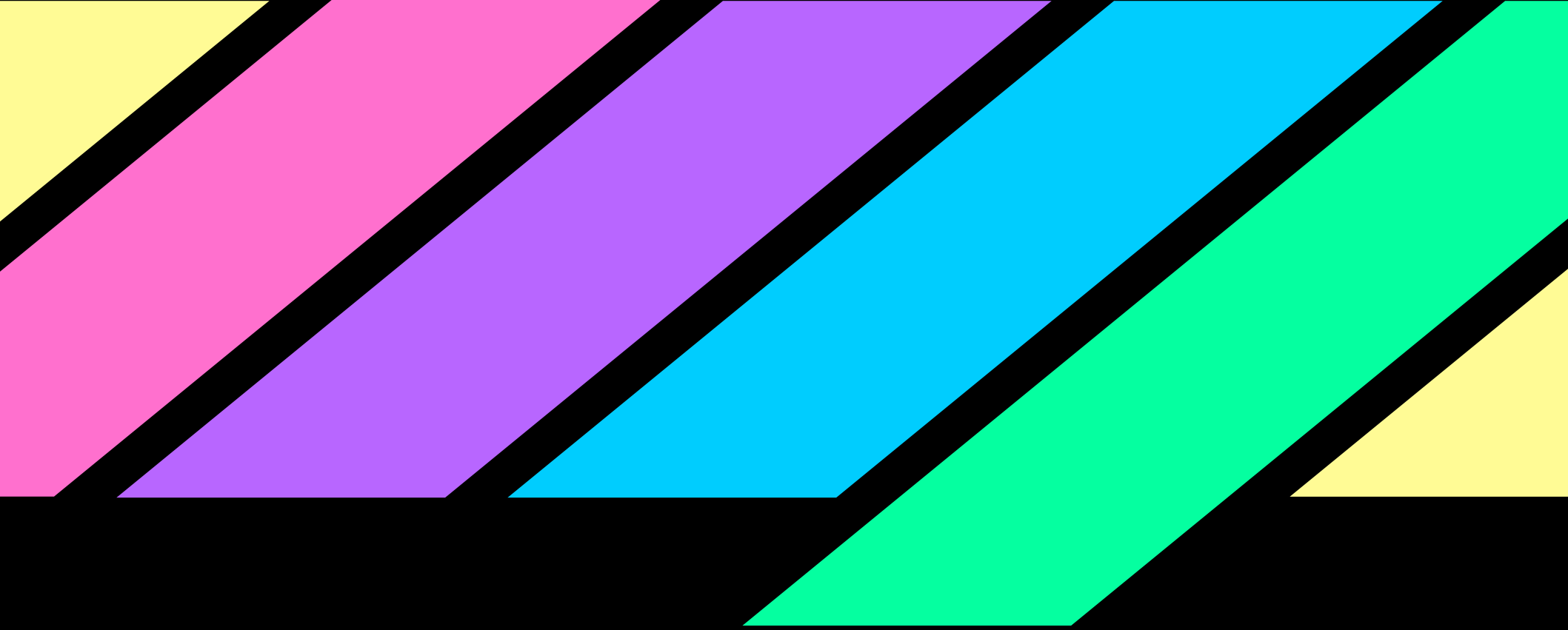


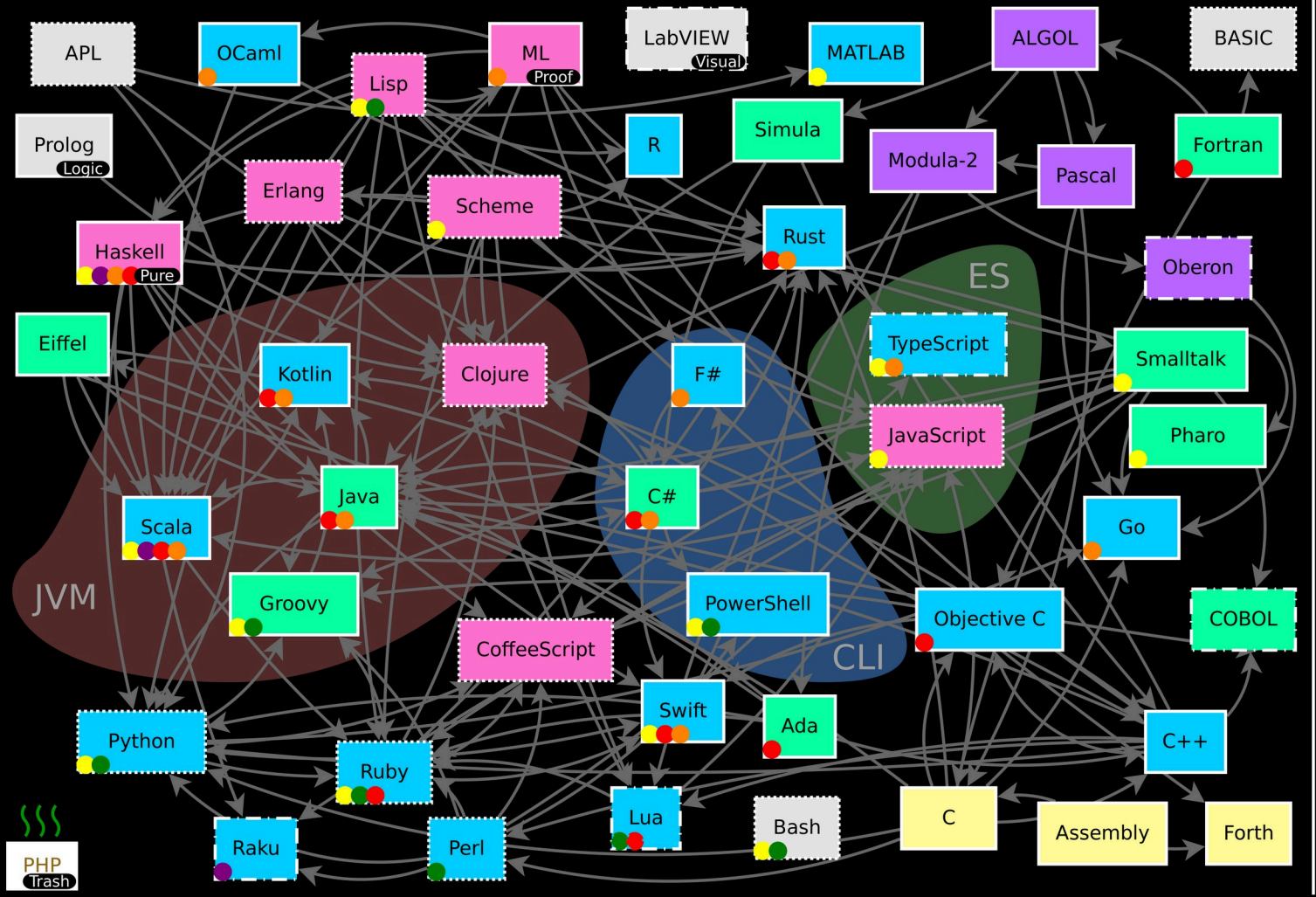
Programming Languages and Paradigms

Carol V. Alexandru-Funakoshi, Dr. sc.
Software Evolution and Architecture Lab
University of Zurich

Seminar – Spring 2022

Here's a tiny selection of languages

(by some accounts, there are ~8900 more: https://hopl.info/)



Paradigm

Mainly Object-Oriented | Mainly Functional | Multi-Paradigm | Mainly low-level | Structured | Other

Influence

Features

- Shell / REPL/ Interactive-mode available
- Also used for scripting
- Lazy evaluation built-in
- Type inference
- Can run on LLVM

Type System

"Weak", Dynamic, or None | Mixed | "Strong", "Strict", or Static

Runtimes / Virtual Machines

JVM: Java Virtual Machine
CLI: Common Language Interface (.NET)
ES: ECMAScript (JavaScript)

*mistakes included!


Before we start...

Registration

- **If you haven't been assigned to this course via IFI seminar allocation, you cannot participate!**
  - If you've booked the module anyway, cancel it in time!
- **Otherwise:**
  - Book BINFS157 (Bachelor's) or MINFS557 (Master's): https://www.students.uzh.ch/en/booking.html
  - In OLAT, we only use the MINFS557 course node. All Bachelor's students have already been added. Master's students are added automatically. https://lms.uzh.ch/auth/RepositoryEntry/17190518785/CourseNode/85421310414617
  - If you want to drop out, please let me know **today**!

Who am I?

- **Carol Alexandru, Dr. sc.**
  - Dissertation in Efficient Software Evolution Analysis and Software Visualization (2019) under Prof. Dr. Harald Gall
  - Senior research associate @ Software Evolution and Architecture Lab (s.e.a.l.)
- **Teaching**
  - Informatics I
  - Advanced Software Engineering
  - PLP
- **Primary Programming Languages:**
  - Bash, Scala, Python, Java, JavaScript
- **Dabbled in:**
  - Haskell, Lisp, Kotlin, Ruby, C, C++, TypeScript, R
- **Want to learn:**
  - C and Haskell properly, Rust

Who are you?

Dörig	Mauro
Zurbriggen	Max
Volontè	Sandro
Eiben	Samuel Christian
Aylward	Matthew Tyler
Puser	Dylan
Ratarov	Daniil
Moser	David
Sidler	Dominik
Villanthanam	Arjun
Rohe	Hannah
Salzer	Melanie
Bugmann	Diego
Stebler	Deborah
Crazzolara	Anton
Salzmann	Yannick

- "Hi, I'm <name>, a <semester/level> student.
- I know <languages> and some day I'd like to know how to work with <languages>.
- From this seminar, I hope to learn <xyz>.
- (Also <…>)"

Lecuture schedule

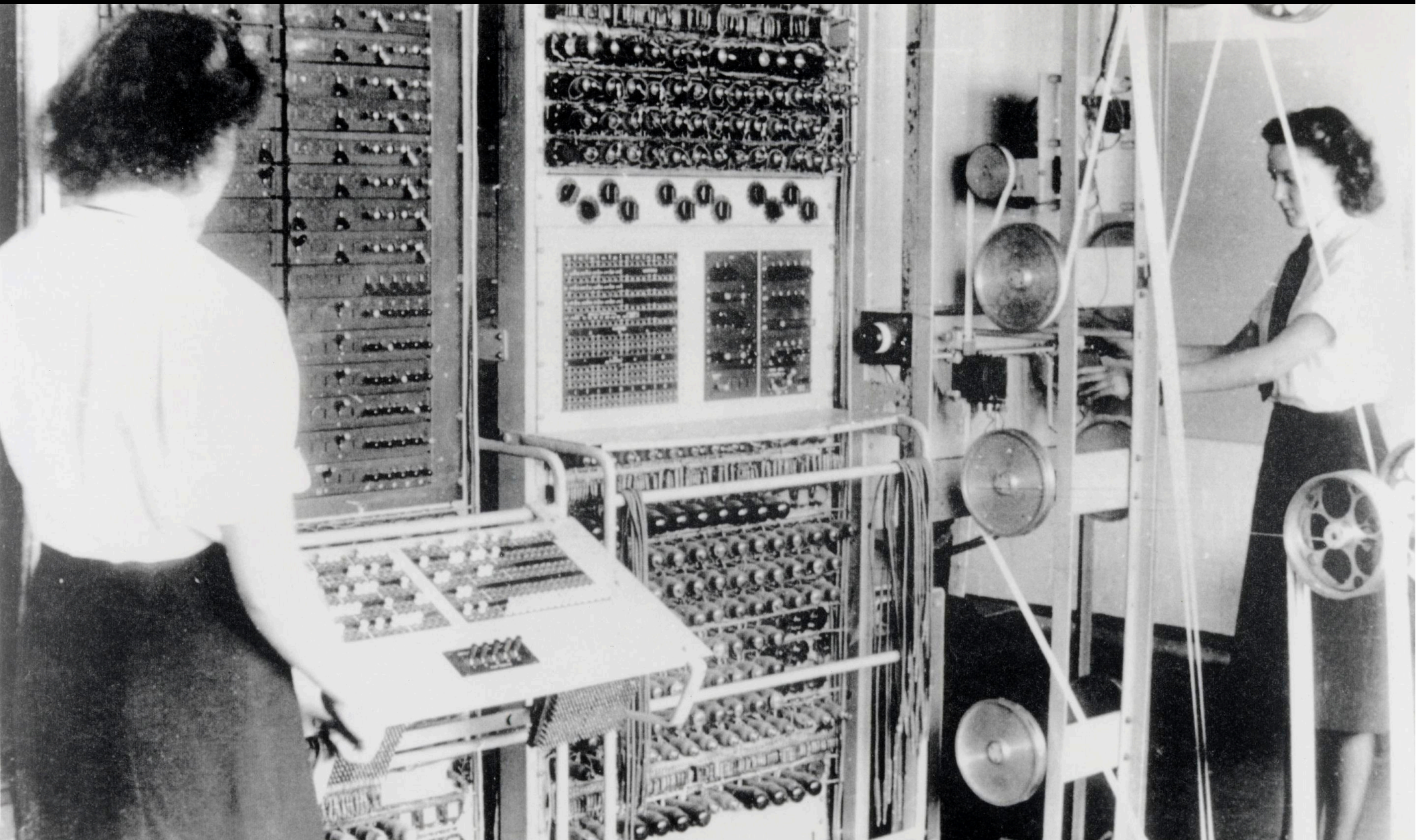
- **Today:**
  - A brief history of programming
  - A selection of programming paradigms and concepts
  - Seminar structure and deliverables
- **Next week:**
  - Lambda calculus, currying, higher-level functions, Hindley–Milner type system, purity, referential transparency, immutable data structures and persistence
  - Reflection & Macros
  - LLVM, Common Language Interface (CLI), JVM, etc.
  - How **not** to design a language: PHP
  - Esoteric programming languages

Course Schedule

	Online Session	Deadline (end of day)
Wed, 23.02.2022	Lecture	
Wed, 02.03.2022	Lecture	Pick your language
Wed, 09.03.2022		
Wed, 16.03.2022		Programming tasks #1
Wed, 23.03.2022		
Wed, 30.03.2022		
Wed, 13.04.2022	“Touch base”	Programming tasks #2
Wed, 20.04.2022		
Wed, 27.04.2022		Programming tasks #3
Wed, 04.05.2022		Seminar paper
Wed, 11.05.2022		Paper reviews
Wed, 18.05.2022	Student presentations	
Wed, 25.05.2022	Student presentations	
Wed, 01.06.2022	Student presentations & Wrap-up	Paper revision

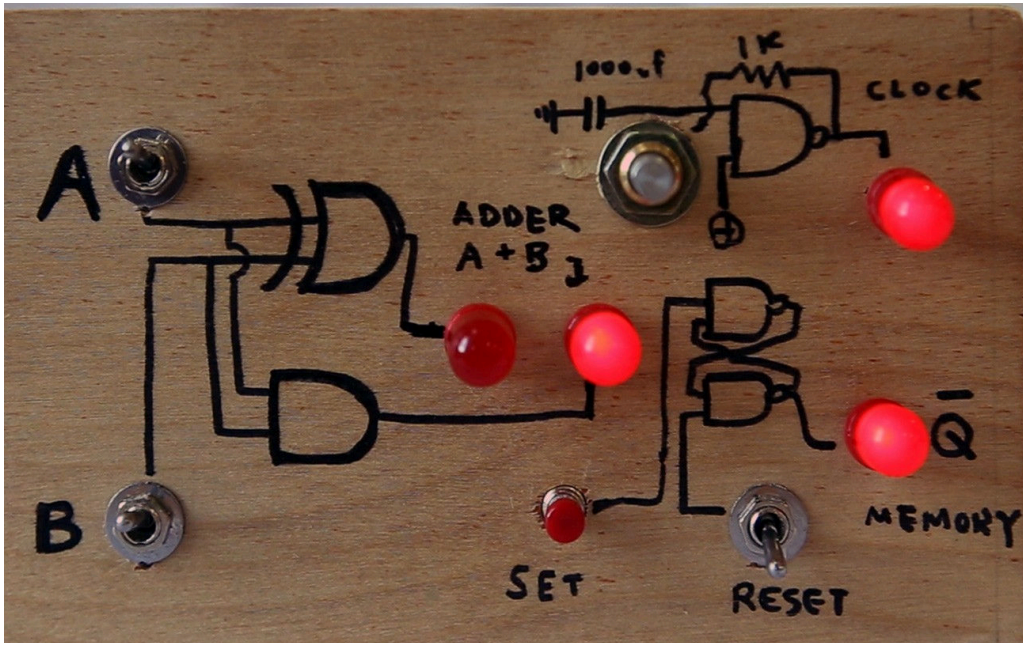
History & Language Generations

1943 Colossus: Plugs & Switches!



How Do Computers Compute?

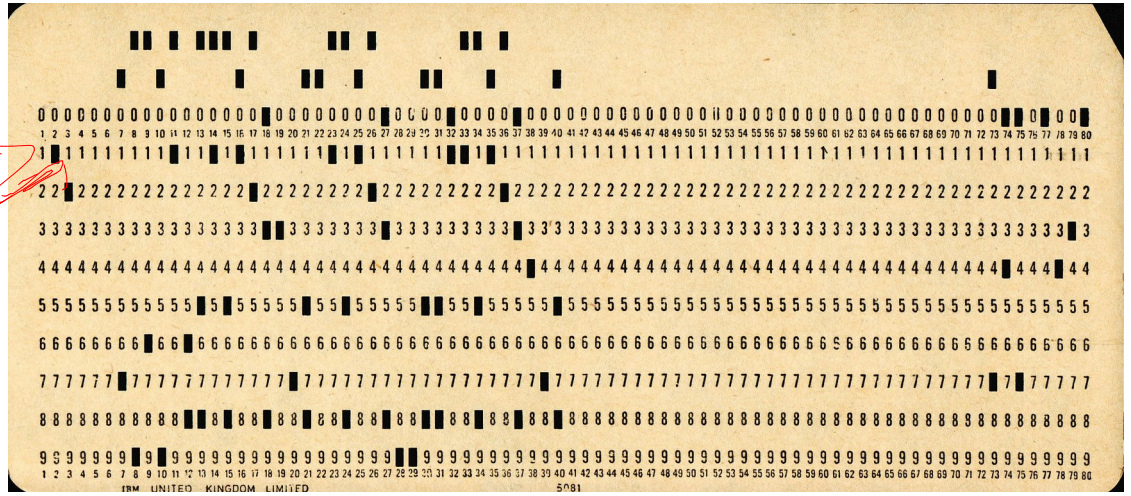
- **Addressable registers, devices and memory**
- **Instructions to move or execute operations on data**
- **So the questions is: how to tell the computer what to do?**



https://taeyoonchoi.com/poetic-computation/handmade-computers/handmade-computer/

1GL – 1st Generation Languages

- **Direct machine code, numbers (0/1, dec, hex...)**
  - Punch cards
  - Paper tape
  - Magnetic tape
  - Switches and Plugs



- **Very machine-specific**
- **Hard to read/debug, not human-oriented at all**

2GL – Giving Names to Numbers

- **Addressable registers, devices and memory**
  https://wiki.osdev.org/CPU_Registers_x86
  - E.g. in x86, the "accumulator" (32bit) is called EAX:

```
                                                  Register Name
| 0000 0001 0010 0011 0100 0101 0110 0111 |       EAX
|                     0100 0101 0110 0111 |       AX
|                               0110 0111 |       AL
|                     0100 0101           |       AH
```

- **Instructions to move or execute operations on data**
  https://en.wikipedia.org/wiki/X86_instruction_listings#Original_8086/8088_instructions
  - E.g. in x86: ADD, SUB, DEC, JMP, MOV…
- **Pretty good introduction to how a CPU works:**
  https://www.howtogeek.com/367931/htg-explains-how-does-a-cpu-actually-work/

2GL – Assembly

- **More human-readable machine code**
  - Words are “assembled” to machine code
- **x86 example:**
  - “store the decimal number 97 in to register AL”:

Machine Code	Semantics	Assembly
10110	“move”	MOV
000	Register AL	AL
01100001	97 in decimal	61h

  - Machine code: `10110000 01100001` or `B0 61`
  - Assembly:

```
MOV AL, 61h ; Load AL with 97 decimal (61 hex)
```

3GL – High-level Programming Languages

- **Favors the programmer, not the computer**
- **Features a (usually complex) translation step from writte source code to machine execution**

Source Code	compile			Machine Code

e.g. C, C++

Source Code	compile	Bytecode	compile	Machine Code

e.g. Java > Java Bytecode > JIT, C# > Common Language Interface > JIT

Source Code	interpret	Bytecode	interpret	Machine Code

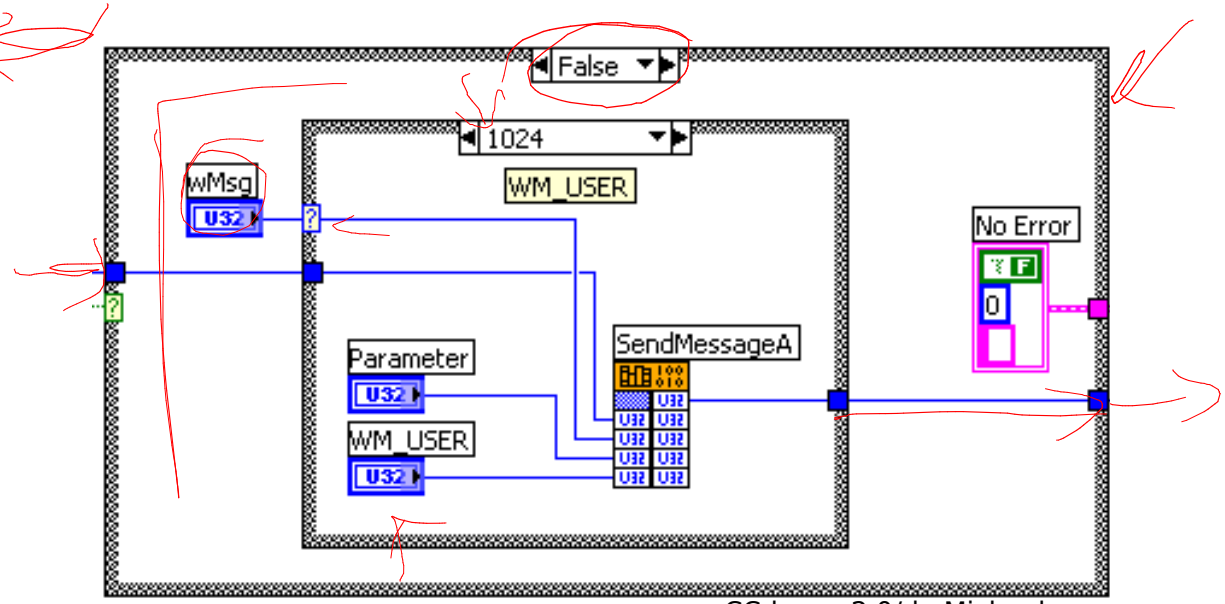
e.g. Python > Python Bytecode > Python runtime (e.g. cpython)

Source Code	transpile	Lower level language source code	compile	Machine Code

e.g. Haskell (Core > STG > C-- > C/ASM/LLVM)

4GL – “Program-generating”

- **Idea from the 1970-1990s**
- **High-level, usually domain-specific, e.g.:**
  - SQL (`select * from … where … order by … limit …`)
  - LabVIEW (visual)
  - R, MATLAB, …
- **Fuzzy definition**



5GL: “Because… computers!!! …?”

- **“The user just states the problem, the computer solves it”.**
- **Constraint-based and Logic Programming**
- **Examples:**
  - OPS5, Mercury, ICAD
- **Mostly a pipe dream, reserved for specific applications where the problem can be formally stated.**

• PL Milestones: Plankalkül (1948)

- **Considered the 1st high-level (3GL) language**

```
P1 max3 (V0[:8.0],V1[:8.0],V2[:8.0]) → R0[:8.0]
max(V0[:8.0],V1[:8.0]) → Z1[:8.0]
max(Z1[:8.0],V2[:8.0]) → R0[:8.0]
END
P2 max (V0[:8.0],V1[:8.0]) → R0[:8.0]
V0[:8.0] → Z1[:8.0]
(Z1[:8.0] < V1[:8.0]) → V1[:8.0] → Z1[:8.0]
Z1[:8.0] → R0[:8.0]
END
```

  - Assignment: →, comparison: < > ≤ ≥ = ≠ etc.
  - Arrays, tuples, conditions, for/while loops
  - **Not actually implemented!**

PL Milestones: Short Code (1949)

- **1st 3GL that actually ran on a computer:**

```
a = (b + c) / b * c

X3 =  (  X1 +  Y1 )  /  X1 * Y1
X3 03 09 X1 07 Y1 02 04 X1   Y1

07Y10204X1Y1
0000X30309X1
```

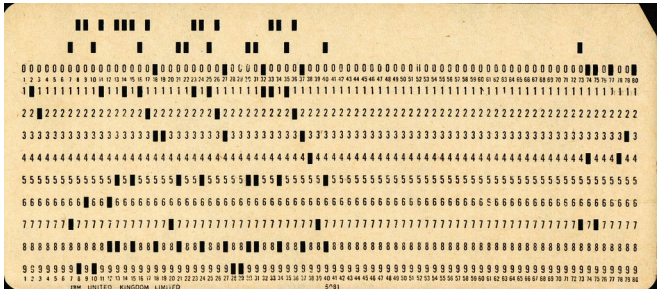
  - Ran on BINAC and UNIVAC I
  - Branching, calls to library functions
  - Interpreted (50 times slower than machine code)

PL Milestones: FORTRAN (1957–2018)

- **IBM Mathematical Formula Translating System**
  - 1st optimizing compiler
  - Hardware makers provided FORTRAN compilers
  - Remains the most popular language for scientific computing even today!
  - Has evolved a lot over the decades:

```
12 PIFRA=(A(JB,37)-A(JB,99))/A(JB,47)          PUX 0430
```

```
program helloworld
      print *, "Hello, World!"
end program helloworld
```

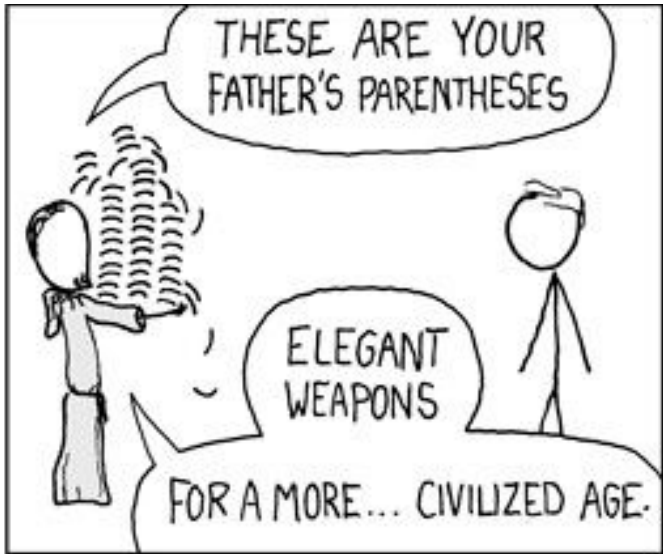


PL Milestones: Lisp (1957–2021)

- **“Everything is a list” (incl. the source code!)**
  - Pioneered tree data structures, automatic storage, dynamic typing, conditionals, higher-order functions, recursion, read–eval–print loop, NIL, macros, …

```
(+ (* (/ 9 5) 60) 32)) ; 140
```

```
(defun AreaOfCircle (prefix)
  (print "Enter Radius: ")
  (setq radius (read))
  (setq area (* 3.1416 radius radius))
  (format t "~a: ~F" prefix area))
(AreaOfCircle "The area is")
```



3GL Milestones: Simula (1962)

- **1st object-oriented programming language**
  - objects, classes, inheritance, garbage collection…

```
Class Rectangle (Width, Height); Real Width, Height; Begin
    Real Area, Perimeter;

    Procedure Update; Begin
      Area := Width * Height;
      Perimeter := 2*(Width + Height)
    End of Update;

    Boolean Procedure IsSquare;
      IsSquare := Width = Height;

    Update;
    OutText("Rectangle created: "); OutFix(Width,2,6);
    OutFix(Height,2,6); OutImage
End of Rectangle;
```

PL Milestones: COBOL(1959-2014)

- **Banking and Business**
  - English-like syntax
  - Extremely verbose
  - “Self-documenting” and “Easily readable”
  - 220 billion LOC in use by banks today
  - Slowly declining use, but still very popular

```
* FIZZBUZZ.COB
* cobc -x -g FIZZBUZZ.COB
*
IDENTIFICATION        DIVISION.
PROGRAM-ID.           fizzbuzz.
DATA                  DIVISION.
WORKING-STORAGE       SECTION.
01 CNT      PIC 9(03) VALUE 1.
01 REM      PIC 9(03) VALUE 0.
01 QUOTIENT PIC 9(03) VALUE 0.
PROCEDURE             DIVISION.
*
PERFORM UNTIL CNT > 100
 DIVIDE 15 INTO CNT GIVING QUOTIENT REMAINDER REM
 IF REM = 0
   THEN
       DISPLAY "FizzBuzz " WITH NO ADVANCING
   ELSE
       DIVIDE 3 INTO CNT GIVING QUOTIENT REMAINDER REM
       IF REM = 0
         THEN
             DISPLAY "Fizz " WITH NO ADVANCING
         ELSE
             DIVIDE 5 INTO CNT GIVING QUOTIENT REMAINDER REM
             IF REM = 0
               THEN
                   DISPLAY "Buzz " WITH NO ADVANCING
               ELSE
                   DISPLAY CNT " " WITH NO ADVANCING
             END-IF
       END-IF
 END-IF
 ADD 1 TO CNT
END-PERFORM
DISPLAY ""
STOP RUN.
```

PL Milestones: Smalltalk (1972-1980)

- **"Everything is an object" + message passing**
- **Objects can only:**
  - Hold state
  - Receive a message from other objects or itself
  - While processing a message, send messages
- **Reflection + live programming**
- **Integrated development environment**
- **Lives on in Pharo (2020), Squeak (2020), …**

PL Milestones: Prolog (1972-2000)

- **4th-generation language (4GL)**
- **Logic Programming**
  - Declarative (i.e. not imperative/procedural)
  - Used for theorem proving, expert systems, automated planning, natural language processing.

```
:- use_module(library(clpfd)).

sudoku(Rows) :-
        length(Rows, 9), maplist(length_(9), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [A,B,C,D,E,F,G,H,I],
        blocks(A, B, C),
        blocks(D, E, F),
        blocks(G, H, I).

length_(L, Ls) :- length(Ls, L).

blocks([], [], []).
blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-
        all_distinct([A,B,C,D,E,F,G,H,I]),
        blocks(Bs1, Bs2, Bs3).

problem(1, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).
```

PL Milestones: ML (1973)

- **Polymorphic Hindley–Milner type system**
  - Static type system with type inference
  - Verified using formal semantics
- **Today, this type system can be found in OCaml and Haskell (among others)**

```
let fizzbuzz i =
  match i mod 3, i mod 5 with
    0, 0 -> "FizzBuzz"
  | 0, _ -> "Fizz"
  | _, 0 -> "Buzz"
  | _    -> string_of_int i

let _ =
  for i = 1 to 100 do print_endline (fizzbuzz i) done
```

PL Milestones: C (1972-2018)

- **Low-level, direct memory access, minimal runtime support**
- **But at the same time: maximum portability**
- **Many other programming languages are implemented in and/or transpile to C**

```
int main(void) {
  char *s = "Hello world";
  *s = "Byebye world";
  printf(s);
}
```

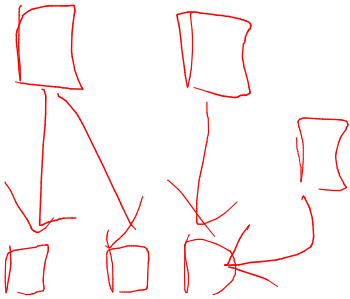
```
> gcc -w hello.c && ./a.out
zsh: segmentation fault (core dumped)  ./a.out
```

PL Milestones: The Internet Age

- **Python (1990)**
  - Duck typing, modularity, productivity, "strong philosophy"
- **Visual Basic (1991)**
  - "Rapid Application Development", UI, DB, ActiveX
- **PHP (1995)**
  - "**P**ersonal **H**ome**p**age", no formal spec until 2014
- **Ruby (1995)**
  - "Object-oriented scripting language"
- **Java (1995)**
  - "Write once, run anywhere", highly portable
- **Delphi / Object Pascal (1995)**
  - "Rapid Application Development", Language + IDE + Libraries

PL Milestones: Current Trends

- **Functional being built into mainstream PLs:**
  - C++11, Perl, PHP, Python, Go, Java, C#
- **Performance + Safety**
  - Go, Rust, Kotlin, Java, C#
- **Extending object-oriented programming:**
  - Mixins, traits, typeclasses, aspects
- **Massively parallel computing / pipelines**
  - GPUs (machine learning), CUDA, OpenCL



Programming Paradigms & Programming Concepts


Paradigms According to Wikipedia
Paradigm
Structured concurrency
Inductive programming
Template
Relativistic programming
Subject-oriented
Role-oriented
By separation of concerns:
Aspect-oriented
Automatic
Macro
Metaprogramming
Attribute-oriented
Reflective
Recursive
Polymorphic
Block-structured
Dynamic/scripting
Concurrent computing
Agent-oriented
Data-driven
Structured
Actor-based
Object-oriented
Intentional
Class-based
Non-structured
Array
Literate
Imperative
Prototype-based
Concurrent
Action
Set-theoretic
Automata-based
Probabilistic
Nondeterministic
Quantum
Function-level
Point-free style
Concatenative
Generic
Stack-based
Procedural
Symbolic
Natural-language programming
Event-driven
Differentiable
Language-oriented
Domain-specific
Value-level
Parallel computing
Array-oriented
Concurrent constraint logic
Constraint logic
Constraint
Process-oriented
Declarative
Ontology
Dataflow
Flow-based
Abductive logic
Functional
Functional logic
Reactive
Functional reactive
Logic
Answer set
Concurrent logic
Purely functional
Inductive logic

Declarative vs. Imperative

	Declarative	Imperative
Core principle	Describe **what** the program should accomplish (not listing explicit steps)	Describe **how** the program accomplishes something

Declarative vs. Imperative

	Declarative	Imperative
Core principle	Describe **what** the program should accomplish (not listing explicit steps)	Describe **how** the program accomplishes something
Advantages (with a grain of salt)	Minimize side-effects (>referential transparency), simplifies parallel programs, higher-level abstractions	Direct hardware control means faster/smaller.

Declarative vs. Imperative

	Declarative	Imperative
Core principle	Describe **what** the program should accomplish (not listing explicit steps)	Describe **how** the program accomplishes something
Advantages (with a grain of salt)	Minimize side-effects (>referential transparency), simplifies parallel programs, higher-level abstractions	Direct hardware control means faster/smaller.
Disadvantages (with a grain of salt)	Less efficient (slower, larger), state-change and IO can be “weird”	Unintended side-effects, too many degrees of freedom

Declarative vs. Imperative

	Declarative	Imperative
Core principle	Describe **what** the program should accomplish (not listing explicit steps)	Describe **how** the program accomplishes something
Advantages (with a grain of salt)	Minimize side-effects (>referential transparency), simplifies parallel programs, higher-level abstractions	Direct hardware control means faster/smaller.
Disadvantages (with a grain of salt)	Less efficient (slower, larger), state-change and IO can be “weird”	Unintended side-effects, too many degrees of freedom
Sub-paradigms / Concepts	Functional, Logic, Constraint, Dataflow	Procedural, Object-Oriented

Declarative vs. Imperative

	Declarative	Imperative
Core principle	Describe **what** the program should accomplish (not listing explicit steps)	Describe **how** the program accomplishes something
Advantages (with a grain of salt)	Minimize side-effects (>referential transparency), simplifies parallel programs, higher-level abstractions	Direct hardware control means faster/smaller.
Disadvantages (with a grain of salt)	Less efficient (slower, larger), state-change and IO can be “weird”	Unintended side-effects, too many degrees of freedom
Sub-paradigms / Concepts	Functional, Logic, Constraint, Dataflow	Procedural, Object-Oriented
Example languages	SQL, HTML, Haskell, Scheme, ML, Prolog	FORTRAN, COBOL, BASIC, C, Java, Python, JavaScript, etc.

Functional vs. Object-Oriented

	Functional	Object-oriented
Core principle	Applying & composing functions/expressions to *map values to other values*	Instances of classes (blueprints) hold state & behavior and interact with each other

Functional vs. Object-Oriented

	Functional	Object-oriented
Core principle	Applying & composing functions/expressions to *map values to other values*	Instances of classes (blueprints) hold state & behavior and interact with each other
Advantages Disadvantages	Largely a matter of opinion revolving around state, mutability and preferred higher-level abstractions / composability	

Functional vs. Object-Oriented

	Functional	Object-oriented
Core principle	Applying & composing functions/expressions to *map values to other values*	Instances of classes (blueprints) hold state & behavior and interact with each other
Advantages Disadvantages	Largely a matter of opinion revolving around state, mutability and preferred higher-level abstractions / composability	
Sub-paradigms / Concepts	1st-class and higher-order functions, purity, recursion, referential transparency	Inheritance, delegation, mixins, encapsulation, design patterns

Functional vs. Object-Oriented

	Functional	Object-oriented
Core principle	Applying & composing functions/expressions to *map values to other values*	Instances of classes (blueprints) hold state & behavior and interact with each other
Advantages Disadvantages	Largely a matter of opinion revolving around state, mutability and preferred higher-level abstractions / composability	
Sub-paradigms / Concepts	1st-class and higher-order functions, purity, recursion, referential transparency	Inheritance, delegation, mixins, encapsulation, design patterns
Example languages (mostly)	Lisp, Scheme, Wolfram Language, Racket, Ocaml, Haskell, ML	Java, C++, C#, Groovy, Smalltalk, Simula, COBOL

Functional vs. Object-Oriented

	Functional	Object-oriented
Core principle	Applying & composing functions/expressions to *map values to other values*	Instances of classes (blueprints) hold state & behavior and interact with each other
Advantages Disadvantages	Largely a matter of opinion revolving around state, mutability and preferred higher-level abstractions / composability	
Sub-paradigms / Concepts	1st-class and higher-order functions, purity, recursion, referential transparency	Inheritance, delegation, mixins, encapsulation, design patterns
Example languages (mostly)	Lisp, Scheme, Wolfram Language, Racket, Ocaml, Haskell, ML	Java, C++, C#, Groovy, Smalltalk, Simula, COBOL
Hybrid Languages	C++11, Kotlin, Python, Rust, Raku, Scala, JavaScript, TypeScript, MATLAB, and many more...	

Sequential vs. Concurrent

- **Sequential / single-threaded:**
  - No race conditions, easier to debug

Sequential vs. Concurrent

- **Sequential / single-threaded:**
  - No race conditions, easier to debug
- **Concurrent / parallel / multi-threaded:**
  - Increased throughput, high responsiveness / low latency
  - Race conditions if not thread-safe, can be harder to debug
  - Related concepts
    - shared memory, message passing, actors, software transactional memory (STM), process calculus

Shared Memory

- **Usually uses a locking mechanism**
  - Mutex: data can only be unlocked by locking process
  - Binary semaphore: any process can unlock
- **Can be very fast**
- **Disadvantages:**
  - Hard to implement correctly, overlapping operations must be considered by the programmer
  - Deadlocks / Livelocks must be avoided
  - Priority inversion (low-priority task may have to wait for high-priority task)

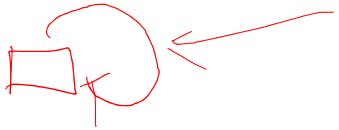
Software Transactional Memory

- **More optimistic: no locking mechanism**
- **Shared access via logged "Transactions"**
  - Begin transaction
  - Modify data "as a copy" ← copy on write
  - Commit: verify that same data has not been altered by other processes and finally write
- **Transactions logically happen at a single moment in time (atomic), much easier to write parallel programs**
- **Has a small performance overhead**

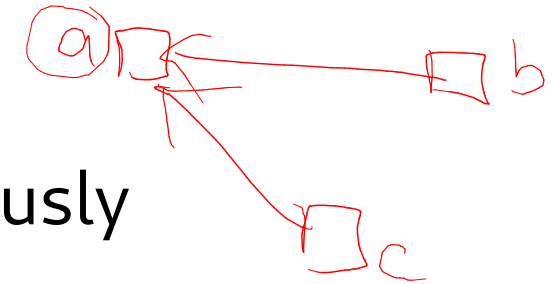
Message Passing / Actors

- **Methods not called directly by name, instead: sender sends a message, object decides what to do with it**
- **Objects typically can only alter their own state**
- **Can make life easier when**
  - Writing concurrent/multi-threaded/distributed programs
  - Managing/debugging state
- **Popular implementations:**
  - Built-in: Erlang, Scala
  - As a library: Java, Rust, Swift, JavaScript, C/C++, …

Event-Driven & Reactive



- **Event-driven:**
  - Main loop listens for events (hardware, network, UI...)
  - Usually asynchronous programs
  - Used heavily for GUI and web (i.e. JavaScript)



- **Reactive:**
  - Pipelines of data updated continously
  - E.g.: a := b+c, a will be updated if b or c are updated
  - Reactive programming is more rare, but a common example is MVC (model-view-controller)

Paradigms: a Conclusion

- **Most languages cannot be assigned to one specific paradigm**
- **Most paradigms have a fuzzy definition**
- **Last century: ideology**
- **Today: mixing the best of all worlds**
- **The goal?**
  - Easier programming, fewer bugs, better maintainability

Seminar Structure & Organization

Course Structure

- **2 introductory lectures + ~~optional sessions~~**
- **1 mid-semester "touch base" session**
- **Everyone…**
  - picks a different language to learn and explore
  - implements the same programming tasks
  - gives a ~~15~~ 10min presentation
  - writes a 3-page seminar paper
  - reviews 3 papers of other students
- **3 presentation sessions + ~~1 wrap-up session~~**

Deliverable: Programming Tasks

- **Three sets of tasks**
  - 1st: Getting started
  - 2nd: Typical, small-scale programming challenges
  - 3rd: Larger, more complete application
- **Tasks will be similar for all students/languages**
- **Deadlines throughout the semester**
- **"Touch base"session:**
  - 5min demo of 2nd programming task running on your own machine (screen sharing), comment on how it's going

Deliverable: Presentation

- **15+5min Presentation**
- **Content:**
  - Brief story/background/purpose of the language
  - Main distinguishing features & paradigms
  - Pros / cons + your experience using the language
  - Show the latest programming task running on your machine
- **The style is up to you, doesn't need to be formal:**
  - Tutorials/Demos welcome
  - Just try to make it interesting for everyone
- **You must share your slides as a PDF in OLAT**

Deliverable: Seminar paper

- **3 Pages (incl. references), IEEE Template:**
  - https://www.ieee.org/conferences/publishing/templates.html
- **Content:**
  - History, motivation for existing, related work
  - Describe paradigms used, distinguishing features
  - Include examples & compare to other languages
  - Discuss implications, opportunities, chances
- **Sources must be cited properly**
- **Should be formal and proper**

Deliverable: Seminar paper

- **Typical paper structure:**
  - Introduction (brief overview, motivation)
  - Related work (relevant literature, history)
  - Approach/Method/The X programming language
    - Describe in detail how things work
  - Discussion
    - Put the language into context, muse about problems, future opportunities, outlook, etc and reflect on the language.
  - Conclusion (brief summary)
  - References (use a bibliography file/tool!)

Deliverable: Peer Reviews

- **You will read 3 papers writte by your peers**
- **For each, you will write a ½ page review consisting of:**
  - Brief summary of the paper (4-6 sentences)
  - Your own opinion/evaluation of the paper. More on how to do this properly later!
- **You will be able to read the reviews and revise your paper if you'd like to.**

Course Schedule

	Online Session	Deadline (end of day)
Wed, 23.02.2022	Lecture	
Wed, 02.03.2022	Lecture	Pick your language
Wed, 09.03.2022		
Wed, 16.03.2022		Programming tasks #1
Wed, 23.03.2022		
Wed, 30.03.2022		
Wed, 13.04.2022	“Touch base”	Programming tasks #2
Wed, 20.04.2022		
Wed, 27.04.2022		Programming tasks #3
Wed, 04.05.2022		Seminar paper
Wed, 11.05.2022		Paper reviews
Wed, 18.05.2022	Student presentations	
Wed, 25.05.2022	Student presentations	
Wed, 01.06.2022	Student presentations & Wrap-up	Paper revision

Study goals

- Recognize and be able to roughly explain the paradigms and concepts taught in the lectures
- Learn to program in 1 new language
- Learn about 11 other programming languages
- Practice writing seminar papers
- Learn how to write a differentiated review
- Broaden your coding horizon!

Before we wrap up…

- **Any requests?**
- **Write in the forum if you have a question or desire a session on a specific topic**
  - Q&A session: informal, you just want to discuss some problem (general attendance not required)
  - Optional lecture: several people wish for some content (general attendance required)
- **Write me an email if you have non-public requests: alexandru@ifi.uzh.ch**