

# Programovacie paradigmy

alias *Koštofka* programovacích jazykov

Peter Borovanský, KAI, I-18, borovan(a)ii.fmph.uniba.sk

<http://dai.fmph.uniba.sk/courses/PARA/>

alias: digmy(a)lists.dai.fmph.uniba.sk

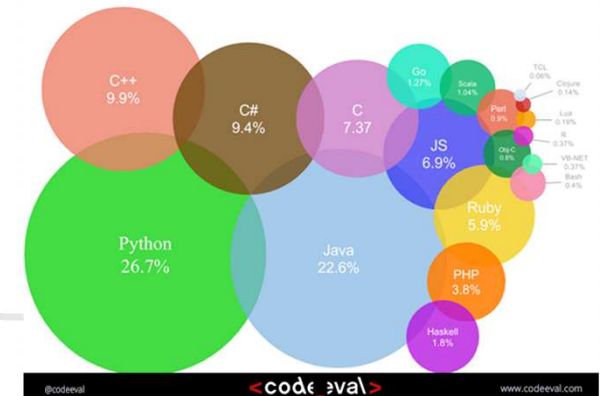
Uto, 13:10, F1



# Paradigmy 2019

(úvodné slovo)

Most Popular Coding Languages of 2016



Keď sa povie programovacie paradigmy, väčšina *klasikov* si spomenie na **funkcionálnu** a **logickú** paradigmu a **procedurálnu**, s **objektovou** odchýlkou.

- procedurálnu paradigmu poznáte z jazykov, ktoré nevznikli za vášho života, napr. v roku 1985 (C++) a 1995 (Java, JavaScript, Python). **Čo sa nič odvtedy neudialo?**
- prvú časť kurzu sa pozrieme na procedurálnu paradigmu z pohľadu novších / novodobjších programovacích jazykov a v skratke si priblížime princípy (procedurálnych) jazykov dneška z triedy **Scala**, Kotlin, Swift, ...
- v úvode preto nezažijete výrazný **paradigm shift** ☹
- na druhej strane, každý z dnešných moderných programovacích jazykov v nejakej miere podporuje **funkcionálnu paradigmu**, takže hranice nie sú striktné ...

~~@Deprecated~~

# Java 10 -release 3.2018

príklad čo poznáte



- Java je *zabetónovaná* od Java 1.0, 1995, ale snaží sa implementovať novinky nespútaných nových jazykov, ktoré nepotrebujú byť spätne kompatibilné.

@Deprecated

- v ktorom z dnešných programovacích jazykov sa dnes ešte píše povinne bodkočiarka (za príkazom) ?
- je to len dôsledok toho, že pred releasom Java 1.0 sa zdalo mnoho iných vážnejších problémov, ktoré riešiť, že sa J.Gosling neobťažoval domyslieť gramatiku jazyka (resp. syntaktický analyzátor), aby bodkočiarky neboli treba.

a odvtedy povinne píšme ;<cr><lf> podľa vzoru jazyka Algol (1960)/C (1985).

- ... a vôbec to nie je o bodkočiarke...  
Cieľom je, aby ste pochopili, koľko rôznych 'bodkočiarok' v jazyku Java existuje najmä kvôli spätnej kompatibilite.

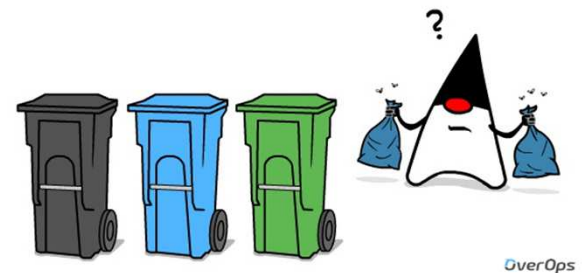
# Java 11

release 9.2018



<http://www.java-countdown.xyz/>

- Je Java mŕtva ? Asi nie, na jar vyšla Java 10 a dnes 11. (rýchla odpoveď...)
- Ale neumiera ? (... to už je iná otázka)
- Pozreli ste si novinky, ktoré Java 10/11 prináša ?
- Viete napísať program v Java 11, ktorý nie je v jazyku Java 10, Java 9 ?
- V histórii Javy ako programovacieho jazyka nastali výraznejšie dve revolúcie:
  - 2004 Java 5 prišla s generickými typmi,
  - 2014 Java 8 prišla s lambdami resp. s funkciami.
- ale fakticky najväčší prínos Java 8 bol asi Stream API... (ste videli v [Prog4](#))



Motto:  
Java is dead, long live JVM !



# Paradigmy 2019

(úvodné slovo)

---

V rámci kurzu sa stretnete s:

- konkuretným jazykom GO,
- funkcionálnym jazykom Haskell,
- logickým programovacím jazykom Prolog.

Prednáška preto dáva intro (ochutnávku) rôznych paradigiem  
nadvazujúce magisterské predmety idú hlbšie do princípov tej-ktorej paradigmy

očakáva sa schopnosť dohľadať si detaily potrebné k DÚ.  
a očakáva to dnes každý z vašich potenciálnych zamestnávateľov...

Ak vám tento prístup nevyhovuje, lepšie si zvoliť iný PV predmet...

# O čom to bude ?

(Paradigmy 2019)

Programming language features:

- static/dynamic typing
- type inference
- duck typing
- co-routines/goroutines
- *block chain* (?)
- tail recursion optimisation
- function as value
- lazy evaluation
- list comprehension
- pattern-matching
- backtracking and lots of recursion
- Horn clause, (SLD) resolution, choice point, unification
- constraint logic programming



# Hlavné paradigmy

- objektovo-orientovaná paradigma
- procedurálna paradigma
- funkcionálna paradigma
- logická (relačná) paradigma
- konkurentná paradigma

Paradigmy nie sú disjunktné množiny  
počas prednášky sa pozrieme na tri z nich v najčistejšej forme:  
CP – Go,  
FP – Haskell,  
LP – Prolog.





# Konkrétnejšie ?

(toto je plán)



- Princípy programovacích jazykov, 1x
  - príklady moderných jazykov na JVM (Scala, Kotlin)
- Go – procedurálna/konkurentná paradigma, 4x - október
  - konkurentné programovanie – go-routines
  - static typing
  - duck typing
- Haskell – funkcionálna paradigma, 4x - november
  - type inference
  - function as value
  - lazy evaluation
  - list comprehension
  - backtracking and recursion
  - pattern-matching
- Prolog – logická paradigma, 3x - december
  - Horn clause
  - resolution
  - choice point
  - unification



# Jazyky rokov

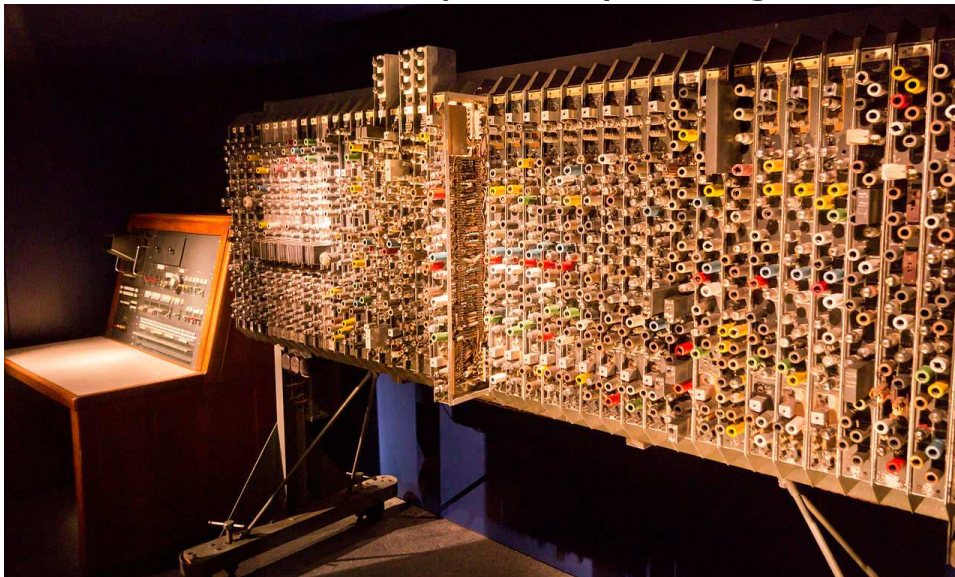
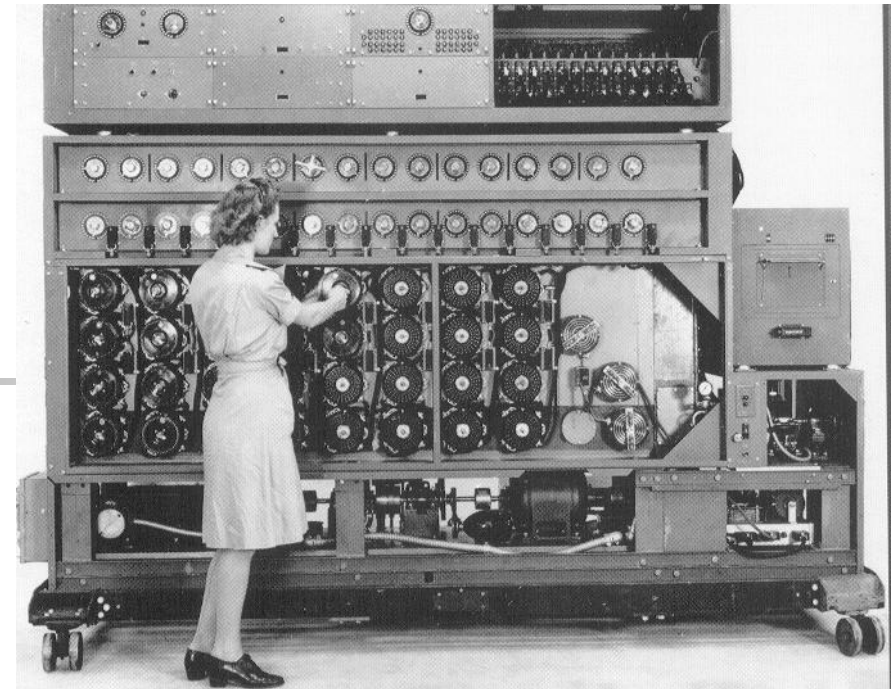
(1945-55-65-75-85-95-05-15)

- 1938 – binary code, The Bombe
- 1946 – Base 32, Ace

#computers =  $O(1)$

#programmers =  $O(1)$  - Alan Turing

We shall need a **great number** of **mathematicians of ability** because there will probably be a good deal of work of this kind to be done.



One of our difficulties will be the maintenance of an appropriate (i.e. discipline), so that we do not lose track of what we are doing

# Jazyky rokov

(1945-55-65-75-85-95-05-15)

- **1953 Fortran, Lisp – Functional Programming**
- IBM

#computers =  $O(10^2)$

#programmers =  $O(10^3)$

- **Inžinieri,**
- **Matematici,**
- **Fyzici,**

...

ľudia, ktorí majú:

- materskú profesiu,
- pracovné návyky,
- vedia komunikovať





# Jazyky roko

(1945-55-65-75-85-95-05)

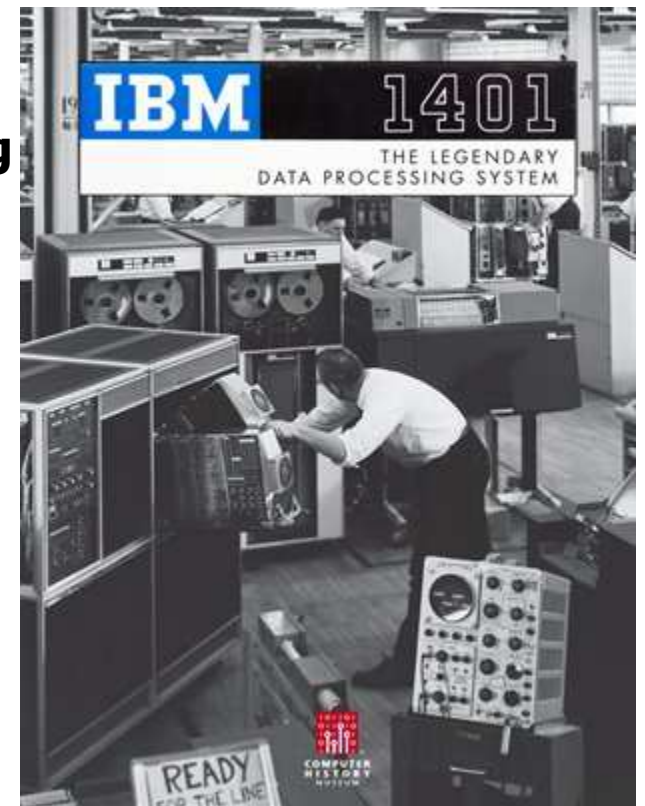


- 1960 Algol
  - 1960 COBOL, PL/1
  - 1967 Simula - OOP
  - 1968 Pascal
  - 1968 No-Goto – Structured Programming
  - 1968 C
- 
- IBM 1401, IBM 360

#computers =  $O(10^4)$

#programmers =  $O(10^5)$

Ukazuje sa prvý vážny nedostatok ľudí schopných ovládať počítače, tak vznikajú prvé univerzitne programy Computer Science...



# Počítač SIEMENS 4004 ÚVTVŠ

v Mlynskej doline



# Apollo 11

- 1969, 2kB+32kB ROM, 1MHz
- 1981, prvé IBM XT 16kB, 4MHz
- OS: multitasking pre 8 taskov
- I/O: signály, senzory, snímače vo frekvencii Hz (raz z sekundu ;-)
- Interface: skôr ako blikajúci switch
- Errors: 1201, 1202 - príliš veľa dát – radšej pristal N.Amstrong bez počítača
- Updates: treba prekódovať EPROM-ku

$$2^{(46/2)} \approx \text{milión} = 8\text{M}$$

$$2\text{kB} * 8\text{M} = 16\text{GB}$$

Dnes:

- používame QuadCore (2.23 GHz, 32..256GB+3GB RAM) na telefonovanie, ...
- často nevieme naprogramovať komunikáciu s odozvou, napr. cez Bluetooth,
- každé ráno nájdeme v mobile niekoľko updatov našich apps
- málo kto (už dnes) vie napísať aplikáciu do 16kB



# Jazyky rokov

(1945-55-65-75-85-95-05-15)

- 1972 Prolog – Logické programovanie
- 1972 SQL – Relačné programovanie
- 1973 C

**Počet programátorov  
sa zdvojnásobí každých 5 rokov**

**Takže každý druhý má <5  
rokov programátorskej praxe !!!**

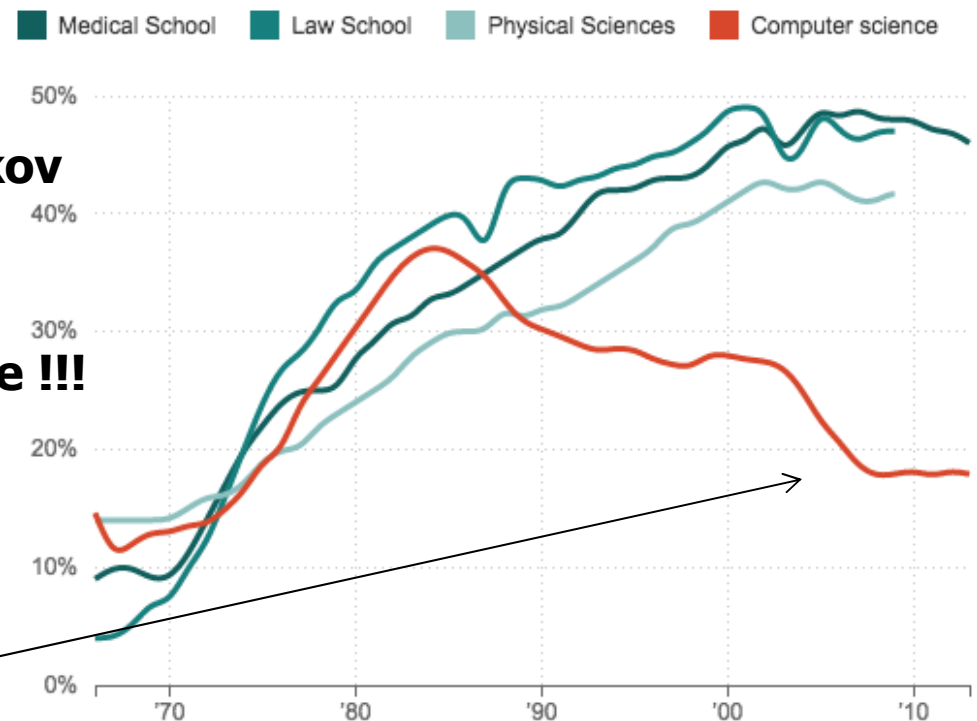
#computers =  $O(10^6)$

#programmers =  $O(10^7)$

Male domination

## What Happened To Women In Computer Science?

% Of Women Majors, By Field





# Jazyky rokov

(1945-55-65-75-**85**-95-05-15)

---

- **1980 C++**
- **1983 Ada**
- **1985 Eiffel**
- **1986 Objective-C**
- **1987 Perl**



# Araine 5



```
L_M_BV_32 := TBD.T_ENTIER_32S ((1.0/C_M_LSB_BV) * G_M_INFO_DERIVE(T_ALG.E_BV));  
if L_M_BV_32 > 32767 then  
    P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;  
elsif L_M_BV_32 < -32768 then  
    P_M_DERIVE(T_ALG.E_BV) := 16#8000#;  
else  
    P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M_BV_32));  
end if;  
P_M_DERIVE(T_ALG.E_BH) :=  
    UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH) *  
        G_M_INFO_DERIVE(T_ALG.E_BH)));
```

<https://itsfoss.com/a-floating-point-error-that-caused-a-damage-worth-half-a-billion/>



# Jazyky rokov

(1945-55-65-75-85-**95**-05-15)

---

## **Internet !**

- **1990 Haskell**
- **1991 Python**
- **1991 VB**
- **1995 Java**
- **1995 JavaScript**



# Jazyky rokov

(1945-55-65-75-85-95-**05**-15)

---

- **2001 C#**
- **2003 Scala**
- **2007 Clojure**
- **2009 Go**



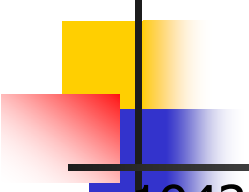
# Jazyky rokov

(1945-55-65-75-85-95-05-**15**)

---

- **2011 Dart**
- **2011 Kotlin**
- **2014 Swift**

# Historia programovacích jazykov

- 
- 1943 - ENIAC coding system
  - 1951 - Assembly Language
  - **1954 - FORTRAN** (J.Backus,IBM)
  - **1958 - LISP** (J.McCarthy)
  - 1958 - ALGOL (Backus-Naur)
  - 1959 - COBOL
  - 1962 - APL
  - 1962 - Simula (J.Dahl)
  - 1964 - BASIC
  - 1964 - PL/I
  - 1970 - Pascal (N.Wirth)
  - 1972 - C (D.Ritchie)
  - **1972 - Smalltalk** (A.Kay,Xerox)
  - **1972 - Prolog** (A.Colmenauer)
  - 1973 - ML
  - 1978 - SQL (T.Codd)
  - 1983 - Ada
  - 1983 - C++ (B.Stroustrup)
  - 1985 - Eiffel (B.Mayer)
  - 1987 - Perl
  - **1990 - Haskell**
  - **1990 - Python**
  - 1991 - Java (Sun)
  - 2000 - C#
  - 2006 - **Scala** (Martin Odersky)
  - 2007 - **Go**
  - 2014 - **Swift**
  - 2016 - **Kotlin**

*Computer Science without  
FORTRAN and COBOL is  
like birthday cake without  
ketchup and mustard.*



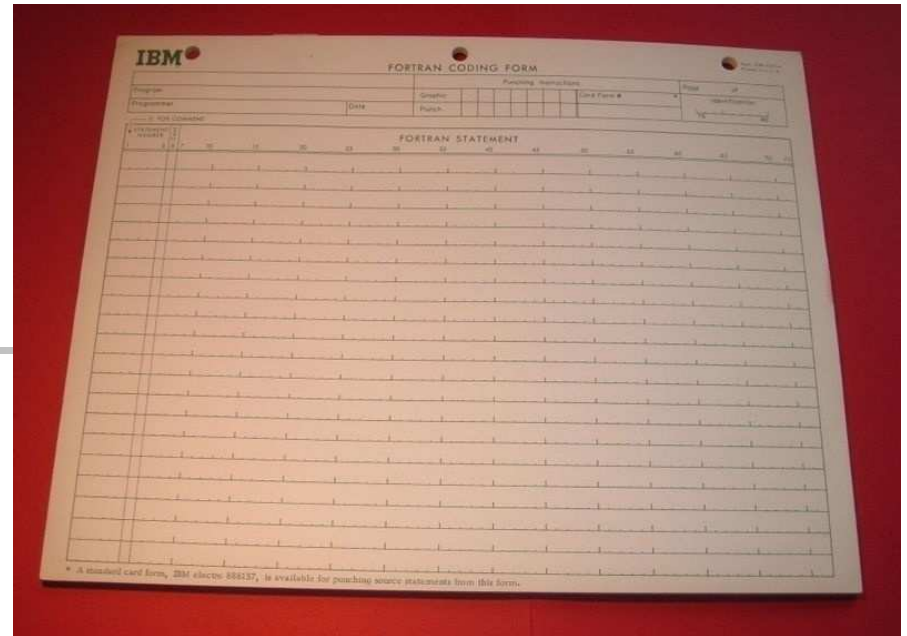
## 50-60 -te roky

---

- 1954 - **FORTRAN** (J.Backus,IBM)
  - vedecko-technické výpočty, numerické výpočty
- *1958 - LISP (J.McCarthy)*
- 1958 - **ALGOL** (Backus-Naur)
  - algoritmický jazyk, štruktúrované programy, riadiace štrukt.
- 1959 - **COBOL** (Pentagon)
  - biznis, financie
- *1962 - APL (Kenneth E. Iverson, Harvard)*
  - vektorovo orientovaný jazyk

# Fortran

Technické možnosti hardware ovplyvňujú  
programovacie jazyky a tým aj paradigmu



C It was the first programming language

C with the comments support!

```
WRITE (6,7)
```

```
7 FORMAT(15H Hello, world! )
```

```
STOP
```

```
END
```

```
WRITE(*,17) A, B, C
```

```
17 FORMAT(F6.3, F6.3, F10.5)
```

```
printf("%f6.3%f6.3%f10.5", A, B, C);
```





# Fortran

*"Consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, when FORTRAN abandoned the practice." —Sun FORTRAN Reference Manual*

## Cyklus, ktorý sčíta nepárne čísla

```
IF (X - Y) 100, 200, 300
```

```
if (x - y) < 0 then goto 100  
if (x - y) = 0 then goto 200  
if (x - y) > 0 then goto 300
```

```
if (x .lt. y) then goto 100  
if (x .eq. y) then goto 200  
if (x .gt. y) then goto 300
```



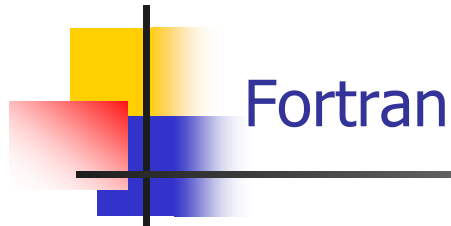
```
integer i, n, sum  
sum = 0
```

```
do 10 i = 1, n, 2  
sum = sum + i  
continue
```

```
do10i=1,n,2  
sum=sum+i  
10 continue
```

```
do10i=1100  
do10i=1,100
```

Fortran IV, Fortran 77, Fortran 95, Fortran 2003, ...  
What will the language of the year 2000 look like? ...  
Nobody knows but it will be called FORTRAN ☺



Fortran

*A good  
FORTRAN  
programmer  
can write  
FORTRAN code  
in any language*

- FORTRAN (IBM)
  - polia, priradenie, 3-IF, GOTO, DO
- FORTRAN II
  - SUBROUTINE, CALL
- FORTRAN III
  - inline assembler, strojovo závislý na IBM
- FORTRAN IV
  - strojovo nezávislý, boolean, logický IF
- FORTRAN 66
  - INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL
  - external, 6-písmenové identifikátory
- FORTRAN 77
  - CHARACTER typ, DO WHILE-END DO, bitové operácie
- Fortran 90
  - case-sensitive, moduly, rekurzívne procedúry, overloading
  - pointre, allocate a deallocate, dynamické dát.štruktúry
  - exit, inline comments
- Fortran 2003
  - objektovo-orientovaný, dedenie, polymorfizmus,
  - procedúry ako pointre
- Fortran 2008

"GOD is REAL (unless declared INTEGER)."



```
FUNCTION NGCD(NA, NB)
  IA = NA
  IB = NB
1  IF (IB.NE.0) THEN
    ITEMP = IA
    IA = IB
    IB = MOD(ITEMP, IB)
    GOTO 1
  END IF
  NGCD = IA
  RETURN
END
```

```
program GCD
  integer m, n, r
10  print *, 'Please give values for m and n'
    read *, m, n
20  if (n .eq. 0) go to 30
    r = mod(m,n)
    m = n
    n = r
    go to 20
30  print *, 'gcd = ', m
    go to 10
end
```

```
! Hello World in Fortran 90 and 95
PROGRAM HelloWorld
  WRITE(*,*) "Hello World!"
END PROGRAM
```



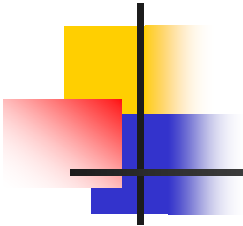
# COME FROM

The author feels that the COME FROM will prove an invaluable contribution to the field of computer science. It is confidently predicted that this solution will be implemented in all future programming languages, and will be retrofitted into existing languages.

```
10 J=1
11 COME FROM 20
12 WRITE (6,40) J STOP
13 COME FROM 10
20 J=J+2
40 FORMAT (14)
```

```
I = 1
IF (I .LT. 10) COME FROM 50
I = I+1
50 WRITE (6,60) I
STOP
60 FORMAT (14)
```

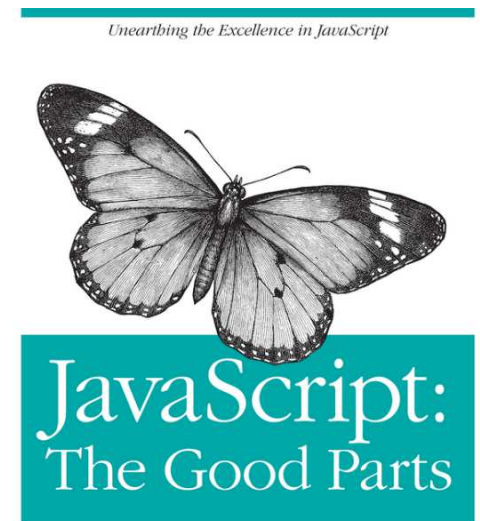
```
DO 200 INDEX=1,10
10 X=1.
20 X=X*2.
30 X=X*3.
40 X=X*4.
50 X=X*5.
60 X=X*6.
70 X=X*7.
80 X=X*8.
90 X=X*9.
100 X=X*10.
COME FROM
(10,20,30,40,50,60,70,80,90,100),INDEX
WRITE (6,500) INDEX,X
200 CONTINUE
STOP
500 FORMAT (14,2X,F12.0)
```



Je toto koniec bizarností v programovacích jazykoch ?

Pokus váš presvečiť, že ich je mnoho aj v dnešných jazykoch.

Motivované knihou [Douglas Crockford](#), 2008





# JavaScript: The Good Parts

(Douglas Crockford)

---

If a feature is sometimes useful and sometimes dangerous, and **if there is** a better option, then **always use the better option**.

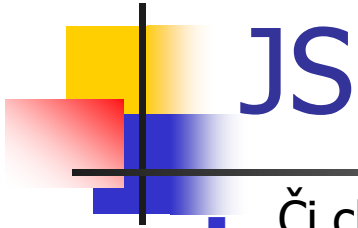
We are not paid to use every feature of the language.

We are paid to write programs that work well and are free of errors.

A good programming language should teach you.

It took a generation to agree that:

- high level languages were a good idea,
- goto statement was a bad idea,
- objects were a good idea,
- lambdas (functions) were a good idea.



- Či chcete, alebo nie, Javascript je assembler internetu,
- pri písaní aplikácií sa mu nedá vyhnúť, viete ho len zakryť frameworkom/iným (napr. typovaným) jazykom, ktorý vám nakoniec JS vygeneruje,
- chýbajú typy,
- môžete v ňom začať písať aj bez príručky, je natoľko intuitívny,
- má špecifikáciu (Ecmascript ES6 = ES 2015, 16, 17)
- a o to viac zradný (najmä pre C++/Java programátora),
- *môžete ho milovať alebo nenávidieť, nič medzi ☺*



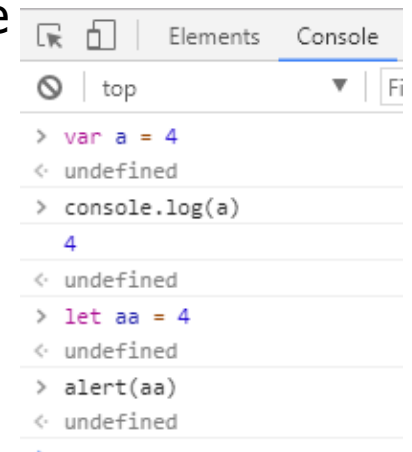
# Ako to začalo

(dnes je to JavaScript ES6)



Brendan Eich

- interactive language for Netscape
- LiveScript: 10 dní na prototyp vychádzajúci z:
  - Java - syntax, aj kus názvu jazyka
  - Scheme/Lisp – lambdas – funkcie a functionálny pohľad,
  - Self/Smalltalk – Smalltalk bez tried.
- 1995, zároveň sa úspešne rozvíja Goslingova Java
- Silná spoločná motivácia pre JS a Java: byť nezávislý od platformy Microsoft  
ten si vyvíja Jscript(Active Script), J++
- Súboj Java Appletov a JavaScript vyhráva JS na plnej čiare
- Na úvodné programovanie v JS nepotrebuje čítať žiadnu obsiahlu dokumentáciu
- Stačí si otvoriť konzolu od browsera (alebo [repl.it](https://repl.it)) a "programujete"



# JS má rôzne implementácie

## (ES6)

COMPAT ES

ECMAScript

5

6

2016+

next

intl

non-standard

compatibility table

Flattr 0

by kangax

Gratip

Sort by Engine types

Show obsolete platforms

Show unstable platforms

V8

SpiderMonkey

JavaScriptCore

Chakra

Carakan

KJS

Other

Minor difference (1 point)

Small feature (2 points)

Medium feature (4 points)

Large feature (8 points)

		Compilers/polyfills					Desktop browsers												
Feature name	Current browser	Traceur	Babel + core-js <sup>[2]</sup>	Closure	Type-Script + core-js	es6-shim	Konq 4.14 <sup>[3]</sup>	IE 11	Edge 14	Edge 15	Edge 16 Preview	FF 52 ESR	FF 55	FF 56 Beta	FF 57 Nightly	CH 61, OP 48 <sup>[1]</sup>	CH 62, OP 49 <sup>[1]</sup>	CH 63, OP 50 <sup>[1]</sup>	
Optimisation																			
proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	
Syntax																			
default function parameters	7/7	4/7	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	6/7	7/7	7/7	7/7	7/7	7/7	7/7	
rest parameters	5/5	4/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
spread (...) operator	15/15	15/15	13/15	12/15	4/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	
object literal extensions	6/6	6/6	6/6	4/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	
for...of loops	9/9	9/9	9/9	6/9	3/9	0/9	0/9	0/9	7/9	9/9	9/9	7/9	9/9	9/9	9/9	9/9	9/9	9/9	
octal and binary literals	4/4	2/4	4/4	4/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	
template literals	5/5	4/5	4/5	3/5	3/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
RegExp "y" and "u" flags	5/5	3/5	3/5	0/5	0/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
destructuring declarations	22/22	20/22	21/22	20/22	15/22	0/22	0/22	0/22	21/22	22/22	22/22	21/22	22/22	22/22	22/22	22/22	22/22	22/22	
destructuring assignment	24/24	23/24	24/24	21/24	19/24	0/24	0/24	0/24	23/24	24/24	24/24	23/24	24/24	24/24	24/24	24/24	24/24	24/24	
destructuring parameters	24/24	19/24	21/24	18/24	16/24	0/24	0/24	0/24	22/24	23/24	23/24	21/24	24/24	24/24	24/24	24/24	24/24	24/24	
Unicode code point escapes	2/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	2/2	2/2	2/2	1/2	2/2	2/2	2/2	2/2	2/2	2/2	
new.target	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	



# Slabé typovanie

---

- no typing - assembler
- weak static typing (C, C++, Objective C)  
`char z = "hello";` // núti definovať typy všetkých objektov, ale interne  
... // si pamätá, koľko bajtov zaberá  
`printf("%i", z);` // keď ju chceme formátovať, treba pripomenúť typ  
viac a viac typových obmedzení počas času kompilácie (C->C++)

```
char z = 'A';  
printf("%d, %c", z, z);  
65, A
```

Čo je 749711097 ?

- 749711097 = 249903699 \* 3
- 74 97 110 97 = "Jana"
- 74 9711097 = **GMT**: Monday 4. October 1993 5:04:57

<http://coding.smashingmagazine.com/2013/04/18/introduction-to-programming-type-systems/>



# Silné statické typovanie

---

- strong static typing (Java)

informácia o type je prístupná v čase behu, a VM teda môže kontrolovať typ  
ak chceme pretypovať objekt na iný typ, Java v čase behu realizuje kontrolu  
výsledok: menej hackov ako C, C++, jazyk je viac rigidnejší

```
public class Person {  
    public String getName() {  
        return "zack";  
    }  
}  
    // mimo classy nemôžeme definovať žiaden kód  
public class Main {  
    public static void main (String args[]) {  
        Person person = new Person();  
        System.out.println("The name is " + person.getName());  
    }  
}
```

[zdroj:http://coding.smashingmagazine.com/2013/04/18/introduction-to-programming-type-systems/](http://coding.smashingmagazine.com/2013/04/18/introduction-to-programming-type-systems/)



# Silné dynamické typovanie

---

- strong dynamic typing (**JS**, Python, Ruby)

```
var person = {  
    getName: function() {  
        return 'zack';  
    }  
};  
if (new Date().getMinutes() > 29) {  
    person = 5;  
}  
alert('The name is ' + person.getName());
```



# Silné dynamické typovanie

---

- strong dynamic typing (JS, **Python**, Ruby)

```
if (new Date().getMinutes() > 29):  
    z = "aaa"  
else:  
    z = 5  
print z  
z = z+z
```

// typ z sa zisťuje až v čase behu programu  
// - pol hodinu je to String + String  
// - a pol hodinu int + int

if it walks like a duck  
and quacks like a duck,  
then it must be a duck.



# Duck typing

---

Objekt patrí do tej  
triedy ako kváka...

- duck typing (**JS**, Python, Ruby)

```
var person = {  
    getName: function() { return 'zack'; }  
};
```

```
person['getBirthday'] = function() { return 'July 18th'; };
```

```
person['getName'] = 5;      // person.getName is not a function  
person['getName'] = null;   // person.getName is not a function  
person['getName'] = undefined; // person.getName is not a function
```

```
console.log('The name is ' + person.getName() + ' ' +  
    'and the birthday is ' + person.getBirthday());
```





# 0.1 + 0.2 == 0.3

---

```
if (0.1 + 0.2 == 0.3) {  
    console.log(":-)");  
} else {  
    console.log(":-(");  
}
```

```
if (0.1 + 0.2 == 0.3):  
    print(":-)");  
else:  
    print(":-(");
```

```
#include <iostream>  
int main() {  
    if (0.1 + 0.2 == 0.3) {  
        std::cout << ":-)";  
    } else {  
        std::cout << ":-(";  
    }  
}
```

```
main = do  
    if (0.1 + 0.2 == 0.3) then  
        putStrLn ":-)"  
    else  
        putStrLn ":-("
```

$(1 \div 10) + (2 \div 10) - (3 \div 10)$



# var vs. let

---

```
function varTest() {  
    var x = 1;  
    if (true) {  
        var x = 2;           // rovnaká premenná, var ignoruje block-scope  
        console.log(x);      // 2  
    }  
    console.log(x);          // 2 - tvrdá rana pre programátora vyškoleného  
                             // v C, C++, Java, ...  
}  
  
function letTest() {  
    let x = 1;  
    if (true) {  
        let x = 2;           // rôzne premenné  
        x = x+1;              // premenná definovaná let je modifikovateľná  
        console.log(x);      // 3  
    }  
    console.log(x);          // 1  
}
```



# const vs. let

---

```
function constTest() {  
  const x = 1;  
  if (true) {  
    const x = 2;          // rôzne premenné  
    // x = x+1;  
    console.log(x);      // 2  
  }  
  console.log(x);        // 1  
}
```

Množstvo konštánt, ktoré používame v programoch, definujeme ako premenné s inicializáciou a nikdy nezmeníme.

Pritom `const` je konštrukcia deklarujúca, že niečo sa nebude meniť.

# null vs. undefined

(javascript unikát)

- JS má dve hodnoty pre nedefinovanú hodnotu
  - null a undefined
- `typeof 12` `'number'`
- `typeof 'wow'` `'string'`
- `typeof null` `'object' ???`
- `typeof undefined` `'undefined'`



Tony Hoare

NULL (okrem iných vecí ako QuickSort, CSP, ...) objavil/zaviedol Tony Hoare pre nedokonalosť typového systému vtedy Algolu.

Dnešné jazyky (Swift, Kotlin) to riešia typmi String (never null), String?

2009, he apologised for inventing the [null reference](#):<sup>[20]</sup>

## **I call it my billion-dollar mistake.**

It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language ([ALGOLW](#)). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

# = VS. == VS. ===

(jediný jazyk, kde == nie je tranzitívne)

- `0 == ''` // true, platí aj opačne
- `0 == '0'` // true, platí aj opačne
- `'' == '0'` // false

- `false == 'false'` // false
- `false == '0'` // true
- `" \t \r \n \t " == 0` // true

- `0 === ''` // false
- `0 === '0'` // false
- `'' === '0'` // false
- `false === 'false'` // false
- `false === '0'` // false
- `" \t \r \n \t " === 0` // false

## false:

- false
- 0 (zero)
- "" or ""
- null
- undefined
- NaN (napr. 1/0)

Všetko ostatné je **true**:

- '0'
- 'false'
- [] (an empty array)
- {} (an empty object)
- function(){} (an "empty" f)

# Vždy === miesto ==

	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[]	[0]	[1]	NaN
true																					
false																					
1																					
0																					
-1																					
"true"																					
"false"																					
"1"																					
"0"																					
"-1"																					
""																					
null																					
undefined																					
Infinity																					
-Infinity																					
[]																					
{}																					
[]																					
[0]																					
[1]																					
NaN																					

true==[] // false

Moral of the story:

Always use 3 equals unless you have a good reason to use 2.

<https://dorey.github.io/JavaScript-Equality-Table/>



# Viacriadkové reťazce

(indentácia je dôležitá)

---

Po **Fortrane**, ktorý zrušil medzery, lebo zaberali drahocenné miesto v 80-stĺpcovom riadku, prišiel **Algol/Pascal/C/C+/Java**, ktorý ignoroval layout programu. Akýkoľvek počet medzier bol ekvivalentný jednej a konce riadkov sa ignorovali (resp. stáli za medzeru). A každý preferoval iný formát.

Viacforemnosť zabíja. Vznikla potreba formátovať programy.

Prišiel **Python**, ktorý za cenu ušetrenia blokových zátvoriek predpísal layout/indentáciu programu.

Ale to nie je všetko, prečo je druhý program (JS) zlý ?

```
var long_line_1 = "This is a \
long line";    // ok
```

```
var long_line_2 = "This is a \
long line";    // syntax error
```



# Priradenie v teste

---

- `let a = 5, b = 7`
- `if (a = b) { ... } // a je 7`

čo je vlastne explicitne zapísané toto:

- `a = b`
- `if (a) { ... }`

avšak programátor možno myslel niečo iné

- `if (a == b) { ... }`
- `if (a === b) { ... }`





# Scope

(deklarujte všetky premenné na začiatku funkcie)

- JavaScript prevzal syntax blokov z C/Java, ale ignoruje **block-scope**, t.j. že premenná deklarovaná v bloku nie je inde viditeľná
- ak deklarujete (a príp. inicializujete) premennú v strede funkcie, viditeľná je aj pred definíciou, ale nemá hodnotu...

```
function foo() {  
    console.log(a); // je definovaná, ale má hodnotu undefined  
    var a = 0;  
    console.log(a); // 0  
}
```

```
for(var i = 1; i <= 10; i++) { } // ...  
console.log(i); // 11 // ... takto to bolo v Basicu...
```

Používajte let lebo akceptuje block-scope !



# Functional scope

---

- JavaScript síce používa blokovú syntax, nerešpektuje **block-scope**, ale pozná funkcie ako hodnoty, a rešpektuje **functional-scope**...

```
var x = 1, y = 2;
function f() {
    var y = 4, z = 6;
    // x = 1, y = 4, z = 6
    x += y+z;
    // x = 11, y = 4, z = 6
};
// x = 1, y = 2, z = undefined
f();
// x = 11, y = 2, z = undefined
```



# with

(pravdepodobne pochádza z Pascalu/Pythonu)

---

```
with (obj) {  
    a = b;  
}
```

a znamená (???):

- a = b;
- a = obj.b;
- obj.a = b;
- obj.a = obj.b;

V skutočnosti znamená toto (takže všetky štyri možnosti):

```
if (obj.a === undefined) {  
    a = obj.b === undefined ? b : obj.b;  
} else {  
    obj.a = obj.b === undefined ? b : obj.b;  
}
```



# Dobrovolné zátvorky bloku

---

Rovnaké v mnohých iných jazykoch

```
if (ok)
    t = true;
```

zmeníte na:

```
if (ok)
    t = true;
    foo( );
```

a myslíte tým toto:

```
if (ok) {
    t = true;
    foo( );
}
```

ale v skutočnosti ste napísali toto:

```
if (ok) {
    t = true;
}
foo( );
```



# Security update for iOS/OSX

Originálny Apple kód na verifikovanie certifikátu (2014, update iOS 7.0.6):  
Vidíte chybu v tomto kóde ? ... nevadí, ani Apple ju nenašiel...

```
if ((err = SSLFreeBuffer(hashCtx)) != 0)
    fail();
if ((err = ReadyHash(SSLHashSHA1, hashCtx)) != 0)
    fail();
if ((err = SSLHashSHA1.update(hashCtx, clientRandom)) != 0)
    fail();
if ((err = SSLHashSHA1.update(hashCtx, serverRandom)) != 0)
    fail();
if ((err = SSLHashSHA1.update(hashCtx, signedParams)) != 0)
    fail();
    fail();
if ((err = SSLHashSHA1.final(hashCtx, hashOut)) != 0)
    fail();
```

# Vsúvanie bodkočiarky

```
var q = {  
  lati_tude : 48.1234,  
  "longi-tude" : 17.234,  
  city : {  
    name: "Bratislava",  
    capital : true  
  }  
}
```

JS za príkazom vyžaduje bodkočiarku

ale svojou benevolentnosťou ju dovolí programátorovi nepísať a dopĺňa ju zaň

Aký je rozdiel, výsledok oboch kódov ?

```
function goo() {  
  ...  
  return ;  
  {  
    errorCode : 0  
  };  
}
```

```
➤ goo()  
=> undefined
```

```
function goo() {  
  ...  
  return {  
    errorCode : 0  
  };  
}
```

```
➤ goo()  
=> { errorCode: 0 }
```



# Globálne premenné

---

- sú zlé, najmä ak projekt presiahne rozumnú veľkosť
- deklarácia globálnej premennej  
`var foo = 5`
- cez globálny objekt  
`window.foo = 5`
- implicitná deklarácia  
`foo = 5`
- Našťastie JS ES 6 prichádza s modulmi

# GOTO story



## A Case against the GO TO Statement.

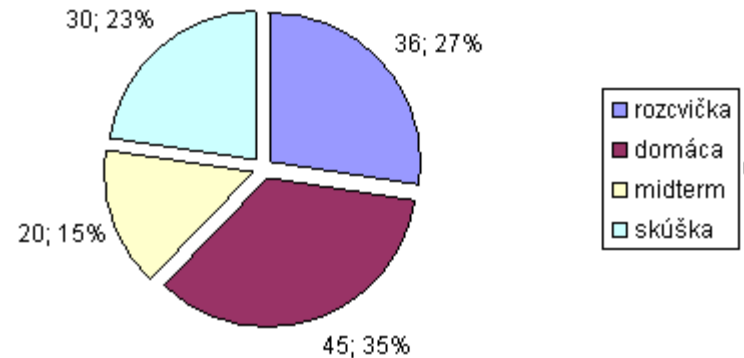
by Edsger W. Dijkstra  
Technological University  
Eindhoven, The Netherlands

Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code).

Edsger Dijkstra (1968). *"Go To Statement Considered Harmful"*, *Communications of the ACM*  
Frank Rubin (1987). *"'GOTO Considered Harmful' Considered Harmful"*, *Communications of the ACM*  
Donald Moore; ... (1987). *"'GOTO Considered Harmful' Considered Harmful' Considered Harmful?,"*  
Dijkstra, Edsger *On a Somewhat Disappointing Correspondence (EWD-1009)*



# Pre koho nie/je prednáška



nie je:

- neznáša, resp. nevidí dôvod naučiť sa, **rekurziu**
- učí sa len to, čo môže **zajtra použiť** v robote
- verí, že s javou, c++, php **prežije do dôchodku...**
- nerád **pracuje cez semester**

je:

- uznáva tzv. **programovanie so zamyslením**, ako analógiu čítania s porozumením
- rád hľadá a rieši malé „triky“ programíky a rébusy