “Objects” collect both data and operations
  - “Objects” provide data abstraction
- Key operations: Message Passing or Method Invocation

Object-oriented example

JAVASCRIPT

```
function Car(make, model, year) {
    this.make = make;
    this.model = model;
     this.year = year;
 }

var mycar = new Car("Eagle", "Talon TSi", 1993);
var  hiscar = new Car("Nissan", "300ZX", 1992);
```

Event-driven programming

- Browsers are interactive: alternate input and output
- GUI events: Windows/Mac/Xwindows
- Features: Icons, menus, dialog boxes, windows, buttons, scrollers, check boxes
- Common feature: User generates *events* via clicks, drags, keystrokes, timeouts
- One kind of interaction in browser: Hyperlinks
- Another type of interaction: Embedding of event-based JavaScript programs in HTML files

Structured programming(1)

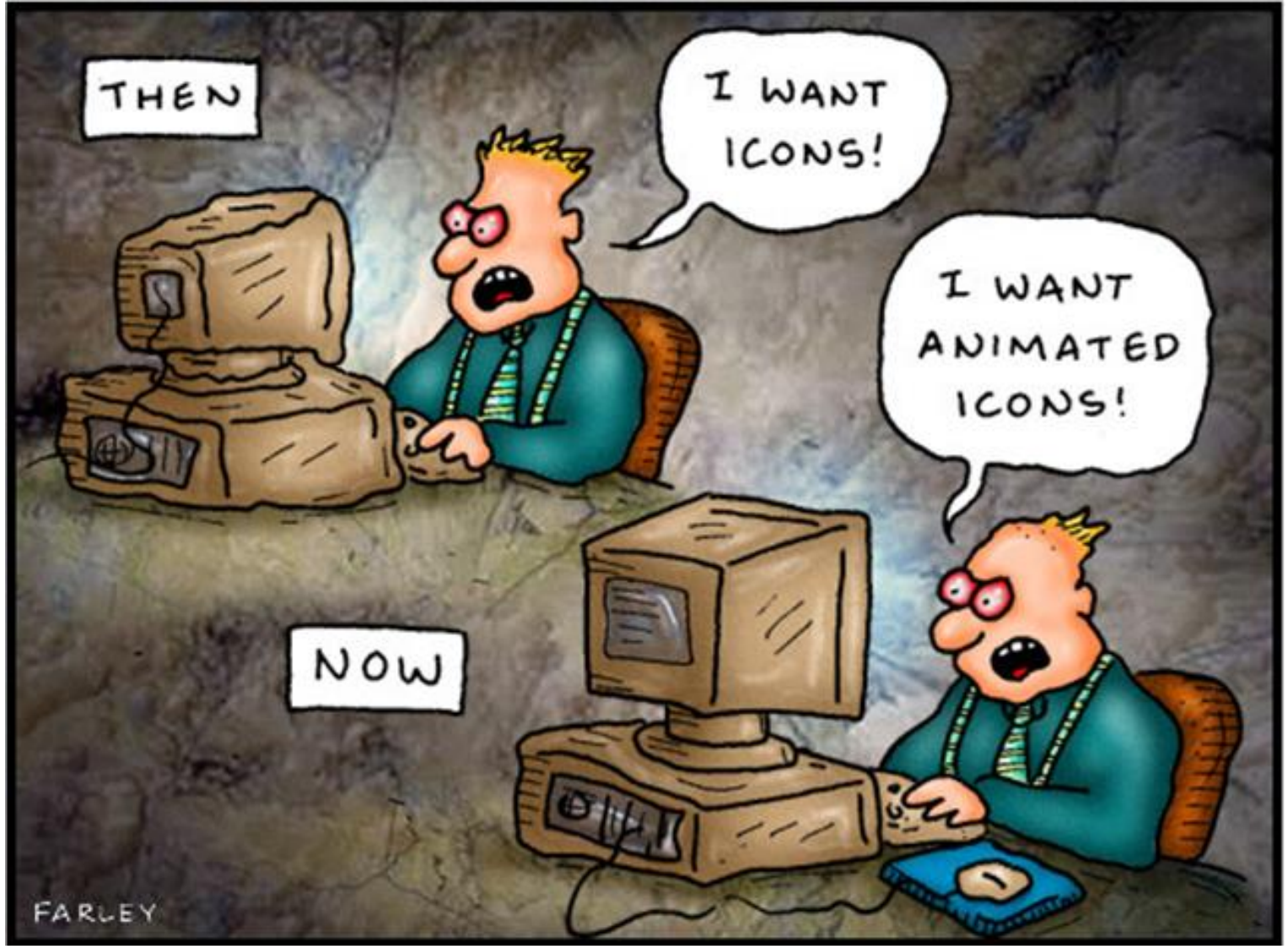
- Recent art of computer programming, also knows as **modular programming**
- Essentially a logical programming technique aimed at improving clarity, quality, and development time of a computer program
- Makes use of regular arithmetic operators just like any other programming technique, but the program flow in this case follows a simple hierarchical model that employs looping constructs such as "for," “do”, "repeat," “until”, and "while”

Note:

Not all programming languages feature all of the looping constructs)

Structured programming(2)

- One important goal is to avoid unmanageable and difficult-to-understand code caused by the usage of GOTO statements
- GO TO statements are often found in BASIC and FORTRAN programs
- Structured programming as promoted by high-level languages like C and Pascal discourage use of GO-TO statements, and encourage an overall program structure that reflects what the program is supposed to do, beginning with the first task and proceeding logically in an intuitive step-by-step manner
- A structured high-level program features loops, branch control structures and subroutines rather than GOTO statements; moreover, indentations and comments are used liberally to make the program and its intended logic clear to whoever is writing or reading the program



Why JavaScript?

The trend

- JavaScript is starting to play a central role in the implementation of cloud based mobile solutions
- Node.js, a software platform used to build scalable network applications based on Google's V8 JavaScript engine, is being adopted by millions of developers and enterprises for a wide range of use-cases
- All this is because JavaScript allows rapid development and maximizing user experience … is able to deliver rich, dynamic web content … is relatively lightweight …. has high ease of use

- In September 2012, industry analyst firm RedMonk, showed JavaScript as the top language of the enterprise
- JavaScript is a great language for teaching computer science basics --- programming, data structures, and algorithms

What is JavaScript (JS)?

- Object oriented language
- Runs in a host environment
- Syntax comes from the Java and C
- No input/output commands … used as a scripting language
- Host environment provides mechanisms for communicating with the outside world
- Most common host environment is the browser
- Code can be inserted into HTML pages and executed by browser
- Code can also be executed in a shell or console

JavaScript's web context

- HTML - used to store the content and formatting of a web page
- CSS - encodes the style of how the formatted content is graphically displayed
- JavaScript - used to add interactivity to a web page or create rich web applications

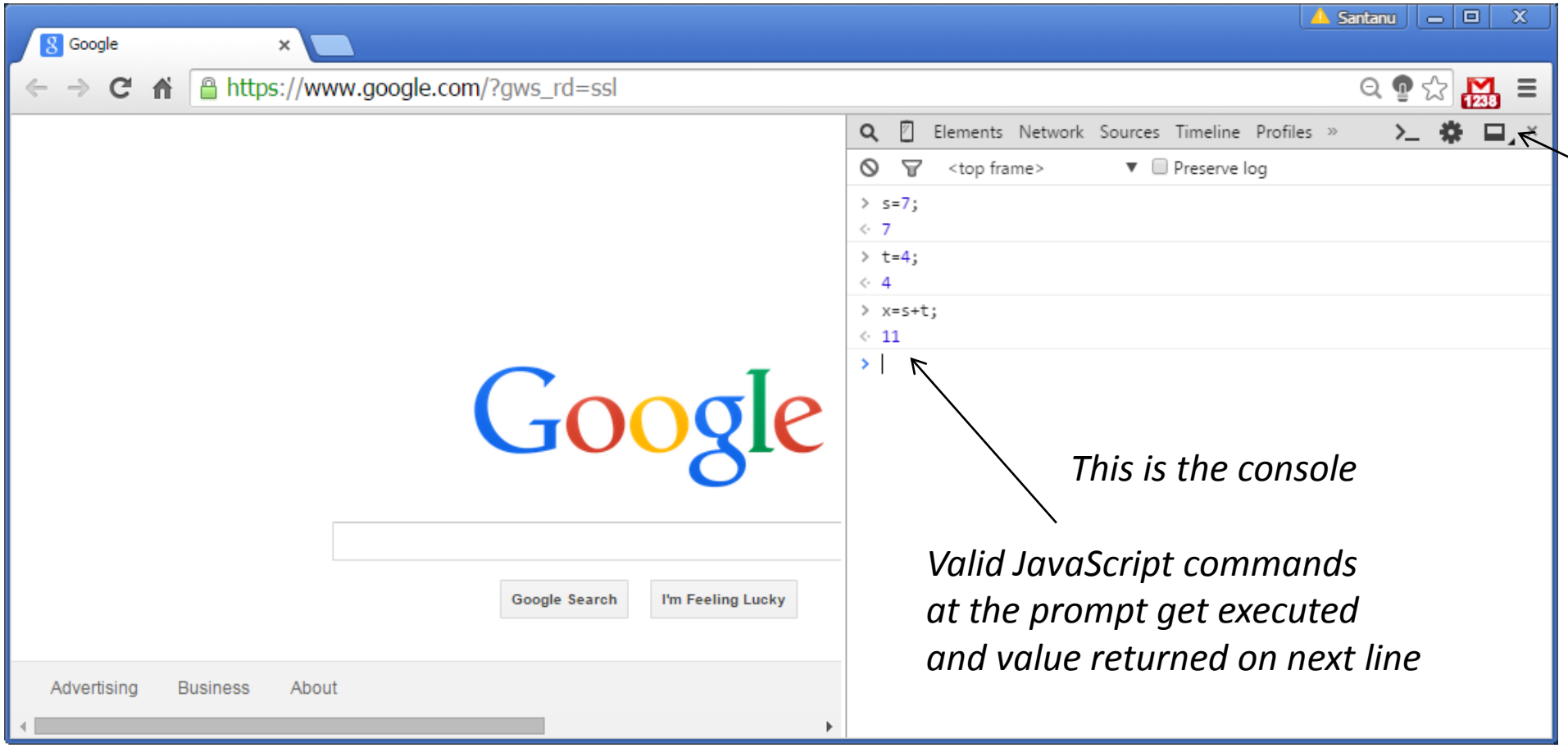
Embedding JavaScript in HTML

- Insert into an HTML page between <script> and </script> tags
- Can be put in <body> or <head> section
- The JavaScript code executes when the HTM page opens in a browser

```
<!DOCTYPE html>
<html>
<body>
.
.
<script>
JavaScript code goes here
(can be a file name with code)
</script>
.
.
</body>
</html>
```

Executing JavaScript in a console

- Control-Shift-J in Windows on Chrome browser brings up the console on the side or below



The first time you may need to long press the undock button for the  *icon to show*

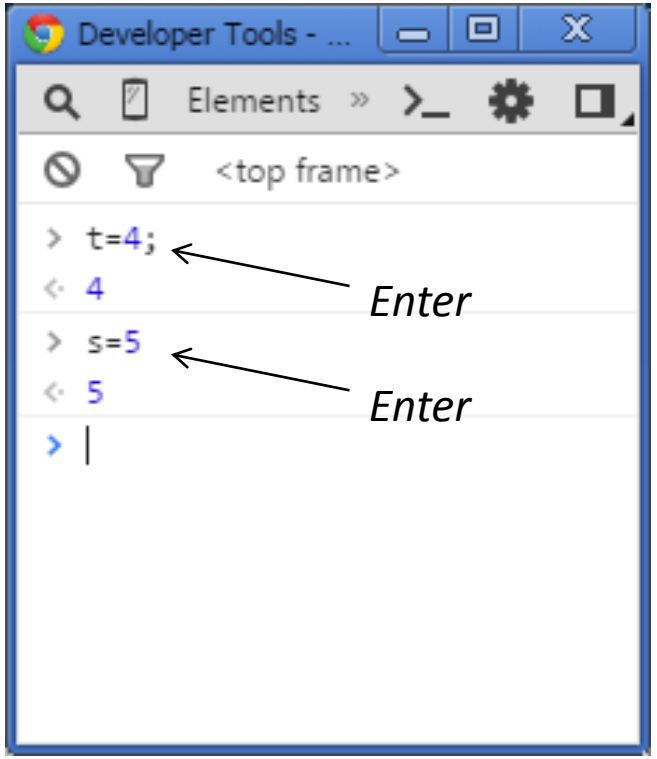
Clicking thereafter will dock console to or undock console from main browser window

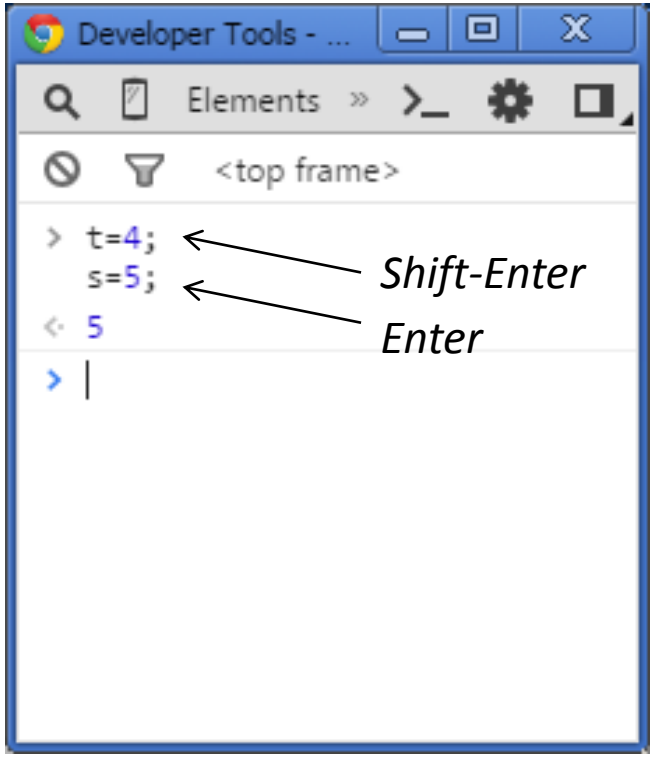
This is the console

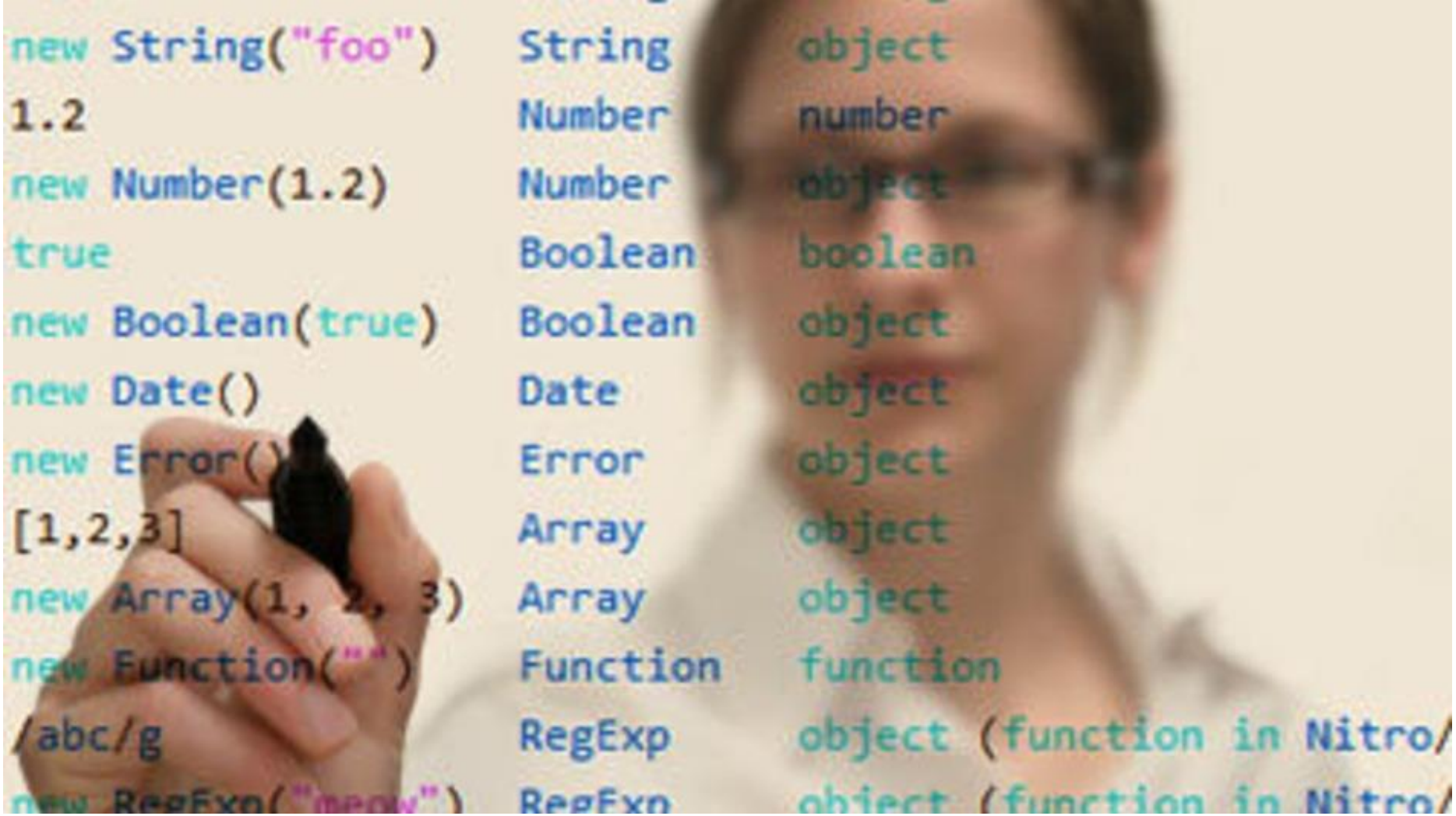
Valid JavaScript commands at the prompt get executed and value returned on next line

Writing multi-lines

- Use <Shift-Enter> to continue multiline without evaluating
- The final <Enter> will evaluate



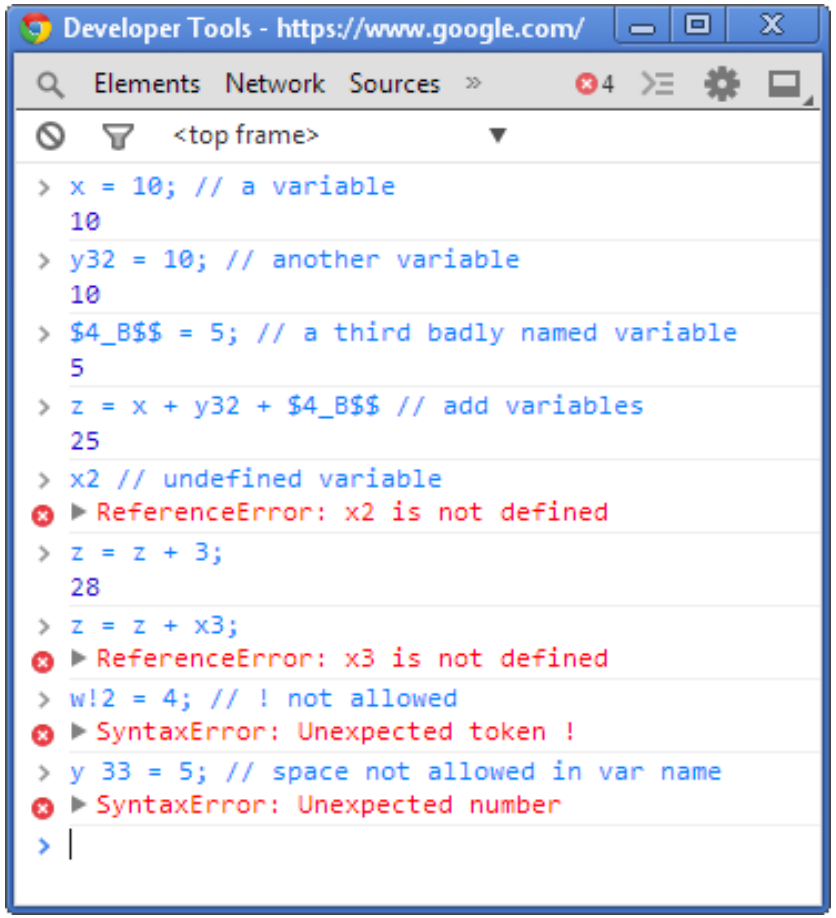




JavaScript programming basics

Variables

- Variables are symbolic names for values
- Names of variables are called **identifiers**
- No spaces or “!” in a name
- No digits in front of a name
- Declared in 2 main ways
  - With keyword *var*
    - Declares a local or global variable
  - Without keyword *var*
    - Declares a global variable



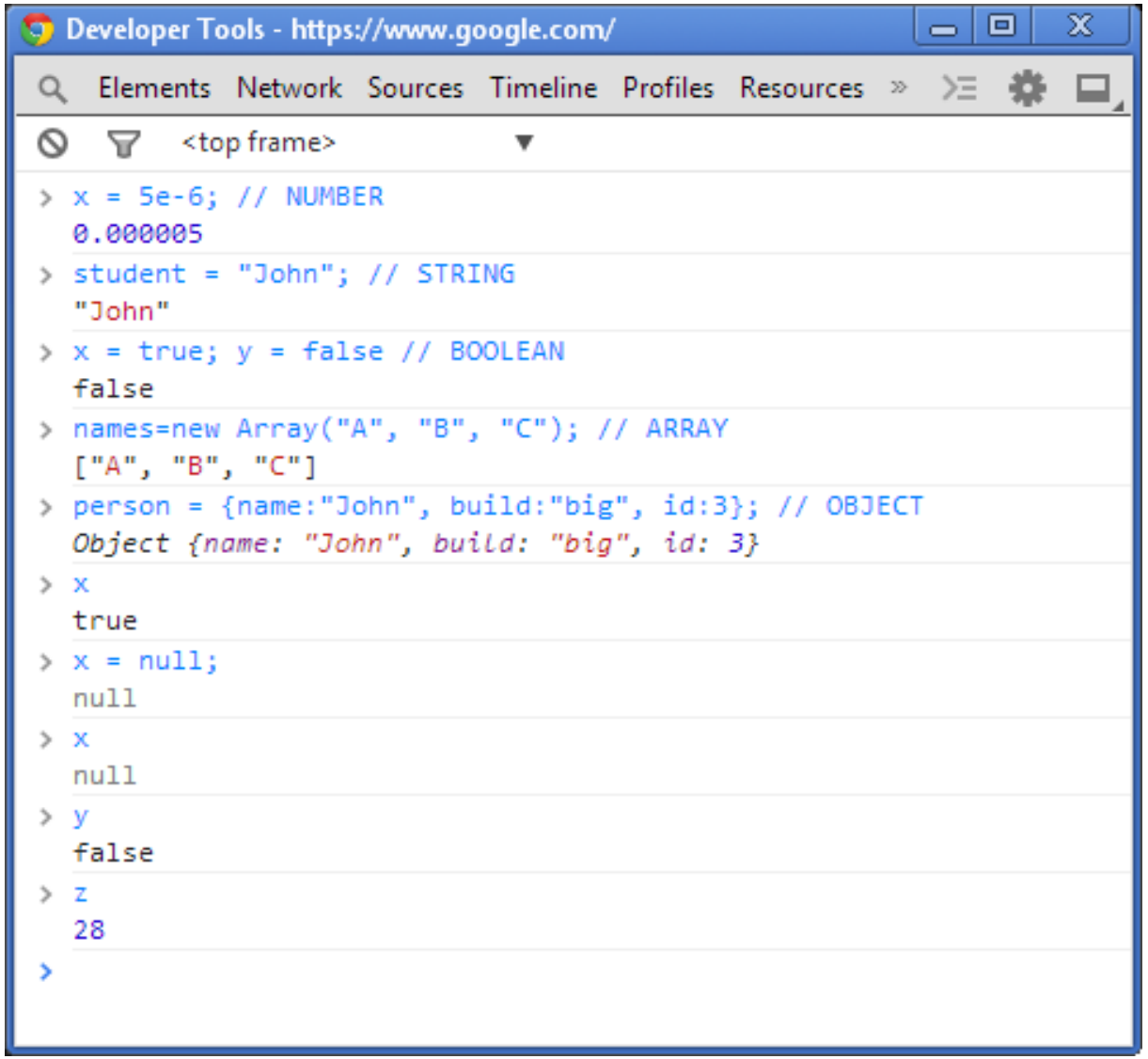
Note: A variable declared using the var statement with no initial value specified (e.g., var x;) has the value *undefined*. An attempt to access an undeclared variable results in a *ReferenceError*

Data types

- Number
- String
- Boolean
- Array
- Object
- Null
- Undefined

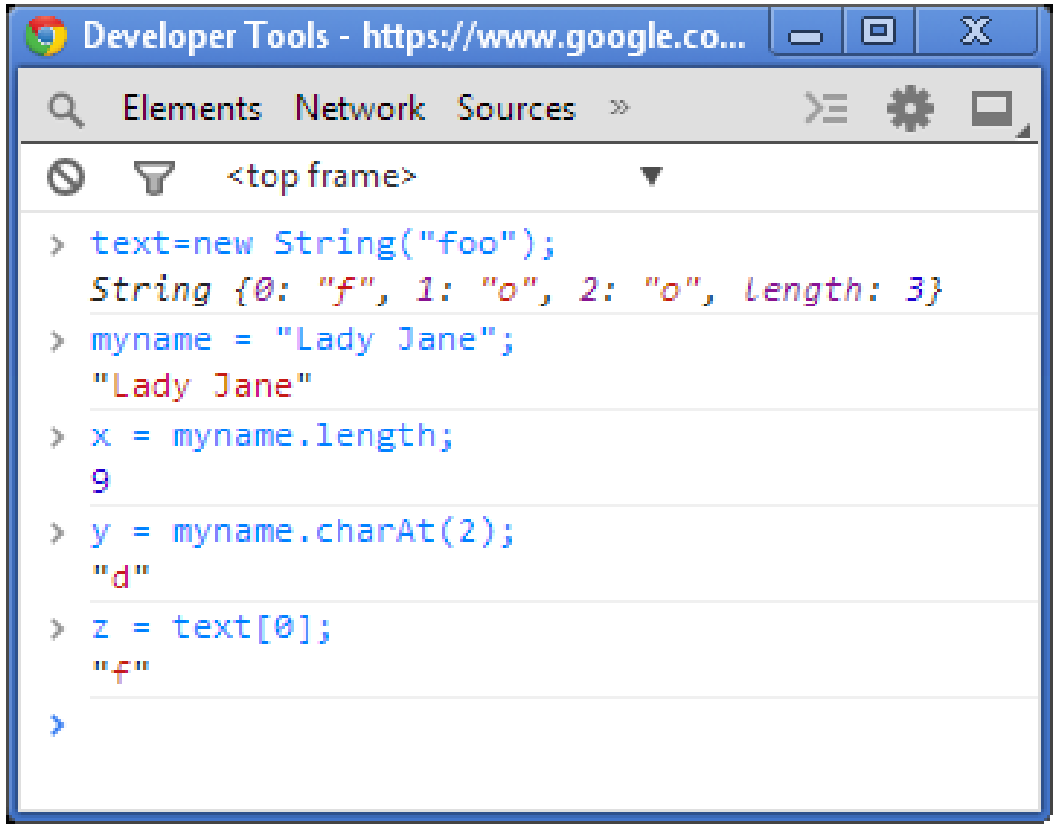
Dynamically typed

x = 5; x = “Sunnyvale”



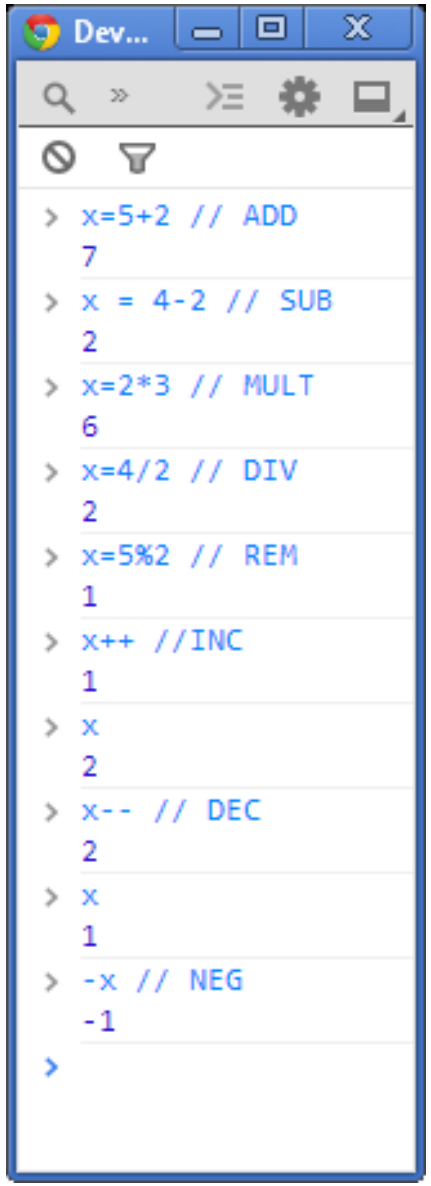
Strings

- Variables can store strings
- String objects are created with **new String()** or by assignment
- A string's properties can be obtained using methods
- Each character in a string has an index starting with 0
- There are many string object methods built into the language
  - length(); charAt(), concat(), indexOf(), match(), split(), substr(), search(), toLowercase(), toString(), trim(), valueOf(), replace(), slice(), ….

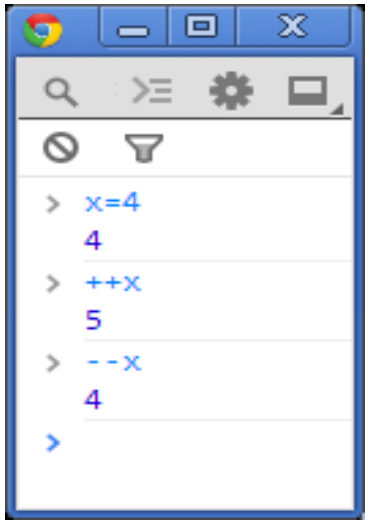


Arithmetic operators

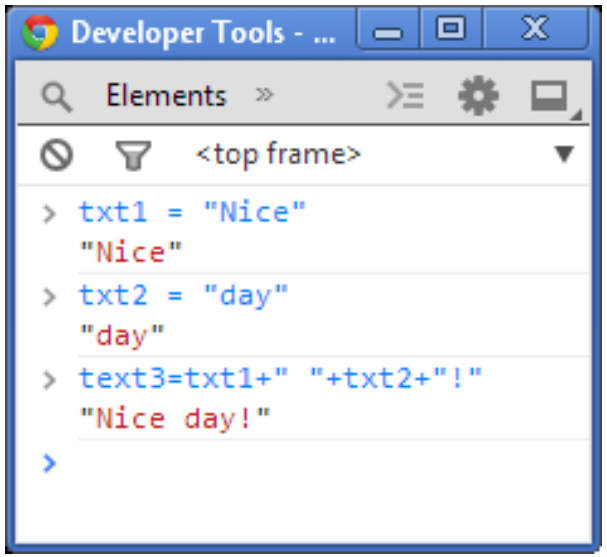
- = assign values
- Addition
- Subtraction
- Multiplication
- Division
- Remainder
- Increment
- Decrement
- Negation



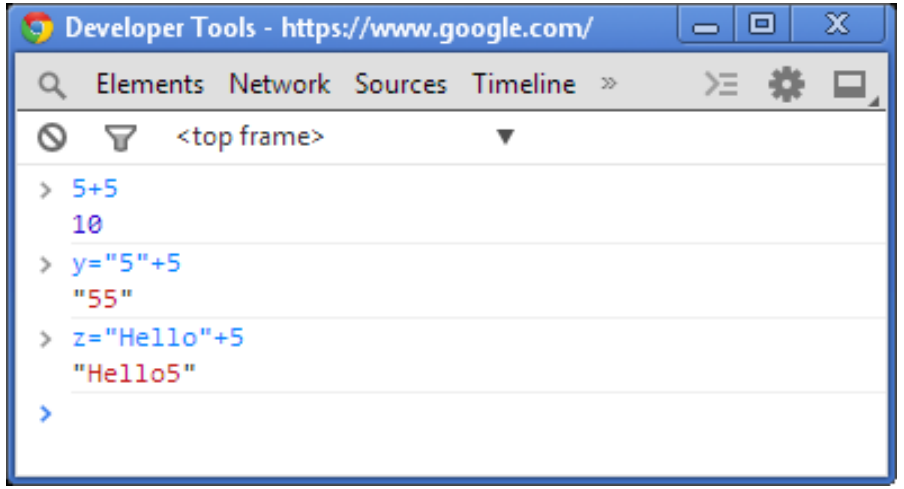
Pre-inc/dec



+ can operate on strings!

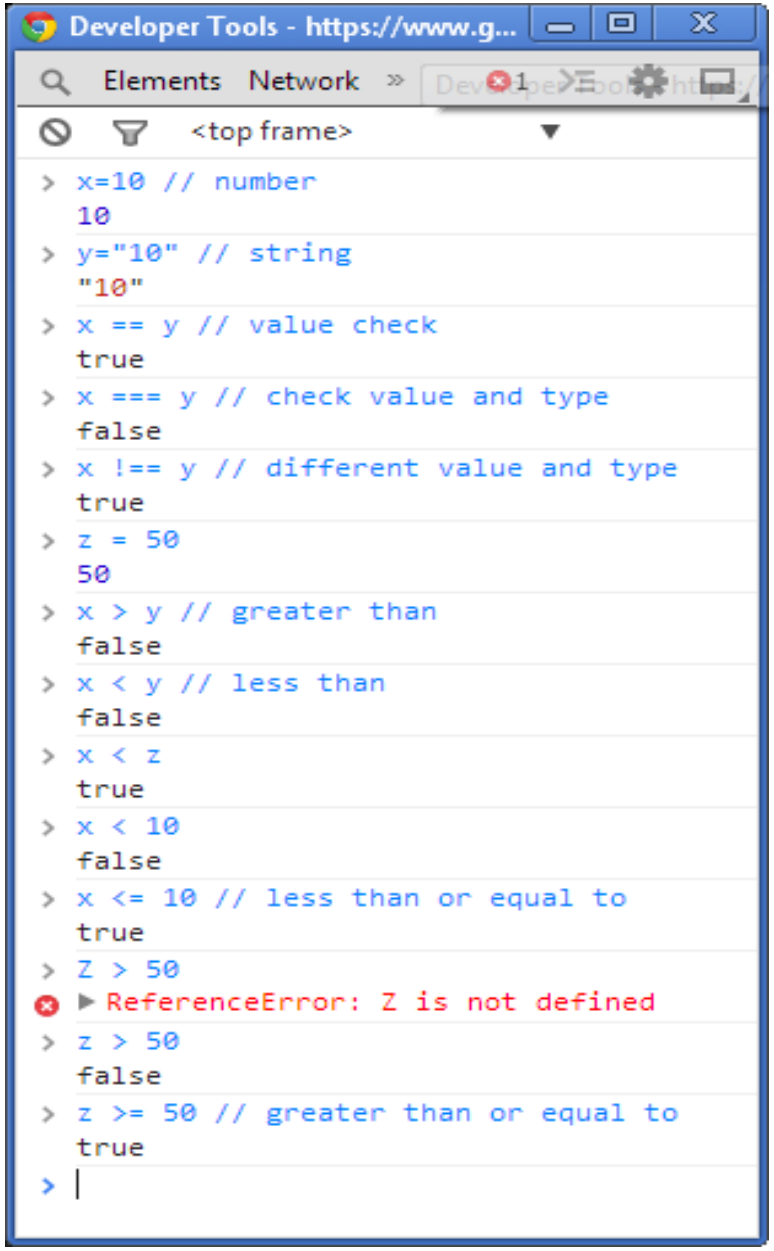


Numbers can be added to strings!



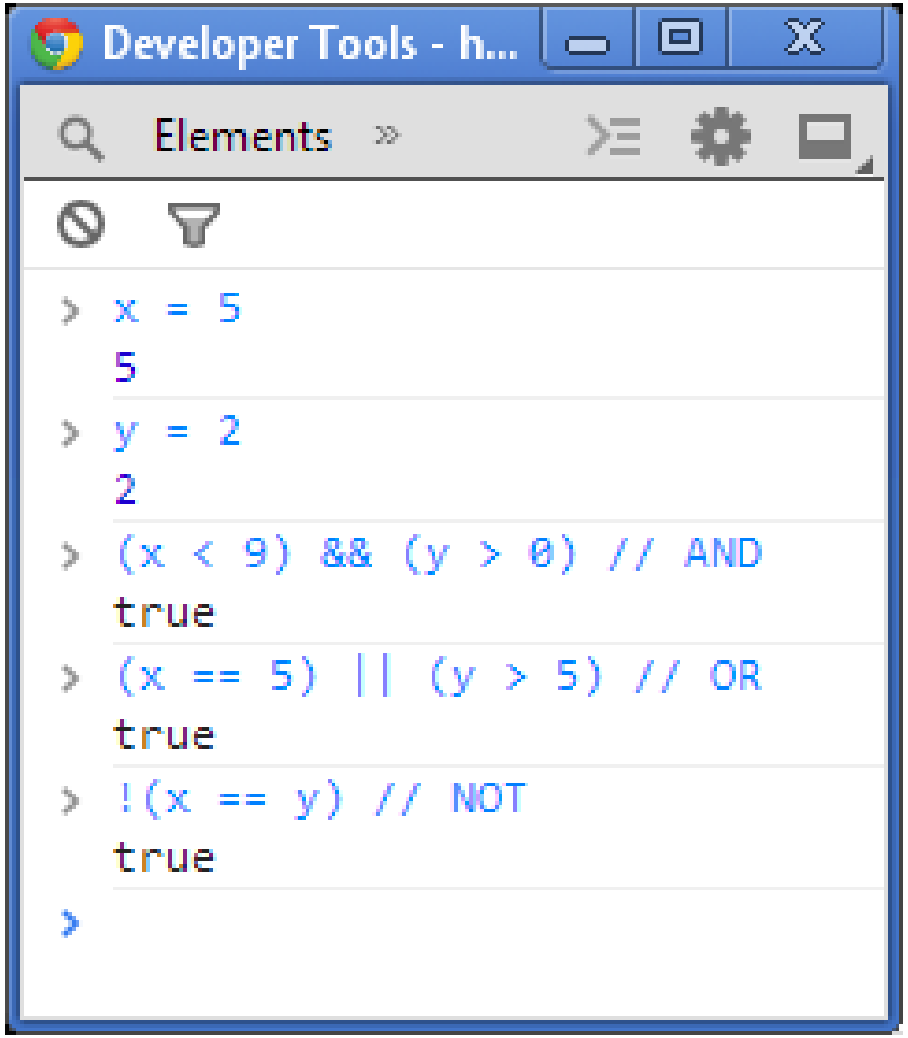
Comparison operators

- == : equal to
- === : equal in value and type
- != : not equal
- !== : different value and type
- > : greater than
- < : less then
- >= : greater than or equal
- <= : less than or equal



Logical operators

- && : and
- || : or
- ! : not



Bit operations

- Operands are converted to 32-bit integers and expressed in binary by a series of bits (zeros and ones)
- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on
- The operator is applied to each pair of bits, and the result is constructed bitwise
- Example for bitwise AND (0.0=0, 0.1=0, 1.0=0, 1.1=1)

- 9 (base 10) = 00000000000000000000000000001001 (base 2)
- 14 (base 10) = 00000000000000000000000000001110 (base 2)

- 14 & 9 (base 10) = 00000000000000000000000000001000 (base 2)
  = 8 (base 10)

AND		IN1	
		0	1
IN2	0	0	0
	1	0	1

Truth tables and logical operations

INV	IN	
	0	1
OUT	1	0

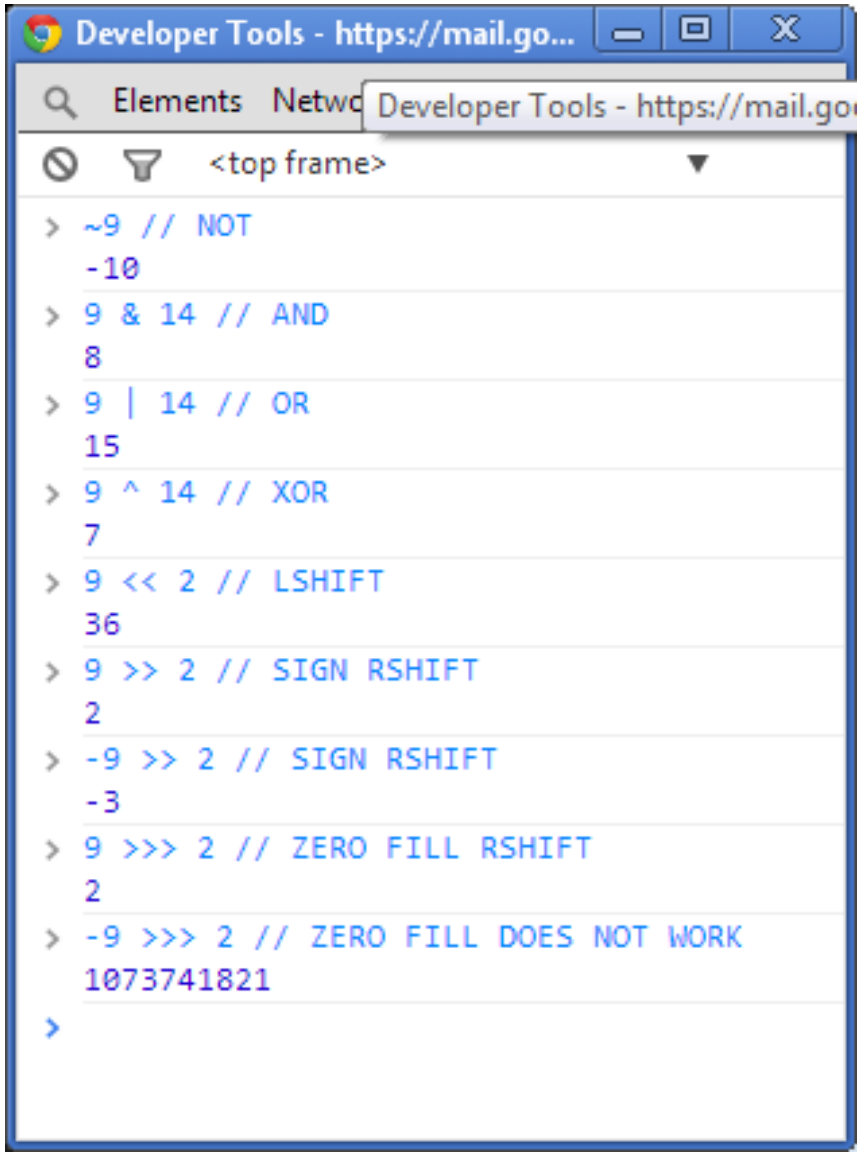
AND		IN1	
		0	1
IN2	0	0	0
	1	0	1

OR		IN1	
		0	1
IN2	0	0	1
	1	1	1

XOR		IN1	
		0	1
IN2	0	0	1
	1	1	0

- ~ : not
- & : and
- | : or
- ^ : xor
- << : left shift
- >> : sign propagate right shift
- >>> : zero propagate right shift

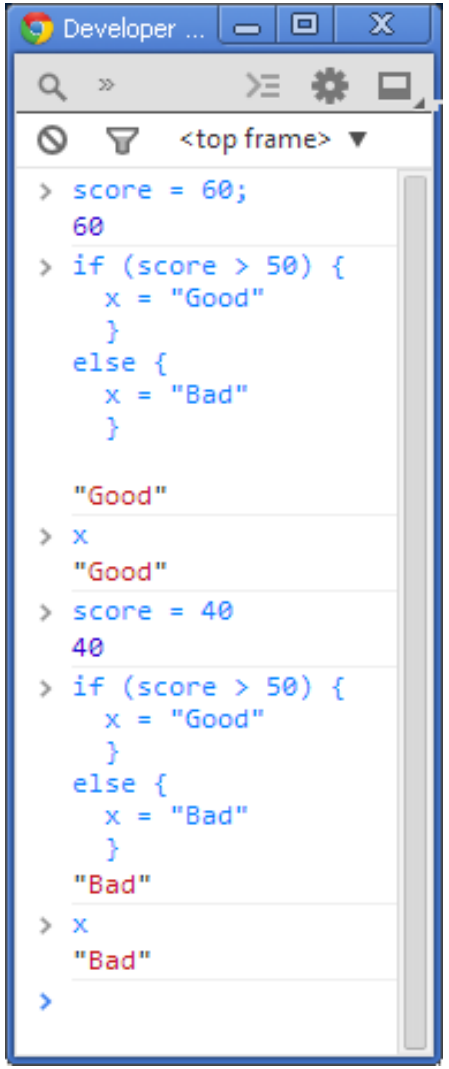
Note: ~x = -(x+1)



Conditional control: if-else

- A conditional statement is a set of commands that executes if a specified condition is true.

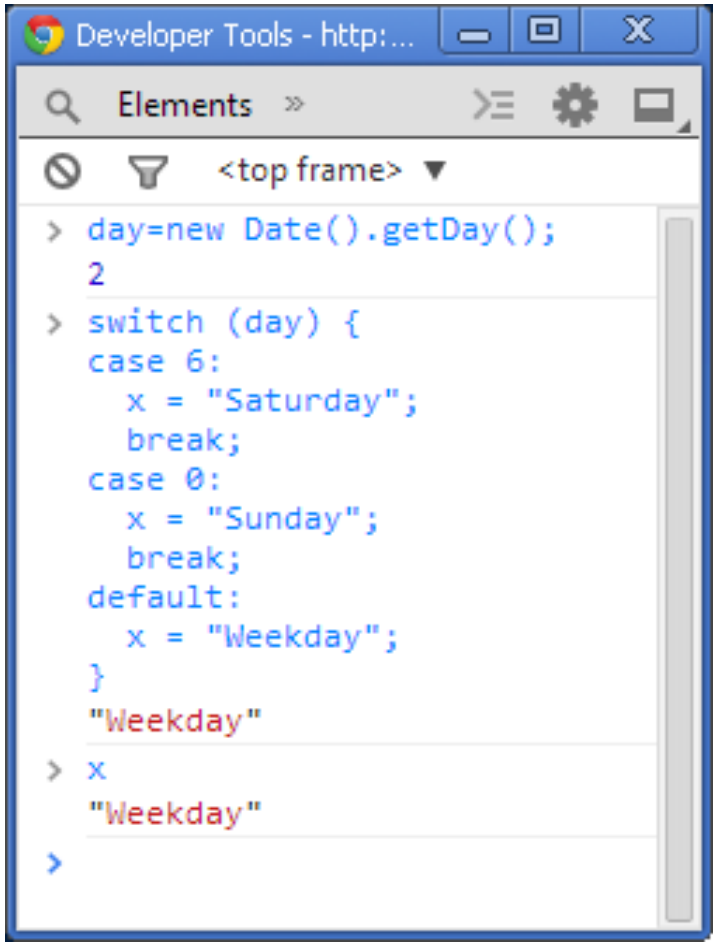
```
if (condition)
  {
  code to be executed if condition is
true
  }
else
  {
  code to be executed if condition is
not true
  }
```



Conditional control: switch

- Evaluates an expression, matching the expression's value to a case label, and executes statements associated with that case

```
switch(n)
{
case 1:
  execute code block 1
  break;
case 2:
  execute code block 2
  break;
default:
  code to be executed if n is different
from case 1 and 2
}
```



Loops: for

- A *for* loop repeats until a specified condition evaluates to false

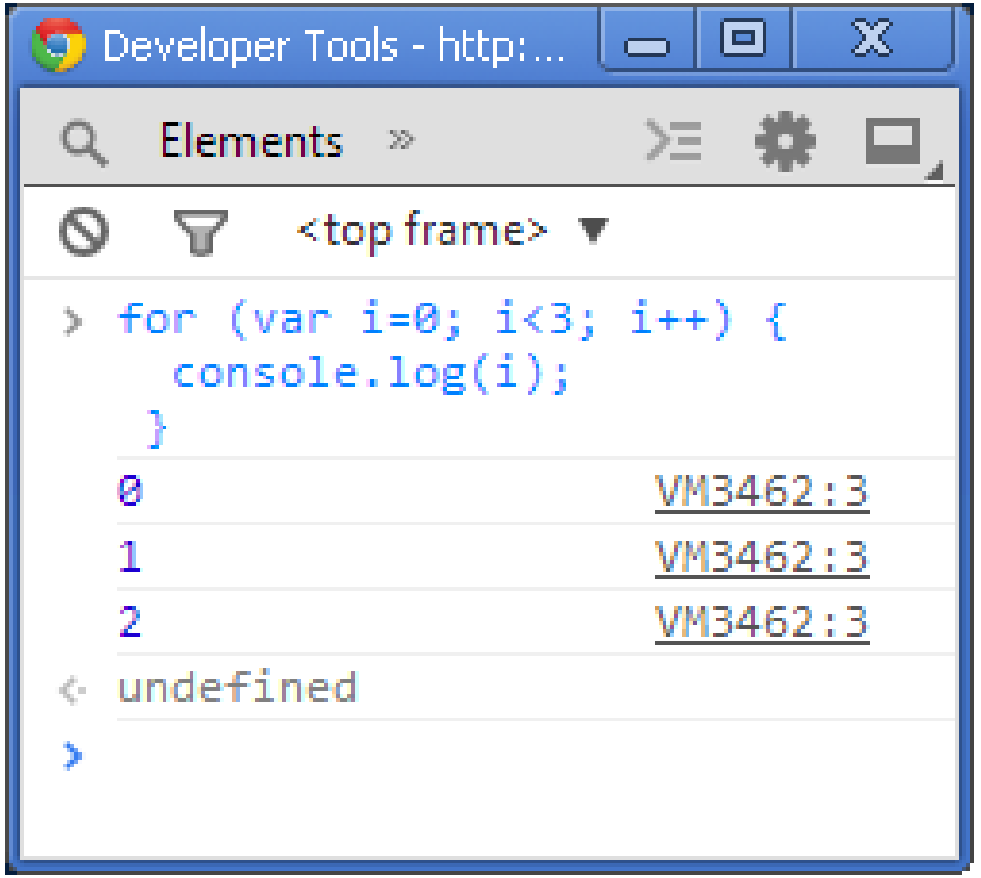
```
for (init counter; test counter; increment counter)
 {
 code to be executed;
 }
```

Parameters:

init counter: Initialize the loop counter value

test counter: Evaluate for each loop iteration. If it evaluates to TRUE, the loop continues. If it evaluates to FALSE, the loop ends

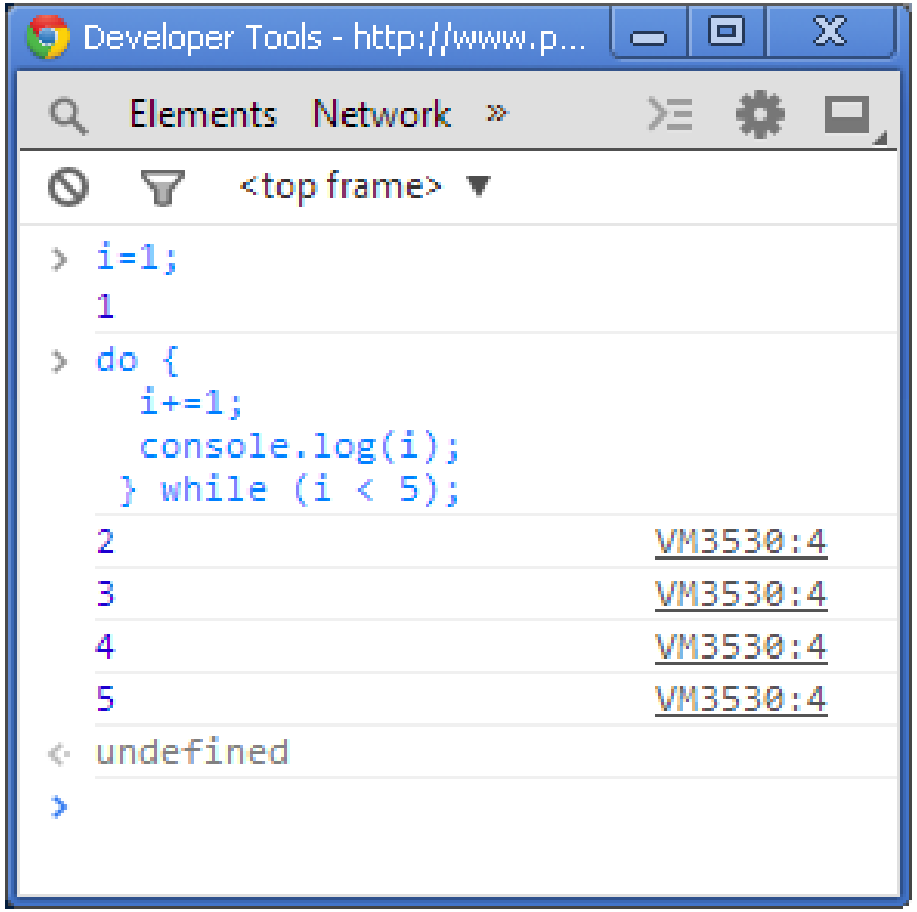
increment counter: Increases the loop counter value



Loops: do-while

- The *do...while* statement repeats until a specified condition evaluates to false

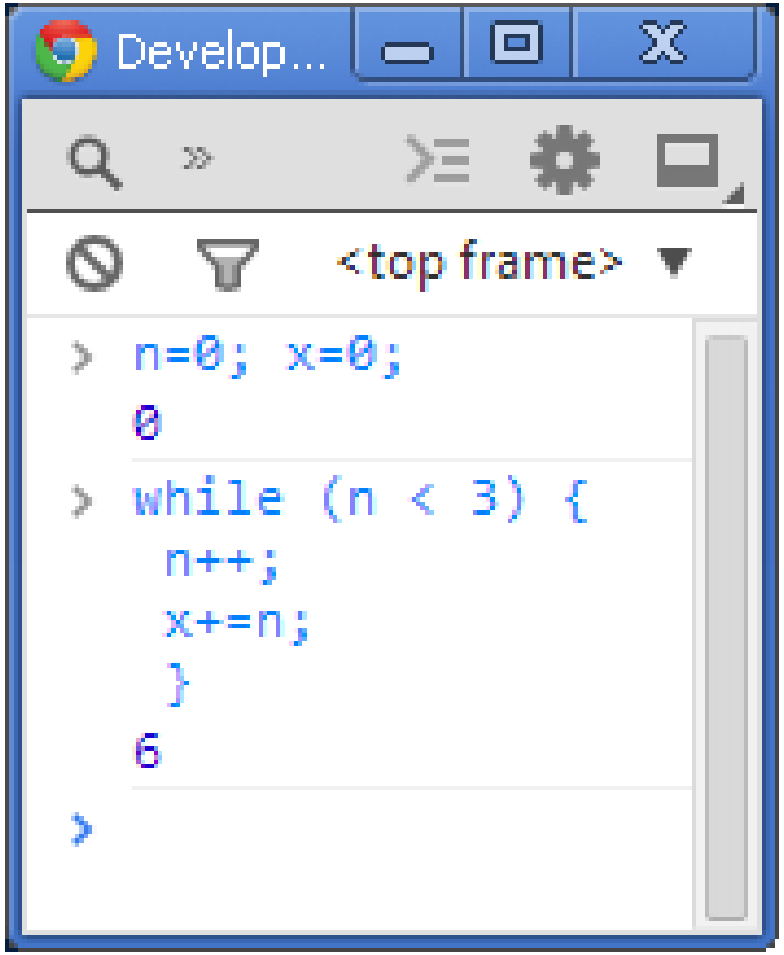
```
do {
  code block to be executed
  } while (condition);
```



Loops: while

- A *while* statement executes its statements as long as a specified condition evaluates to true

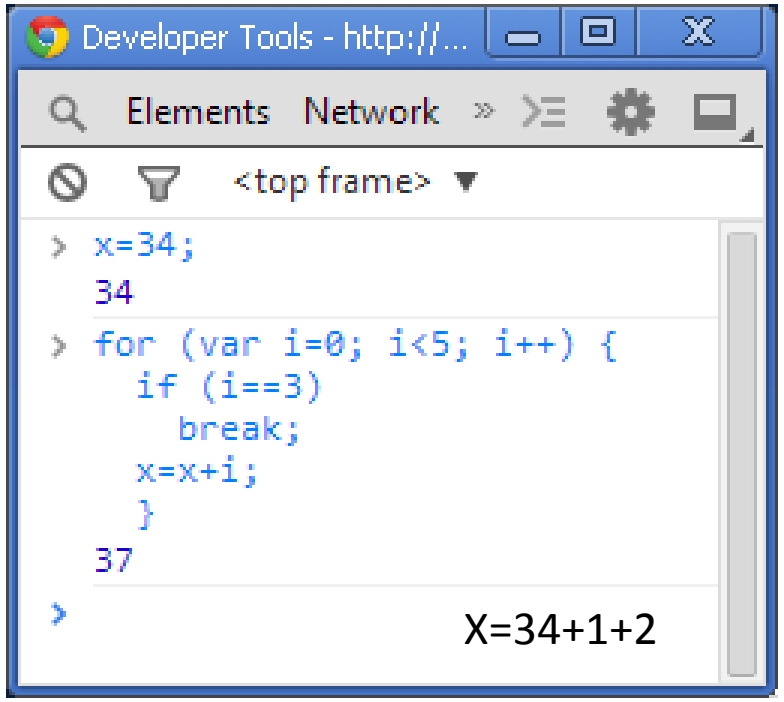
```
while (condition) {
  code block to be executed
  }
```



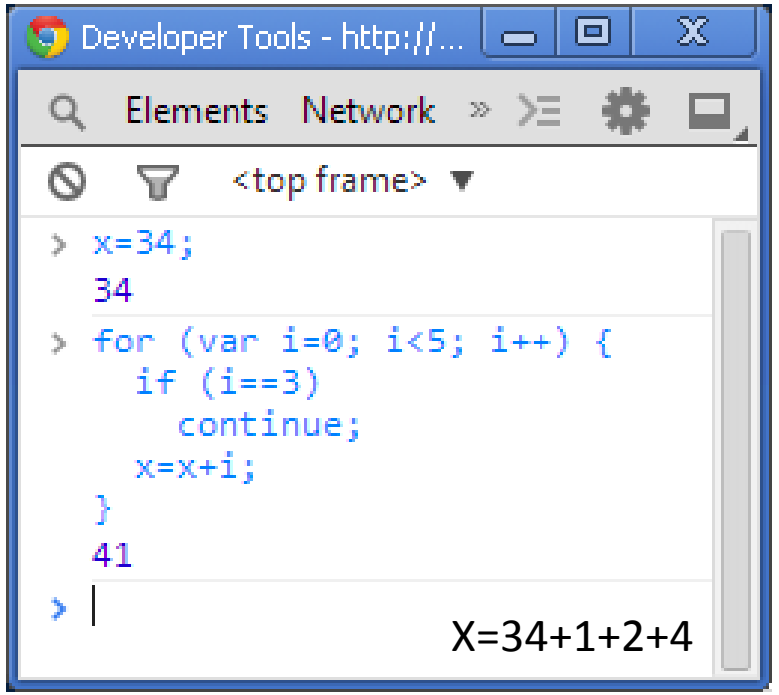
Break & Continue

- The *break* statement "jumps out" of a loop
- The *continue* statement "jumps over" one iteration in the loop

Breaks out of the loop when i=3

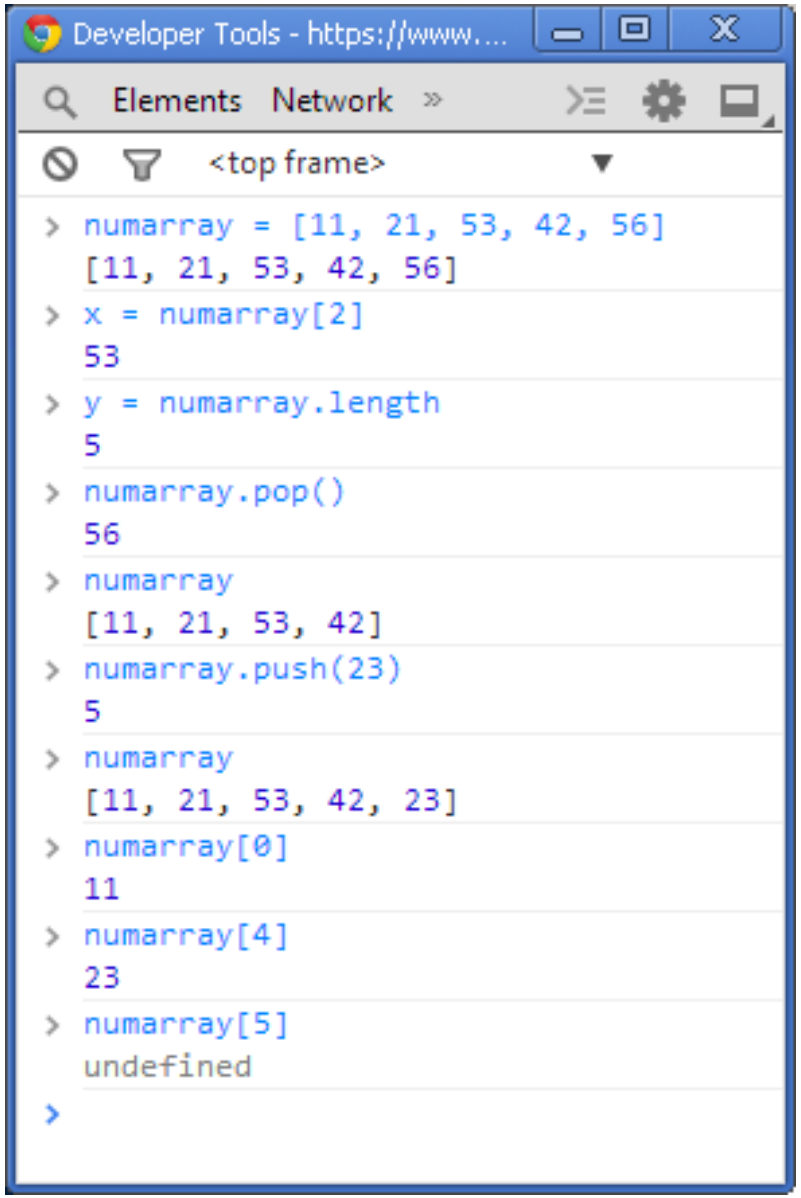


Skips iteration loop when i=3



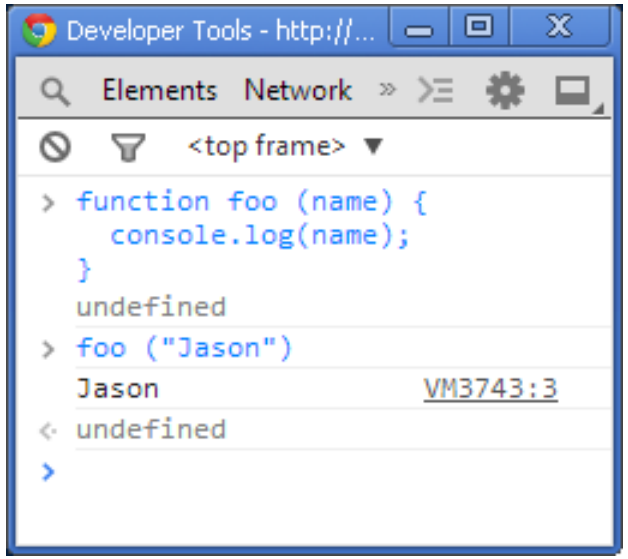
Arrays

- An array is a data structure
- Each cell in an array has an automatic index starting with 0
- pop() function deletes the last item and gets its value
- push() function adds a last item
- Array of arrays are also possible



Functions

- Block of code that gets executed when “called”
- Arguments can be passed to a function
- function foo (var1) {
  code to do something
  }

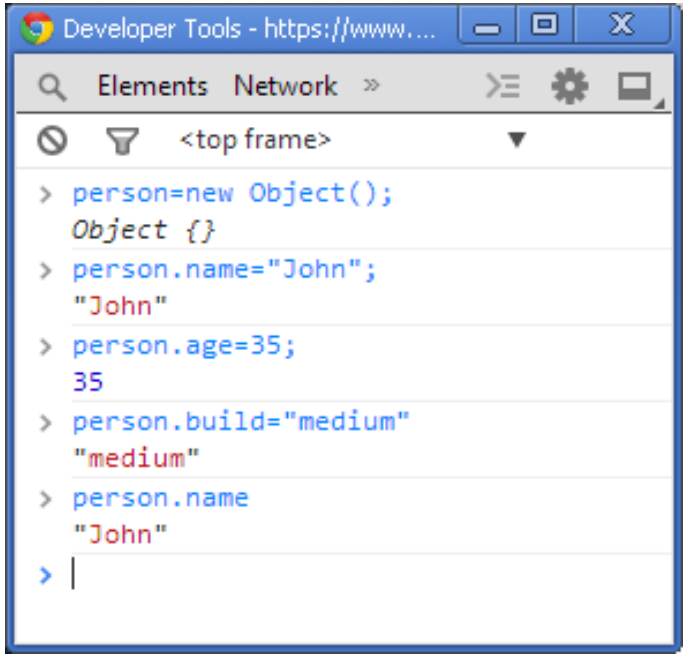


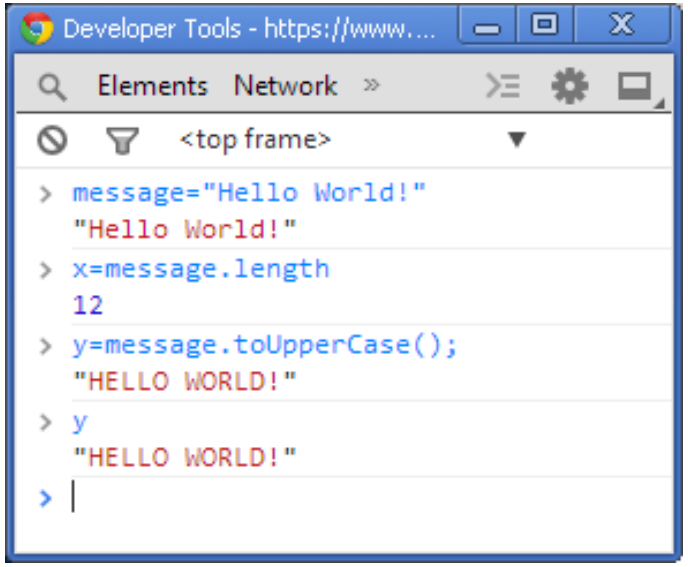

```
<!DOCTYPE html>
<html>
<head>
<title>Trying a JavaScript function </title>
<script>
function foo (name)
{
alert("Hello “+ name + “!”);
}
</script>
</head>

<h1> This is an example </h1>
<p> Click button to call function with an argument</p>
<body>
<button onclick=“foo(‘Professor’)">Click me</button>
</body>
</html>
```

Objects

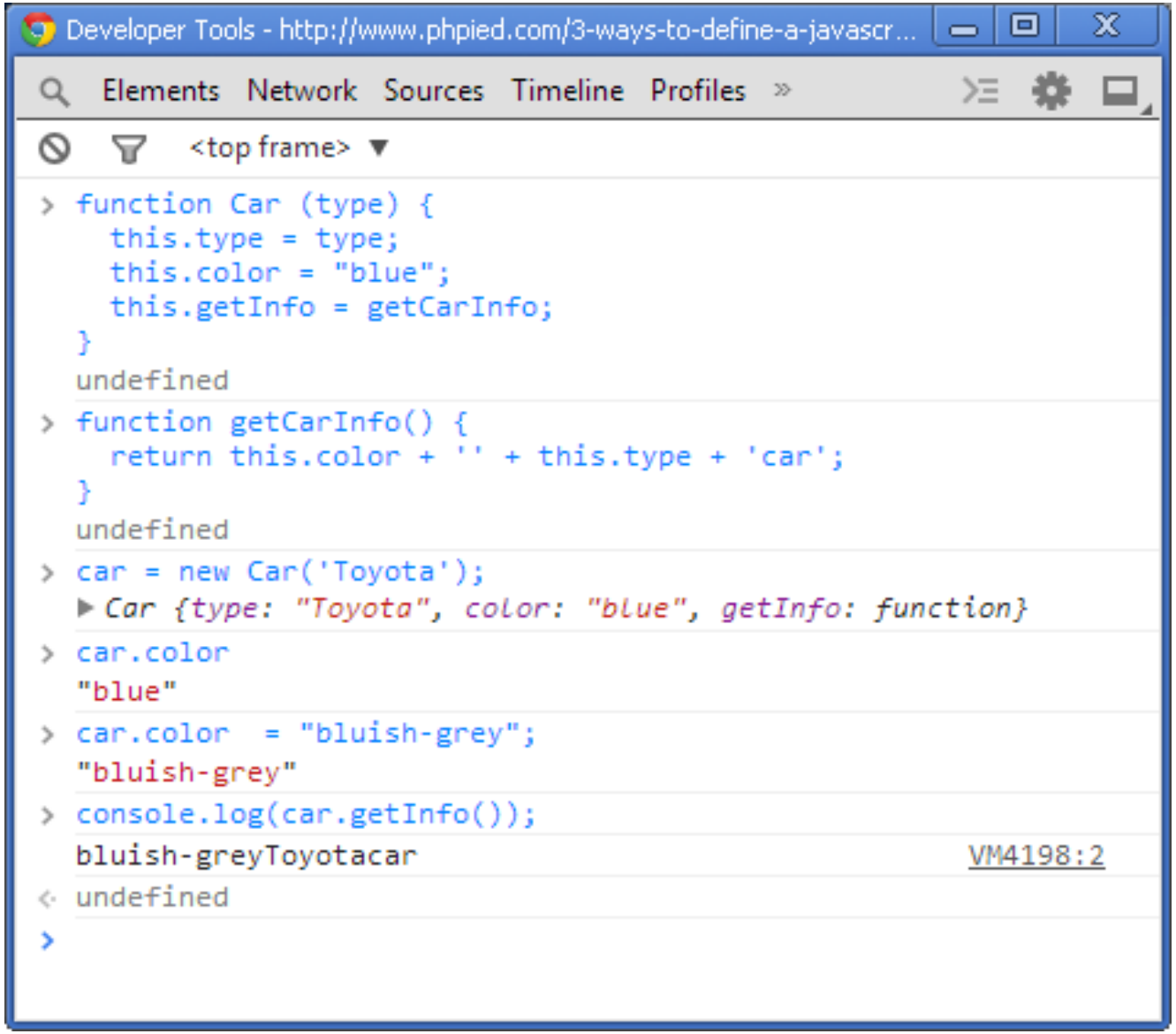
- Objects are just data, with properties and methods
- Strings, Dates, Arrays, and Functions can be objects
- Can create own object, *e.g.*, person, with properties
- Properties are accessed easily
- Methods are actions that objects can perform
  - Syntax: *objectName.methodName()*





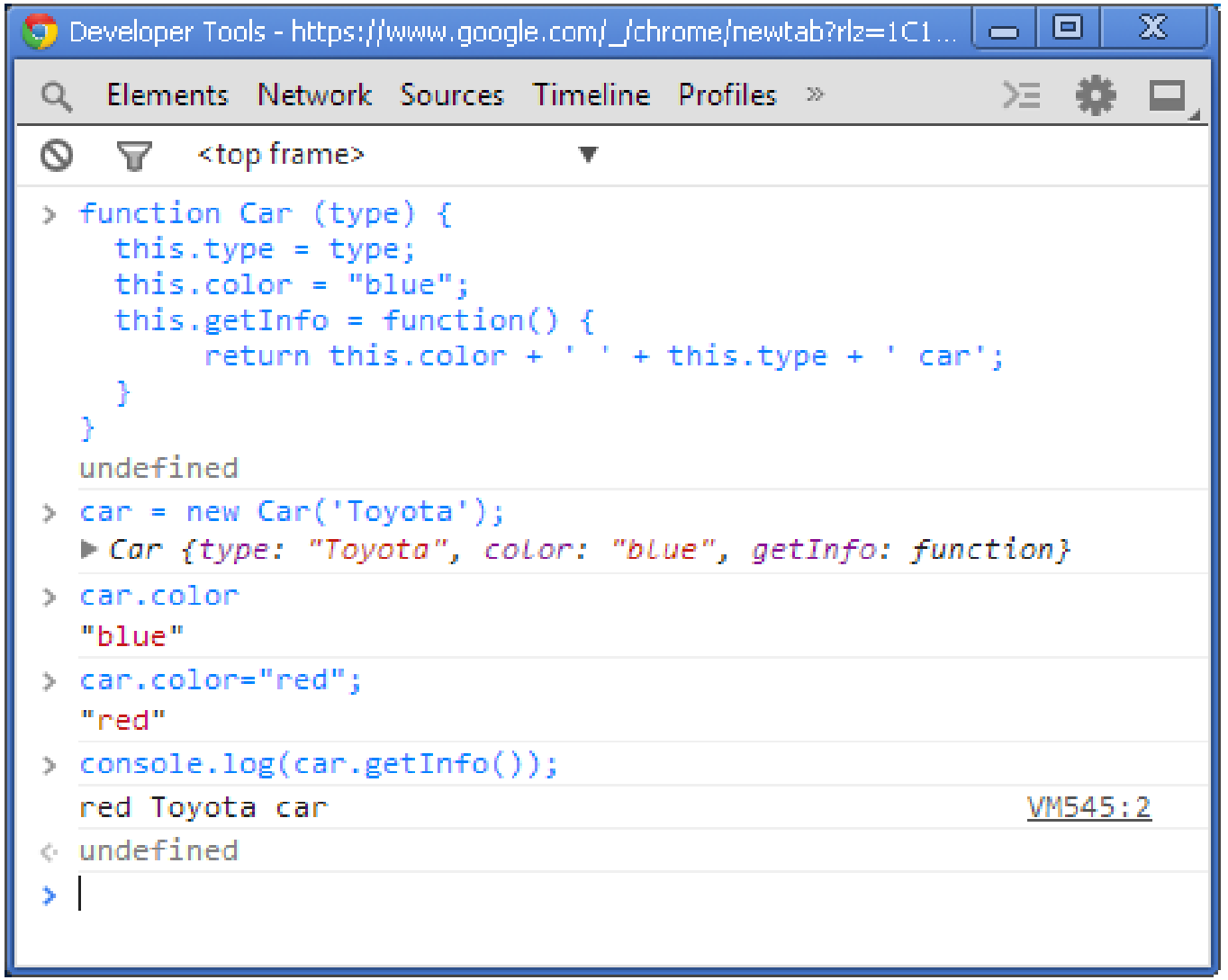
JavaScript object - Trick 1

- Define a class using constructor *function()* Create object properties with *this* keyword
- Create objects with *new* keyword that calls the constructor
- Instantiate object calling *constructor* function
- Set properties and call methods



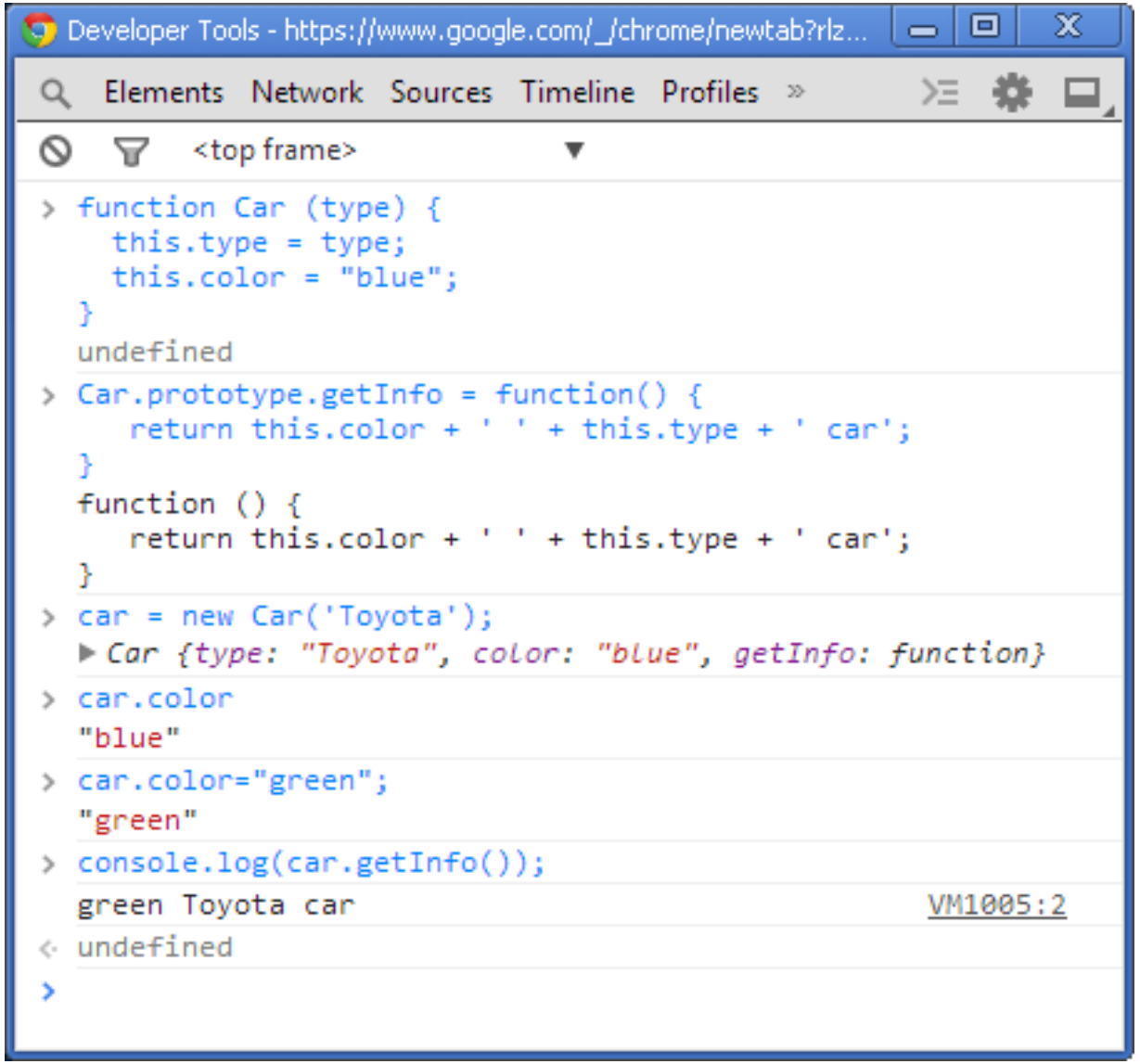
JavaScript object - Trick 2

- Methods can be defined within constructor function
- Helps prevent pollution of global namespace, but method gets recreated every time object created



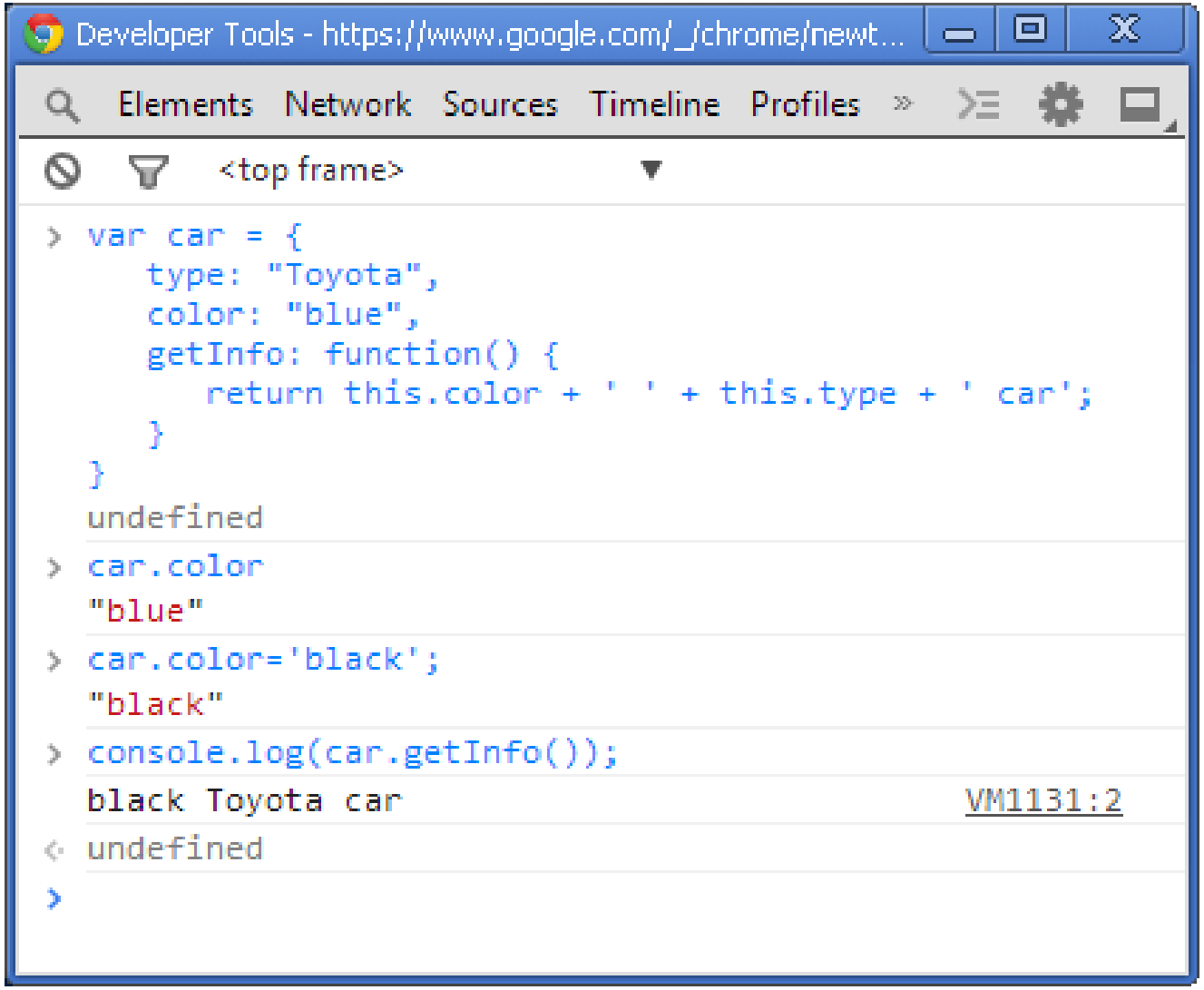
JavaScript object - Trick 3

- Prevent recreation of method *getInfo()* every time new object created
- Add to *prototype* of constructor function



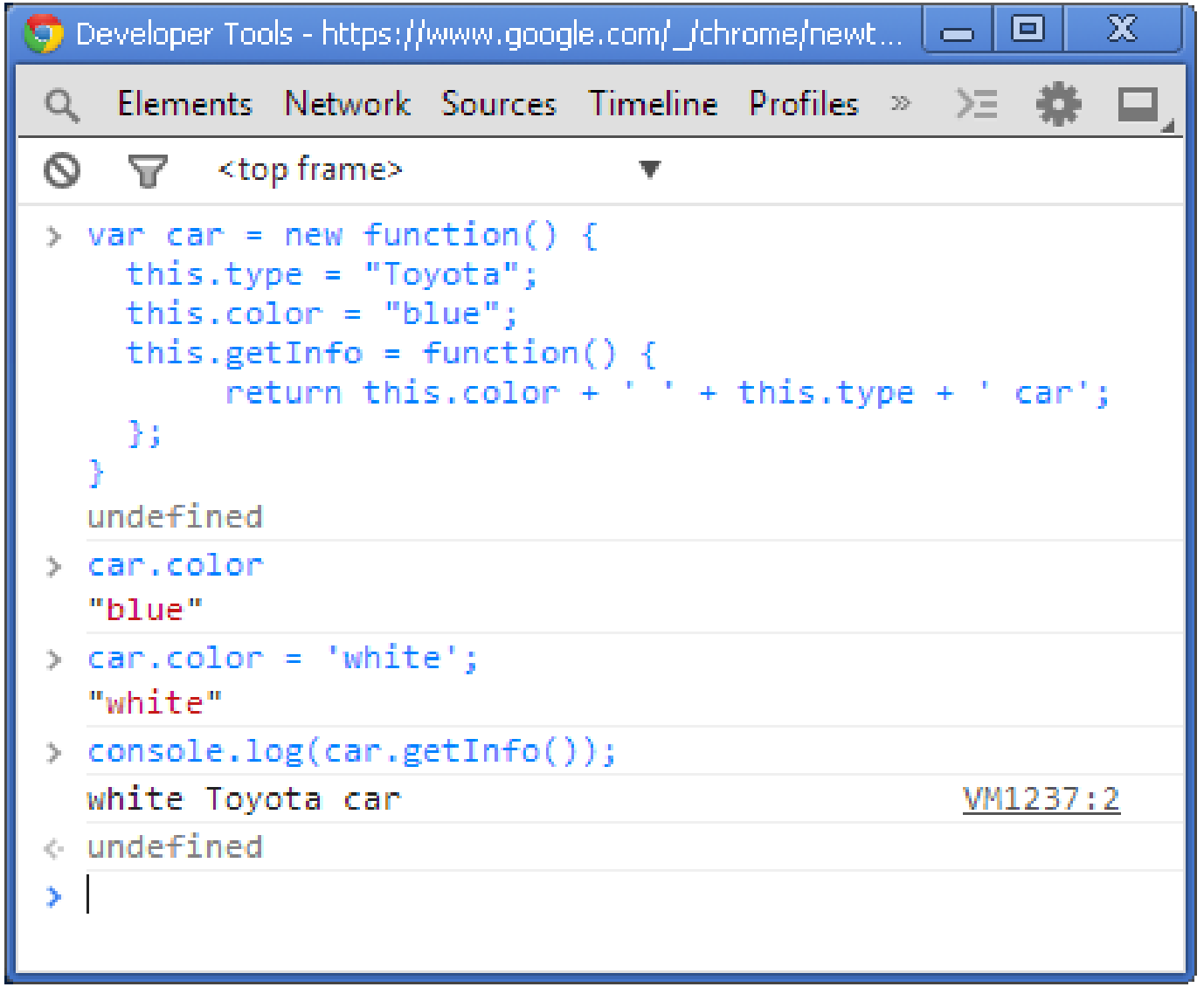
JavaScript object - Trick 4

- Object *literals* are shorter way to define objects and arrays
- No need to (and cannot) create instance, it already exists
- Start using



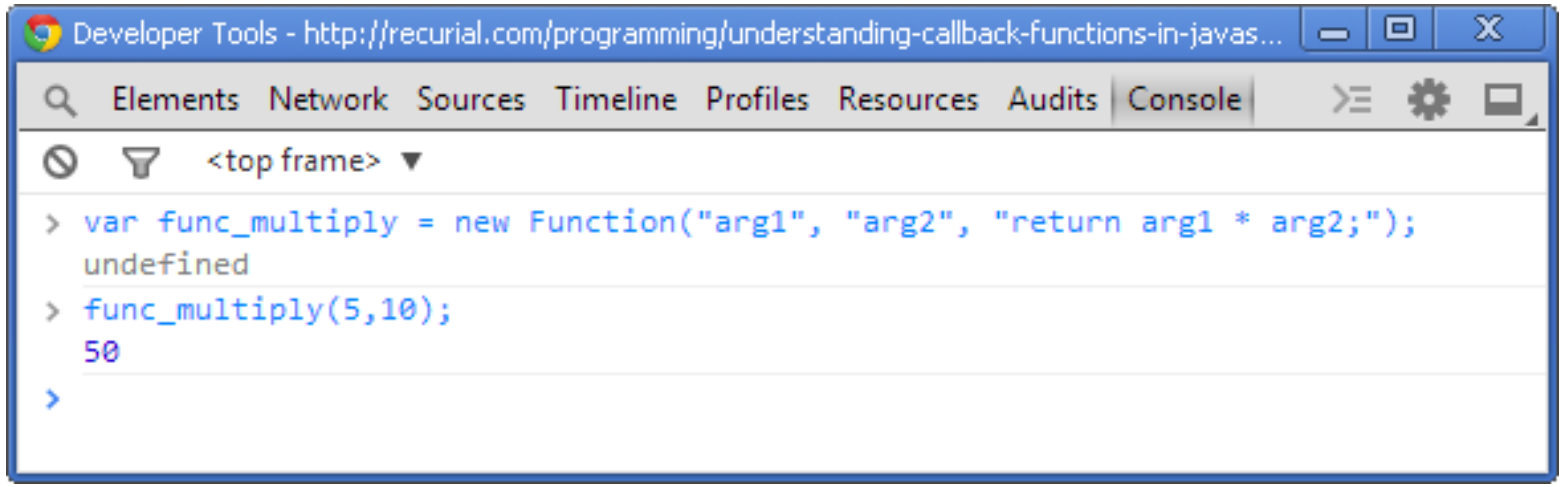
JavaScript object - Trick 5

- Use singleton for a constructor function with no name that will be used few times
- new function(){...} defines an anonymous constructor function and invokes it with *new*



Objective look at functions

- JavaScript functions are *function objects* created with the *Function* constructor
- A Function object contains a string which contains the Javascript code of the function!
- Can pass code to another function in the same way you would pass a regular variable or object



So many *functions*!

- function A(){}; // function declaration
- var B = function(){}; // function expression
- var C = (function(){}); // function expression with grouping operators
- var D = function foo(){}; // named function expression
- var E = (function(){
  return function(){} //(IIFE) that returns a function
  })();
- var F = new Function(); // Function constructor
- var G = new function(){}; // special case: object constructor

Type name conventions

Standard function statement:

```
function getarea(w,h){
  var area=w*h
  return area
}
```

Conditional function statement:

```
if (calculateit == true){
   function getarea(w,h){ //standard function
   var area=w*h
   return area
   }
}
```

Function Literal (an anonymous function assigned to a variable):

```
var getarea=function(w,h) {
  var area=w*h
  return area
  }
```

Function Constructor … creates a function on the fly, which is slower and generally discouraged:

```
  var getarea=new Function("w", "h", "var area=w*h; return area")
```

Function subtleties


```
function myFirstFunc(param) {
    //Do something
};
```

Defines the function as soon as the enclosing scope is entered

---

```
var myFirstFunc = function(param) {
    //Do something
};
```

Only creates the function once execution reaches that line

Function.call

Function.apply

Callbacks(1)

- DOM and JavaScript are not for good multithreading
- JavaScript is dynamically typed. Since almost everything can be modified and redefined at run time, what happens to when a programmer modifies the body of a method while another thread is actually calling that method? The only safe way to handle this would be to synchronize any operations modifying the any method on any object -- since this a fairly common task in JavaScript, you would essentially revert it back to a single-threaded language again. As a result, browsers do not give access to the threading model and provide a single thread for everything accessing the user interface (i.e. the DOM). This means that all the application logic accessing and modifying the user interface elements is always in the same thread.

Inheritence

Closure

Polymorphism

Error

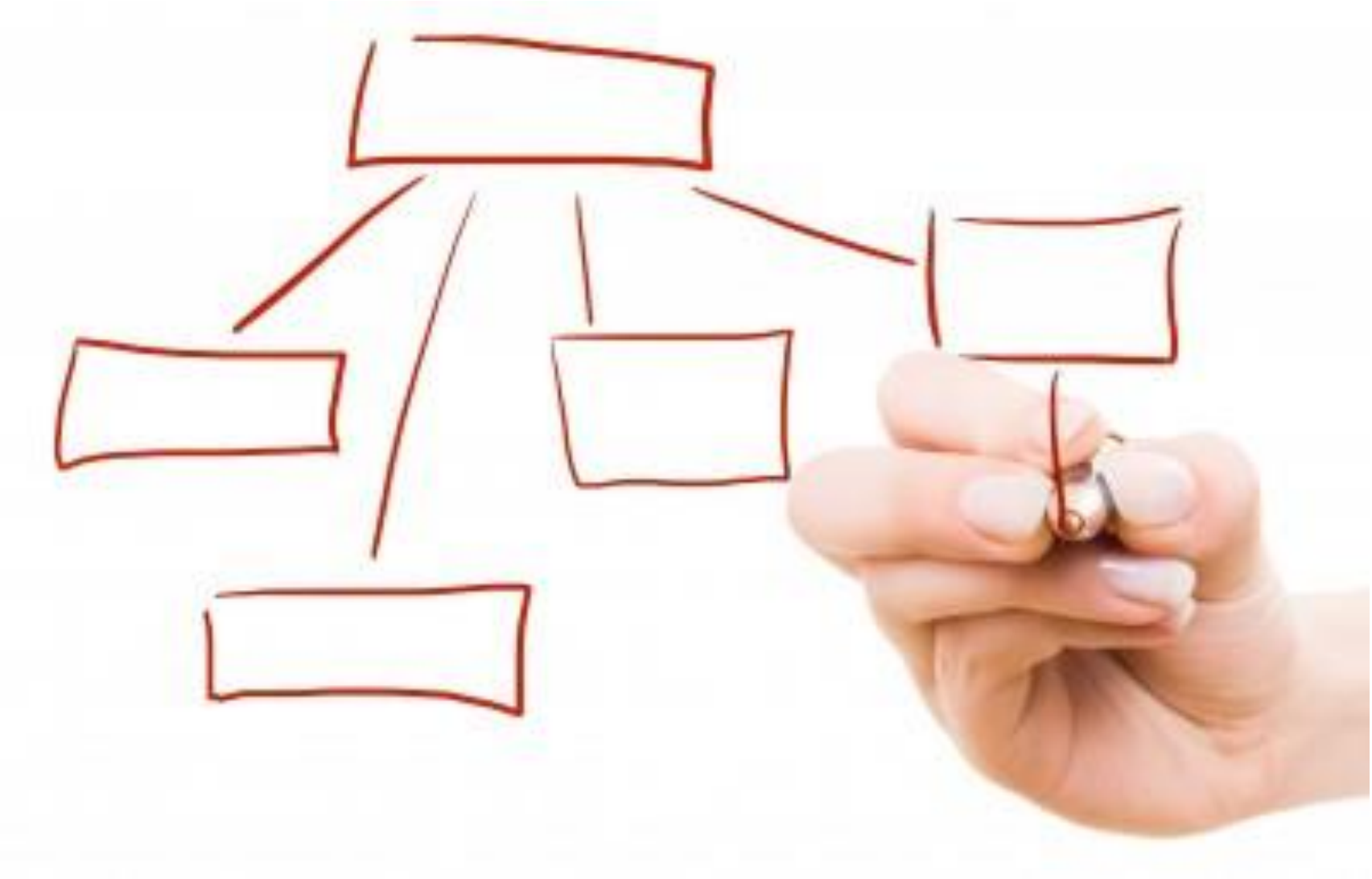
Dialogs

- Alert (message box) dialog
- Confirm dialog
- Print dialog
- Find dialog
- Add Favorite dialog
- Prompt

Validation

Regular expressions

Document Object Module (DOM)



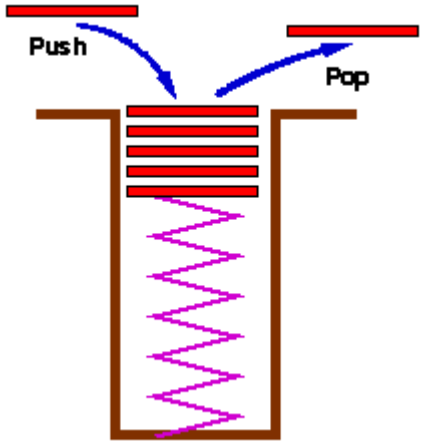
Data structures

What are data structures?

- A particular way of storing and organizing data in a computer so that it can be used efficiently, *e.g.* arrays, lists, *etc*.
- Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks
  - B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers
- Efficient data structures are a key to designing efficient algorithms
- Implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure
- JavaScript features inbuilt support for *Array*, *String*, and *Object* that can be used to implement various useful data structures such as stacks, linked lists, *etc*.

Stack

- A data structure consisting of a collection of elements in which the operations allowed on the collection are the addition of an element, known as *push*, and removal of an element, known as *pop*
- The relation between the push and pop operations is such that the stack is a Last-In-First-Out (LIFO) structure --- the last element added to the structure must be the first one to be removed
- A stack can be implemented using an array or a linked list
- A common model of a stack is an element stacker, where elements are "pushed" onto to the top and "popped" off the top



Stack implementation

Developer Tools - http://www.i-programmer.info/programming/...

Elements Network Sources Timeline Profiles Resources »

<top frame>

```
> function Stack() {
    this.stac = new Array();
    this.pop=function() {
              uval = this.stac.pop();
              if (uval == undefined)
                 console.log("Trying to pop an empty stack");
              return uval;
    }
    this.push=function(item) {
              this.stac.push(item);
    }
  }
  undefined
>
```

Developer Tools - http://www.i-program...

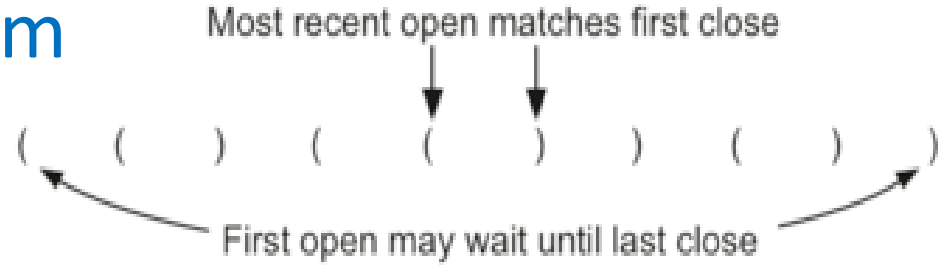
Elements Network Sources »

<top frame>

```
> var stack=new Stack();
  undefined
> stack.push("A");
  stack.push("B");
  stack.push("C");
  undefined
> stack.pop();
  "C"
> stack.pop();
  "B"
> stack.pop();
  "A"
> stack.pop();
  Trying to pop an empty stack      VM10481:7
<· undefined
>
```

Problem solving with a stack

- Method to read a string of parentheses and decide whether the symbols are balanced is important for programming language structures that use parentheses or curly brackets
  - As we process symbols from left to right, the most recent opening parenthesis must match the next closing symbol
  - The first opening symbol processed may have to wait until the very last symbol for its match
  - Closing symbols match opening symbols in the reverse order of their appearance; they match from the inside out
- Stacks can be used to solve the problem



Parentheses matching – the idea

- Starting with an empty stack, process the parenthesis strings from left to right on symbol at a time
- If a symbol is
  - an opening parenthesis: *push* it on the stack as a signal that a corresponding closing symbol needs to appear later for a match
  - a closing parenthesis: *pop* the stack
- As long as it is possible to pop the stack to match every closing symbol, the parentheses remain balanced
- If at any time there is no opening symbol on the stack to match a closing symbol, the string is not balanced properly
- At the end of the string, when all symbols have been processed, the stack should be empty

Parentheses matching – the program

Developer Tools - http://interactivepython.org/runestone/static/...

Elements Network Sources Timeline Profiles Resources

<top frame>

```
> //Stack definition
  function Stack() {
     this.stac = new Array();
     this.pop = function() {
                uval = this.stac.pop();
                if (uval == undefined)
                   console.log("Trying to pop an empty stack");
                return uval;
     }
     this.push=function(item) {
                this.stac.push(item);
     }
     this.empty=function() {
                value = (this.stac.length == 0);
                return(value);
     }
  }
  undefined
>
```

Developer Tools - http://interactivepython.org/runestone/sta...

Elements Network Sources Timeline Profiles

<top frame>

```
> //Check function
  function CheckParen (symbol_string) {
     var mystack = new Stack();
     var balanced = true;
     var index = 0;
     while ((index < symbol_string.length) && balanced) {
        symbol = symbol_string[index];
        if (symbol == "(")
           mystack.push(symbol);
        else
           if (mystack.empty())
              balanced = false;
           else
              mystack.pop();
        index = index + 1;
     }
     if (balanced && mystack.empty())
        return "Matched";
     else
        return "Not matched";
  }
  undefined
>
```

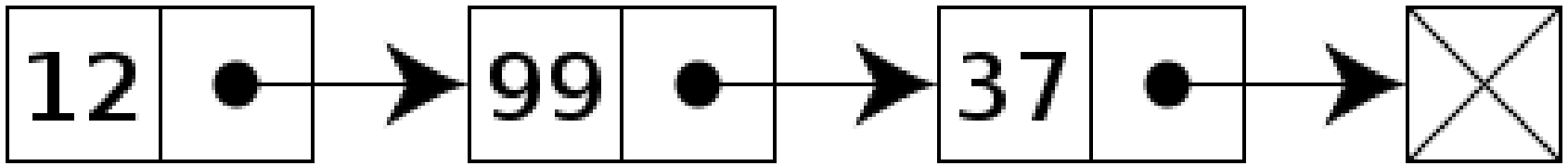
Developer Tools...

Elements

```
> CheckParen( "()()()" )
  "Matched"
> CheckParen( "()()()(" )
  "Not matched"
> CheckParen( "()()())" )
  "Not matched"
>
```

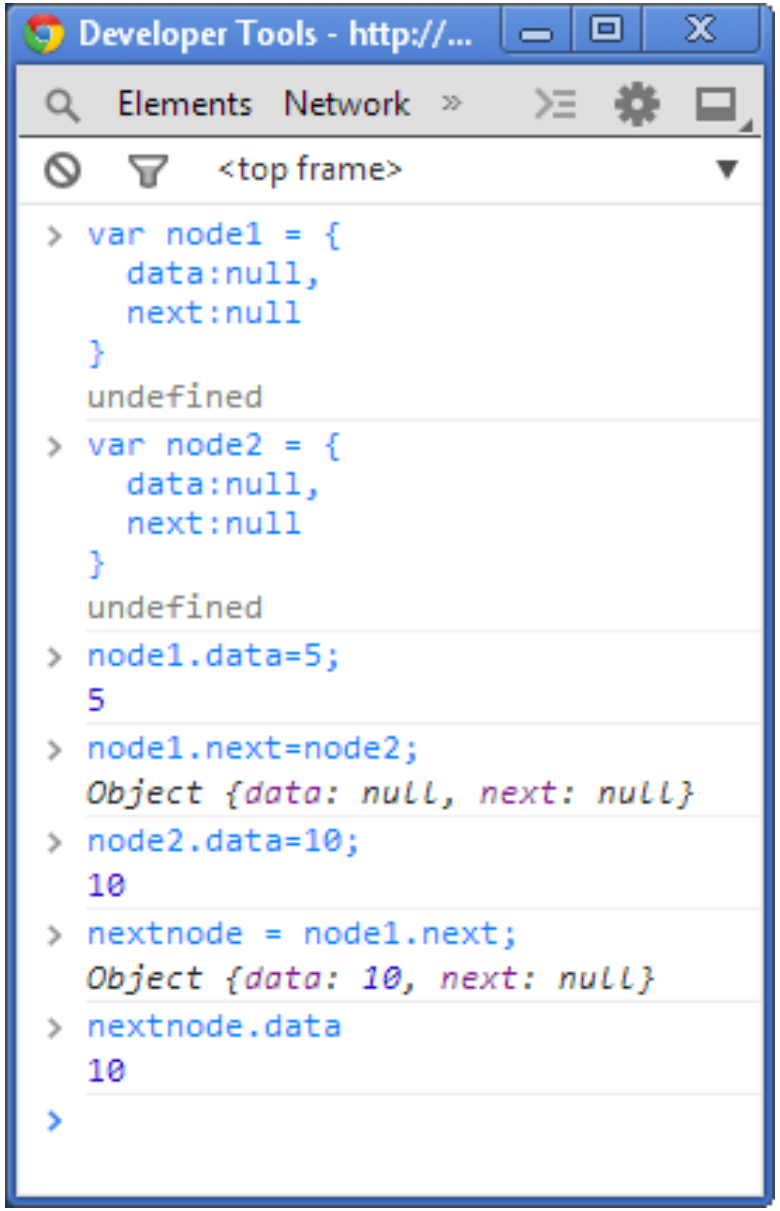
Linked list

- A data structure consisting of a group of nodes which together represent a sequence
- In a simple form, each node comprises a *value* and a *link* to the next node in the sequence … more complex variants add additional links.
- This structure allows for efficient insertion or removal of elements from any position in the sequence (as opposed to an array)



Node

- A node is a data structure that can be linked to another data structure
- Has two fields: *data* and *next*
  - Data stores the data held in the node and next is a reference to the next node in the list
- JavaScript is dynamically typed … *data* can store numbers, strings, objects, even another list
- This ability for a list to have another list stored as one of its nodes allows building more complex structures such as trees and acyclic graphs



Linked list implementation

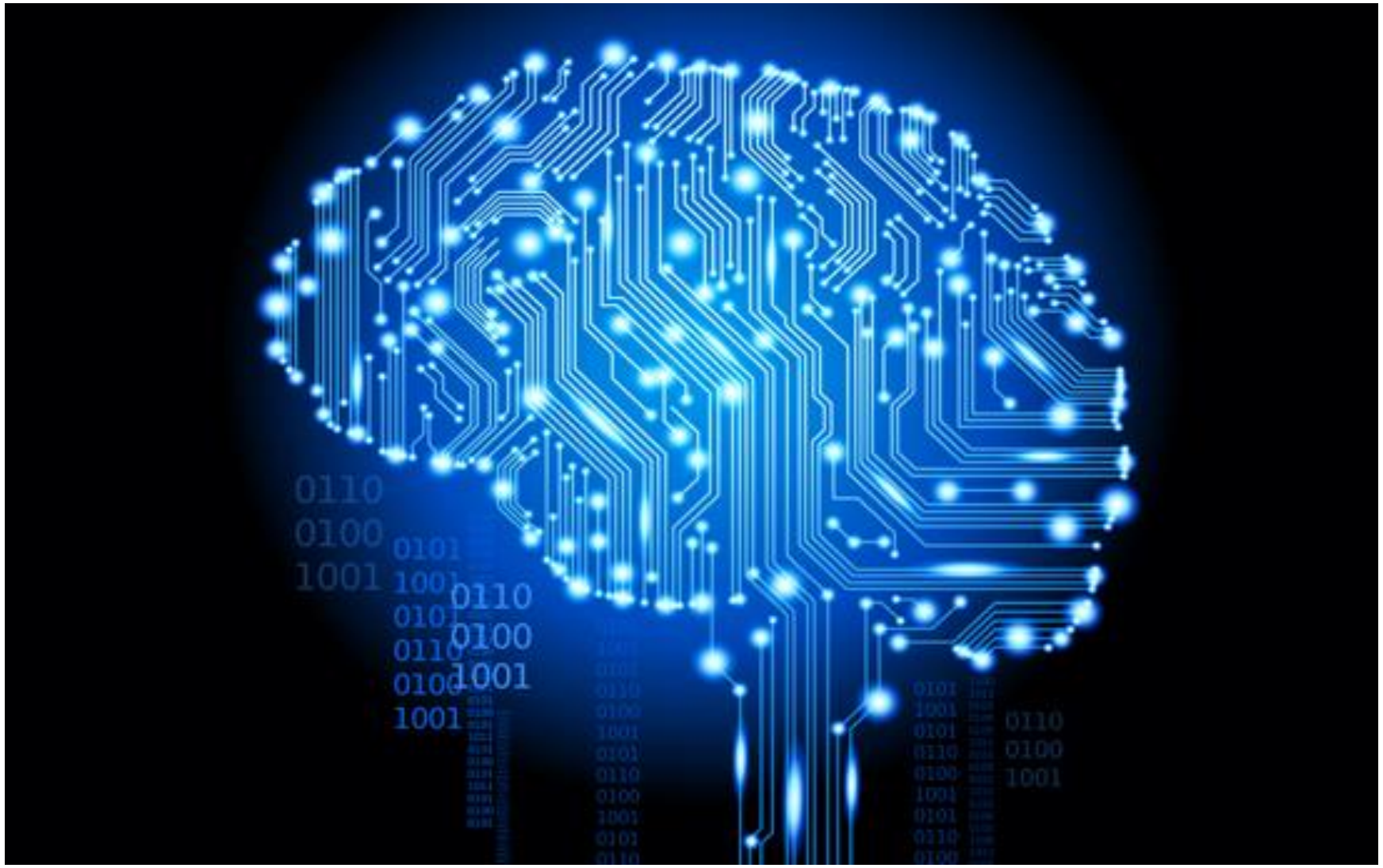
- Use a constructor function to create a List object that stores and retrieves data from a linked list
- By storing a reference to the *tail* of the list, avoid list traversal for adding a node at the end of the list

Binary tree

Hash table

Heap

Doubly linked list



Algorithms

References