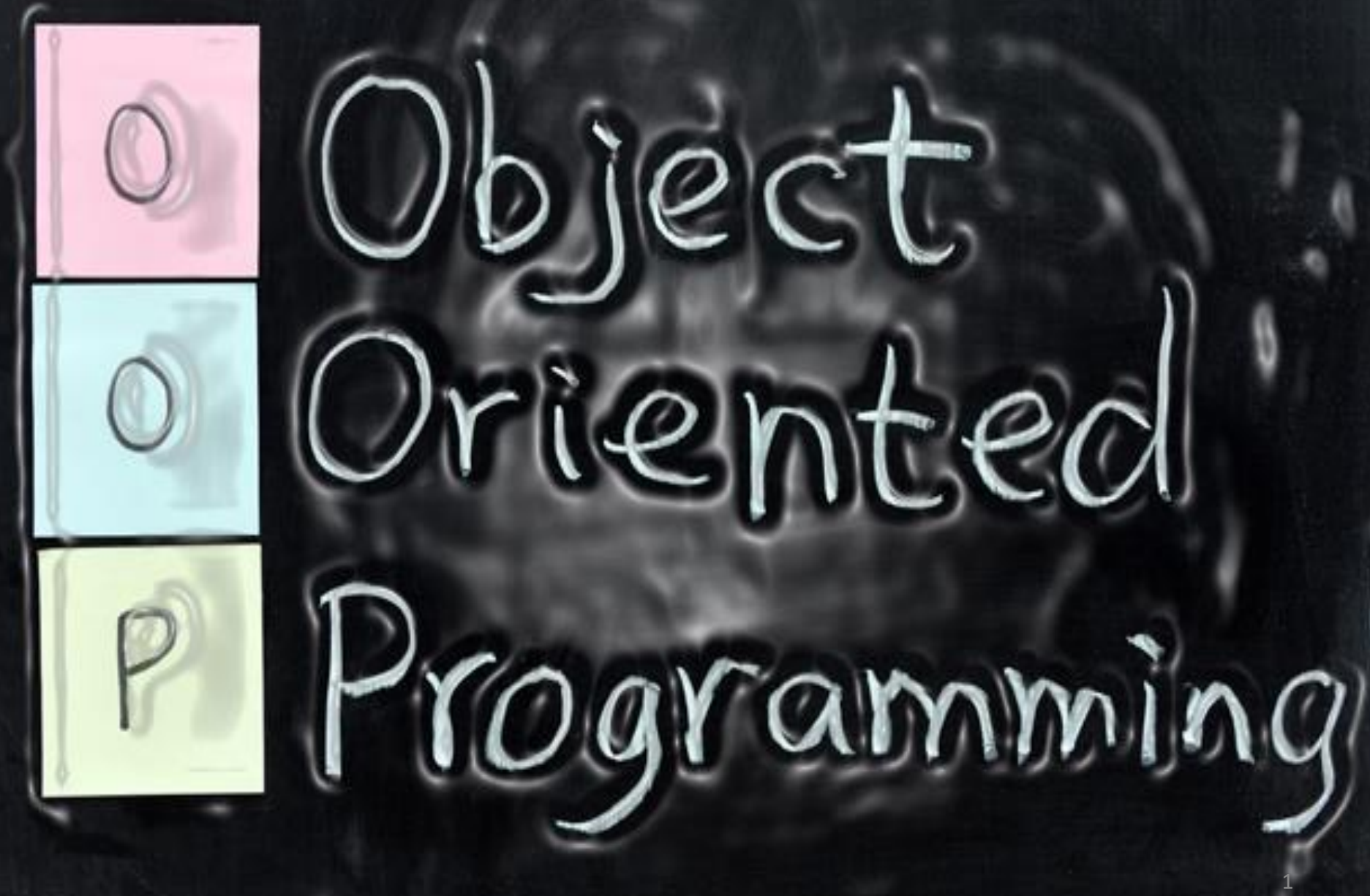

O
O
P
Object
Oriented
Programming

Objectives of This Lecture

- Review the pertinent nature of hardware.
- Discuss abstraction in computing science.
- Discuss the historical context – the S/W Crisis.
- Introduce Object-Oriented programming.
- Mention the shortcomings of OOP.

I really hate this darn machine;
I wish they would sell it;
It won't do what I want it to,
but only what I tell it.

Anonymous Programmer's Lament

Preamble

Question

What can the architecture of contemporary computer hardware tell us about the fundamental nature of programming such machines?

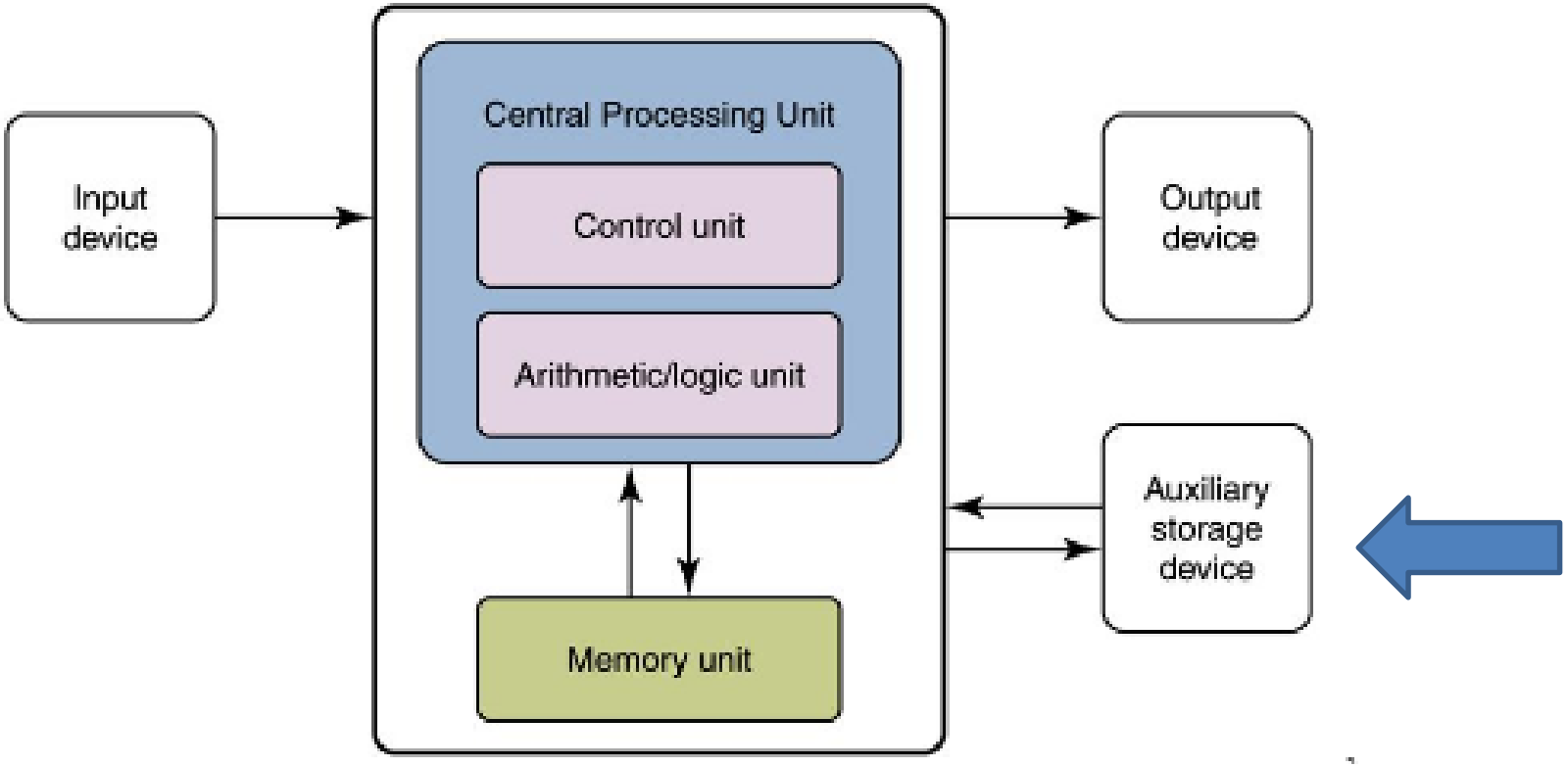
The Core of Contemporary Computing: The Fundamental Nature of the Machine

Von Neumann Architecture

A model for designing and building computers that is based on the following three characteristics:

1. Four major subsystems called MEMORY, Input/Output (I/O), Arithmetic/Logic Unit (ALU), and the Control Unit (CU) – note that ALU+CU=CPU.
2. The STORED PROGRAM CONCEPT
3. SEQUENTIAL EXECUTION OF INSTRUCTIONS

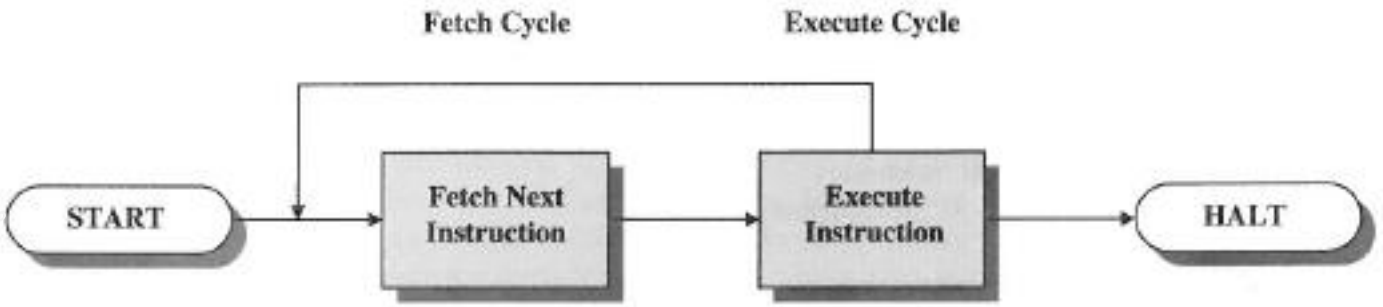
The Stored Program Concept



Sequential Execution of Instructions

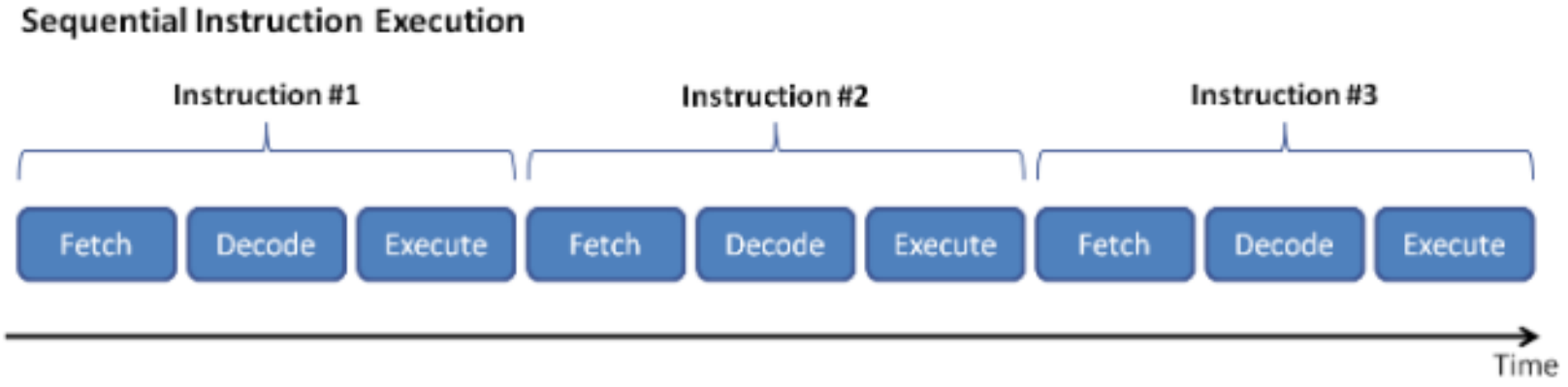
Definition: Instruction Cycle

- The fundamental cycle between memory and CPU for Von Neumann architecture (*a.k.a.*, fetch/decode/execute cycle).
- It consists of the following steps:
  - fetching an instruction from memory,
  - decoding the instruction (not shown in the figure below),
  - executing the instruction.

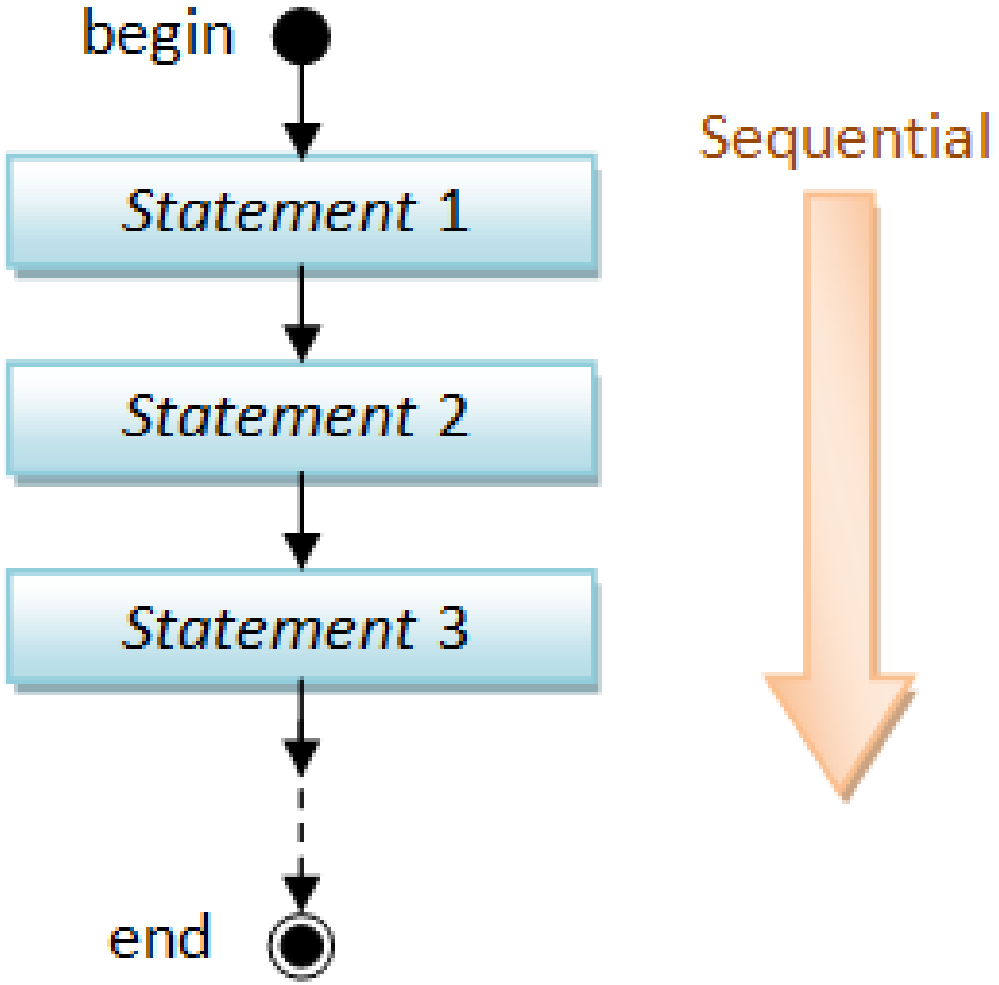


Preliminary Conclusions

- The most basic programming language is purely sequential – step by step execution of instructions describing exactly what to do.
- Accordingly, **verbs** are at the heart of computing from a linguistic perspective (a verb is a word used to describe an action, state, or occurrence).



The Sequential Programming Model



The Sequential Programming Model

The original notion of computer programming focused on a strict sequence of instructions much like a kitchen recipe like this example of assembly code with statement numbers:

```
001        i12 = _y;
002        i4 = _x;
003
004        lcntr = 20, do (pc,4) until lce;
005          f2 = dm(i4,m6);
006          f4 = pm(i12,m14);
007          f8 = f2*f4;
008          f12 = f8 + f12;
009
010        dm(_result) = f12;
```

Definition: Procedural Programming

- **Procedural programming** was an early approach to programming based on the adjective, *procedural*, that delineating the steps necessary to achieve a goal state, where the control flows sequentially from one statement to the next (verb after verb, action after action, state after state), but with occasional jumps and loops handled with *goto* statements.
- The notion of Procedure Calls, Subroutine Calls and Function Calls were introduced, providing an early attempt to modularize computer programs.

Notes on Grammar

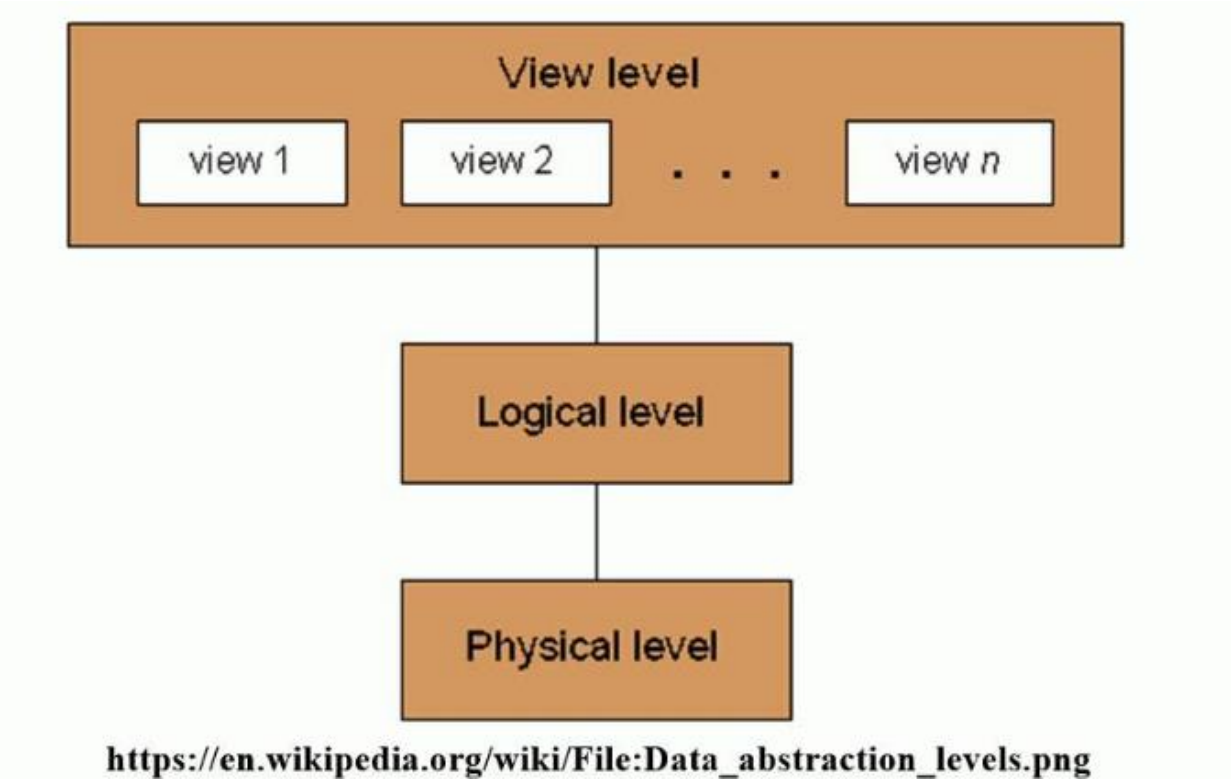
- **Verb**: a word used to describe an action, state, or occurrence, and forming the main part of the predicate of a sentence;
  - *i.e.,* do, print, link, compile, execute, *etc*.
- **Noun**: a word (other than a pronoun) used to identify any of a *class* of people, places, or things;
  - *i.e.,* personal "computer", a "computer" error, *etc*.
- **Adjective**: A word or phrase naming an attribute, added to or grammatically related to a noun to modify or describe it;
  - *i.e., "*Procedural" programming.
- **Caveat**: Some words are both verbs and nouns;
  - *i.e.,* drink your drink.

End Preamble


Abstraction

Abstraction

- Abstraction hides unnecessary details for some immediate purpose.
- In other words, abstraction is the process of finding similarities or common aspects, and forgetting about unimportant differences.



https://en.wikipedia.org/wiki/File:Data_abstraction_levels.png

Abstraction

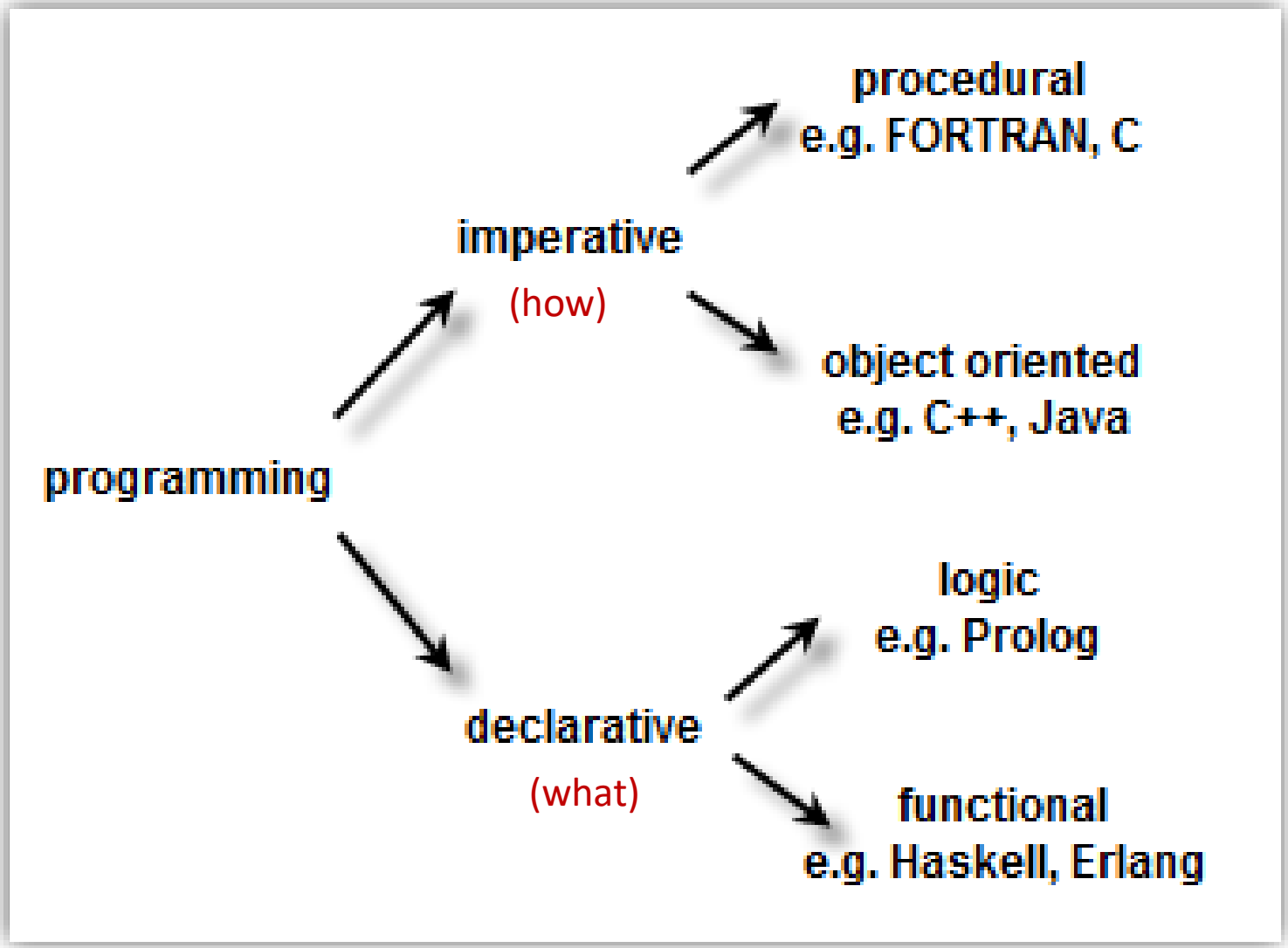
- From the preamble, a program is a set of sequential commands explicitly directing the computer on *how* to solve a problem.
- However, it is not difficult to envisage a programming abstraction that hides the details of how to solve a problem, focusing instead on *what* must be done.

Abstraction

The two fundamental programming "paradigms":

- ***Imperative*** programming is an abstraction that focusses on *how* to do what you are trying to do (commands). This is the most natural and, hence, most common paradigm .
- ***Declarative*** programming is an abstraction that focusses on *what* to do. How the algorithm is actually executed is hidden (abstracted away) from the programmer. Specifically, the compiler determines *how* to accomplish *what* must be done.

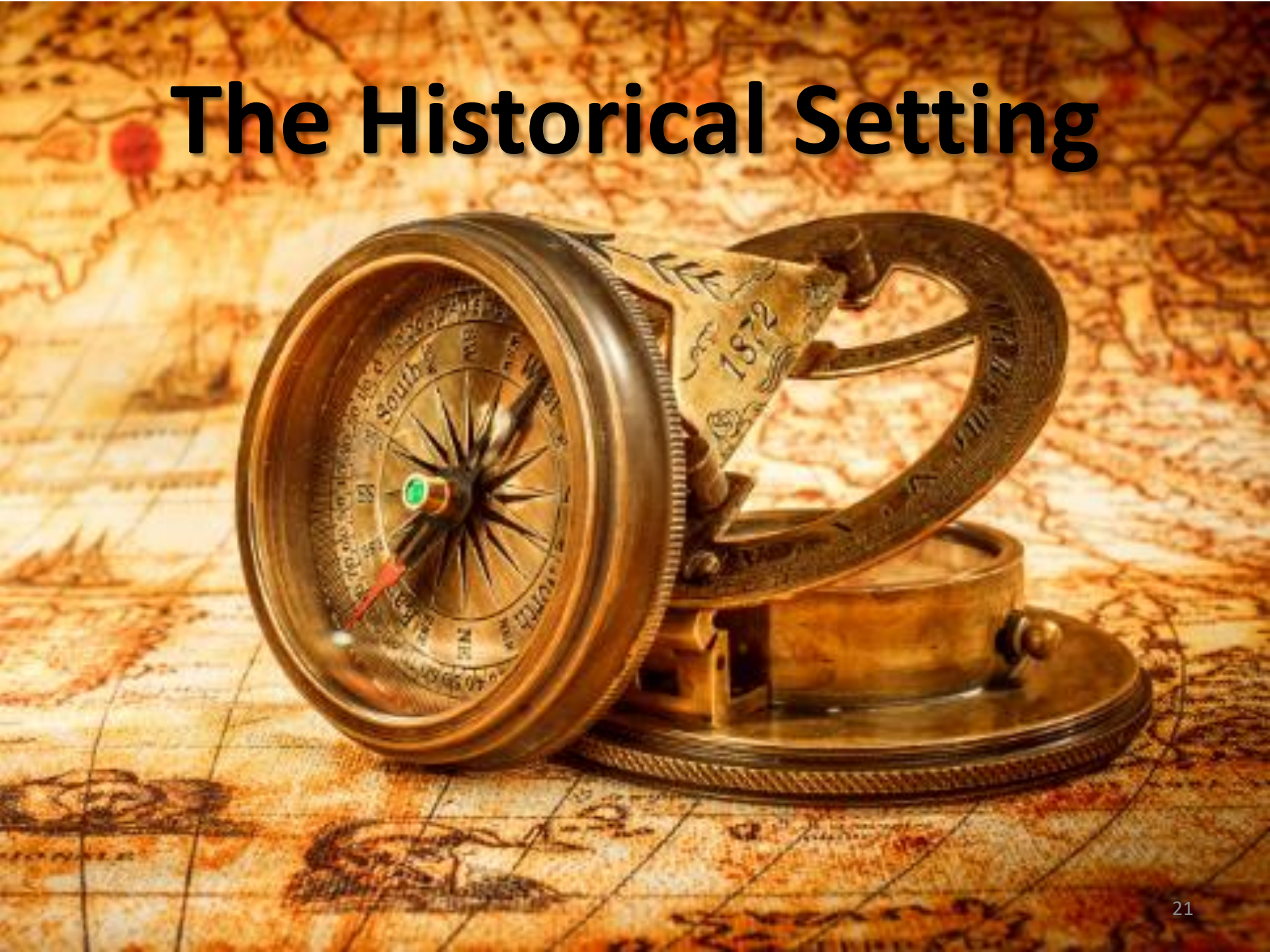
Abstraction



Other Computing Abstractions

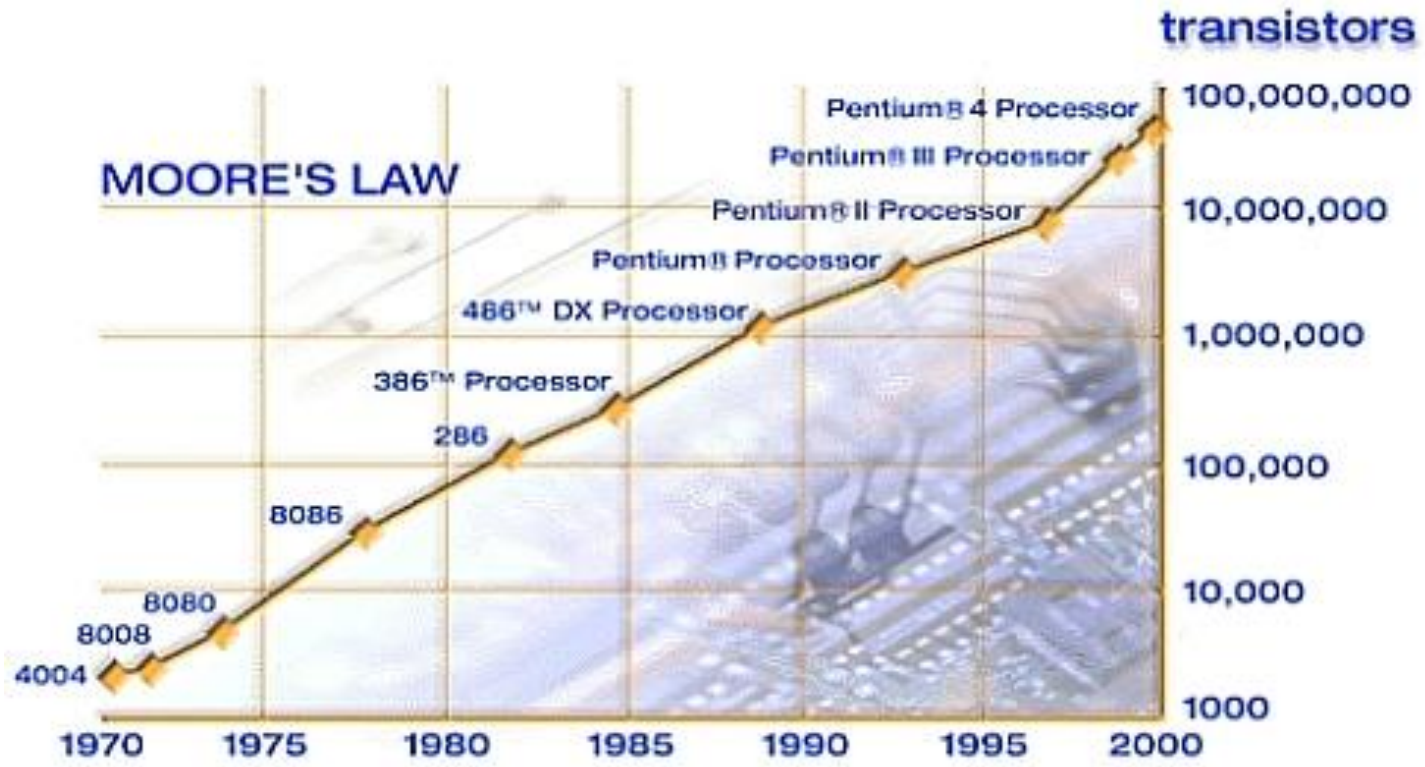
- **Functional programming** is a further abstraction within the declarative paradigm that treats a computation as the evaluation of mathematical functions rather than focusing on the changing state of a program.
- **Object-Oriented programming** is a further abstraction within the imperative paradigm that abstracts away the *actions* of programming, allowing the programmer to focus on real-world **objects** as the name implies (focusing on **nouns** instead of verbs).

The Historical Setting



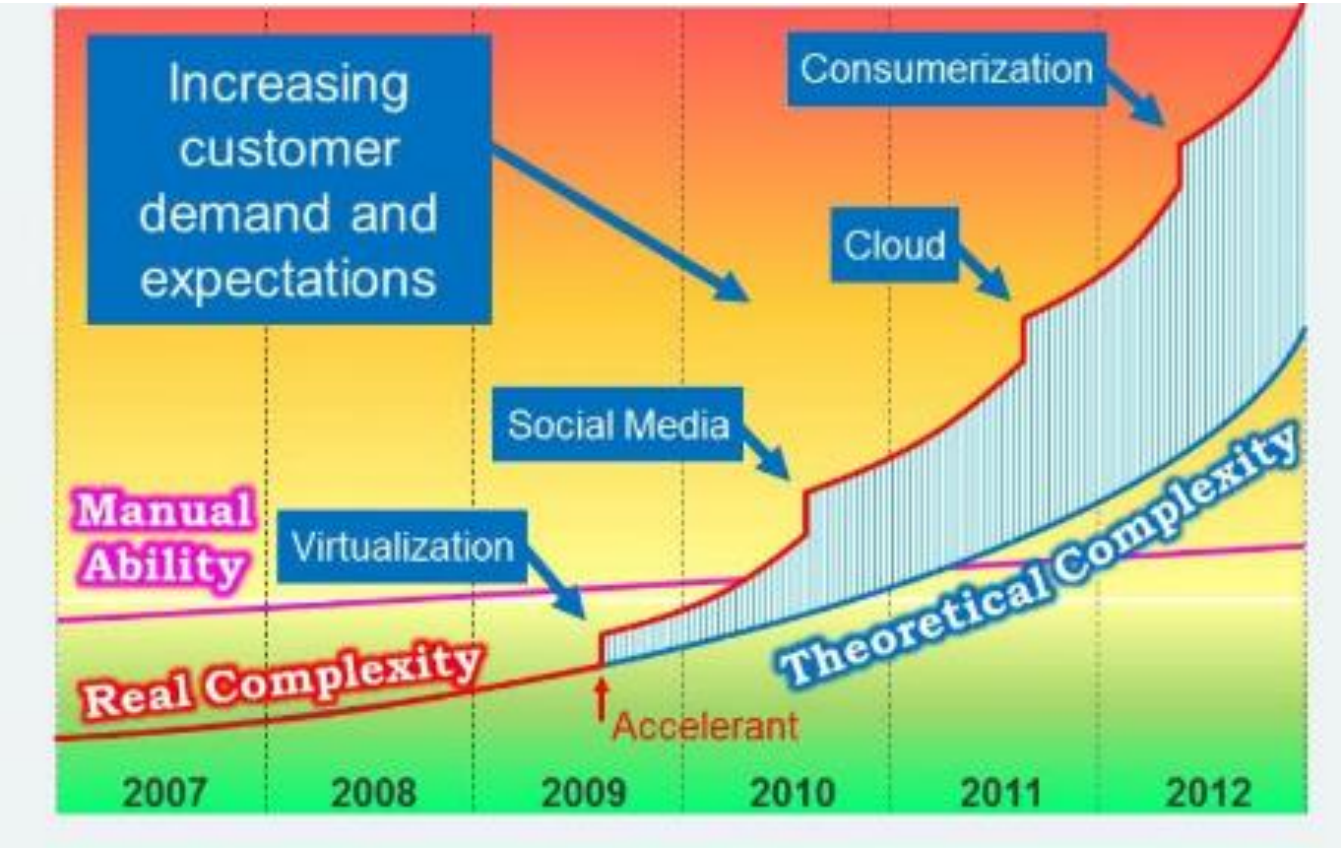
Some Facts

Computer power and memory have grown according to Moore's Law – doubling every 18 to 24 months.



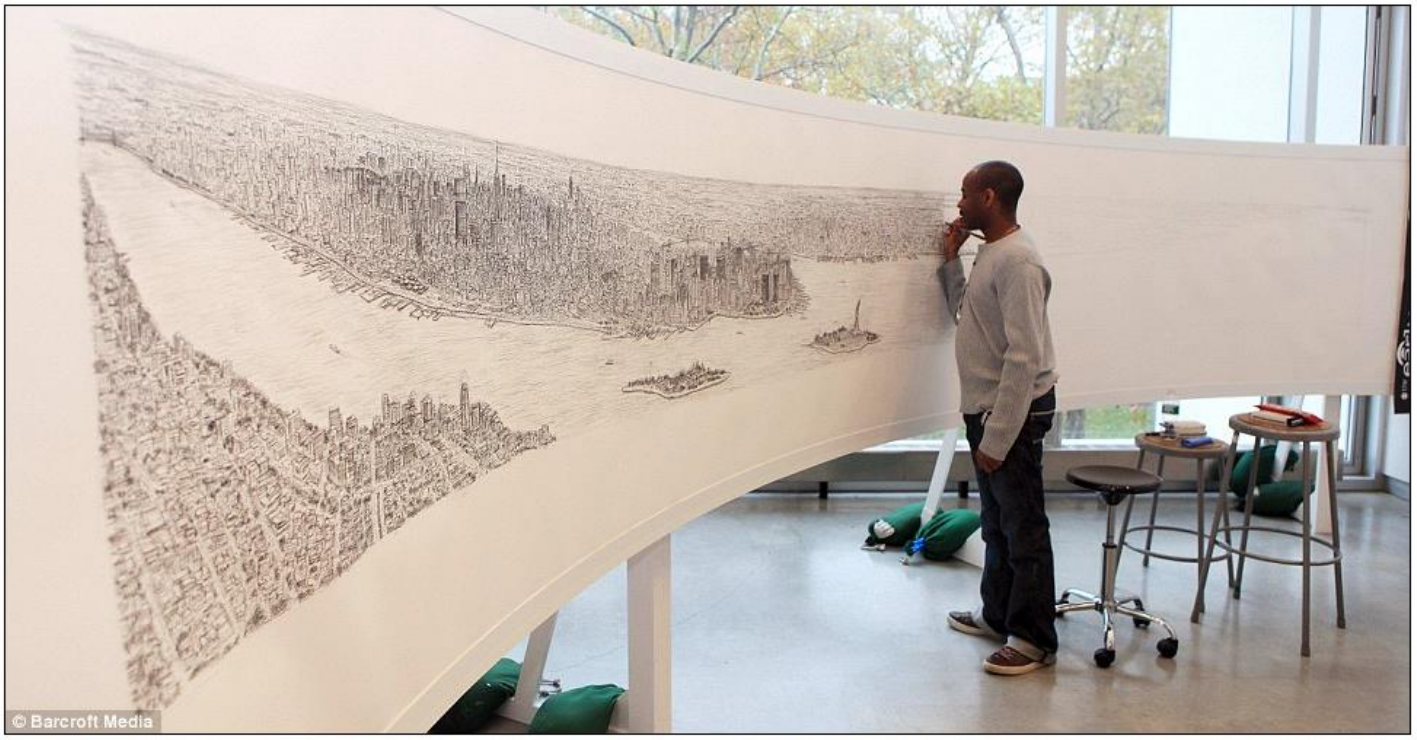
Some Facts

Accordingly, software has continually become larger and more complex.



Some Facts

The human mind did not evolve to deal directly with complexity like a computer can – we naturally "abstract away" most of what we sense, focusing only on that which optimizes our survival.



https://www.dailymail.co.uk/news/article-1223790/Autistic-artist-draws-18ft-picture-New-York-skyline-memory.html

Some Facts

- Procedural programming focuses on the *state* of the program – memory of preceding events and information about the system.
- This methodology by itself produces code that becomes more difficult to read and maintain the larger the problem becomes.
- Such code is referred to as *spaghetti code*.

Consequences

- This led to the so-called **software crisis** – a term coined by some attendees at the first NATO Software Engineering Conference in 1968 at Garmisch, Germany.
- New practices were developed to circumvent the crisis and this all culminated in the development of so-called *software engineering*.
- The first bold attempt was introduced via the *Structured Program Theorem*.

Structured program theorem

From Wikipedia, the free encyclopedia

The **structured program theorem**, also called **Böhm-Jacopini theorem**,[1][2] is a result in programming language theory. It states that a class of control flow graphs (historically called charts in this context) can compute any computable function if it combines subprograms in only three specific ways (control structures). These are

1. Executing one subprogram, and then another subprogram (sequence)
2. Executing one of two subprograms according to the value of a boolean expression (selection)
3. Executing a subprogram as long as a boolean expression is true (iteration)

Structured program theorem

From Wikipedia, the free encyclopedia

The **structured program theorem**, also called **Böhm-Jacopini theorem**,[1][2] is a result in programming language theory. It states that a class of control flow graphs (historically called charts in this context) can compute any computable function if it combines subprograms in only three specific ways (control structures). These are

1. Executing one subprogram, and then another subprogram (sequence)
2. Executing one of two subprograms according to the value of a boolean expression (selection)
3. Executing a subprogram as long as a boolean expression is true (iteration)

Simply put, these three control flow elements are sufficient for computing any solvable problem.

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been 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. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete

dynamic progress is only characterized when
call of the procedure we refer. With the in
we can characterize the progress of the proc
textual indices, the length of this sequenc
dynamic depth of procedure calling.

Let us now consider repetition clauses (lik
or **repeat** *A* **until** *B*). Logically speaking,
superfluous, because we can express repeti
recursive procedures. For reasons of realism
clude them: on the one hand, repetition
mented quite comfortably with present day
the other hand, the reasoning pattern ki
makes us well equipped to retain our intel
processes generated by repetition clauses.
the repetition clauses textual indices are
describe the dynamic progress of the process.
a repetition clause, however, we can associ
namic index," inexorably counting the or
corresponding current repetition. As repeti
procedure calls) may be applied nestedly,
progress of the process can always be uniqu
(mixed) sequence of textual and/or dynamic

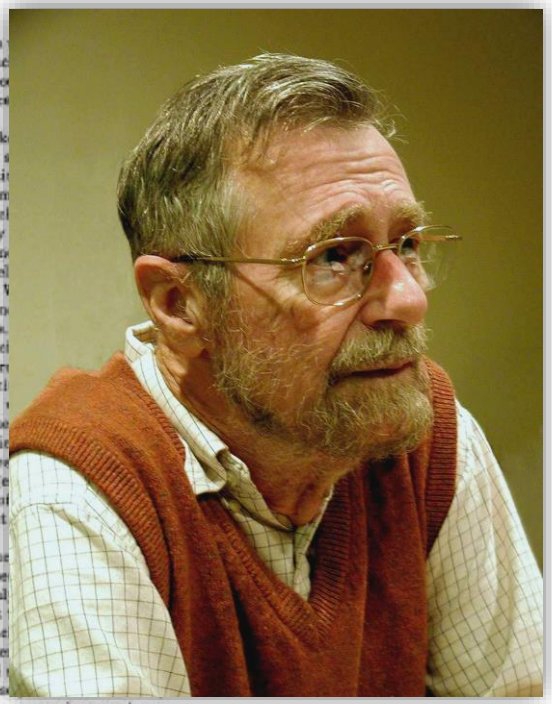
The main point is that the values of the
programmer's control; they are generated (e
of his program or by the dynamic evolution o
he wishes or not. They provide independent
to describe the progress of the process.

Why do we need such independent coo
is—and this seems to be inherent to seque
we can interpret the value of a variable onl
progress of the process. If we wish to count
people in an initially empty room, we can se
ing *n* by one whenever we see someone ente
in-between moment that we have observed
room but have not yet performed the subs
its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

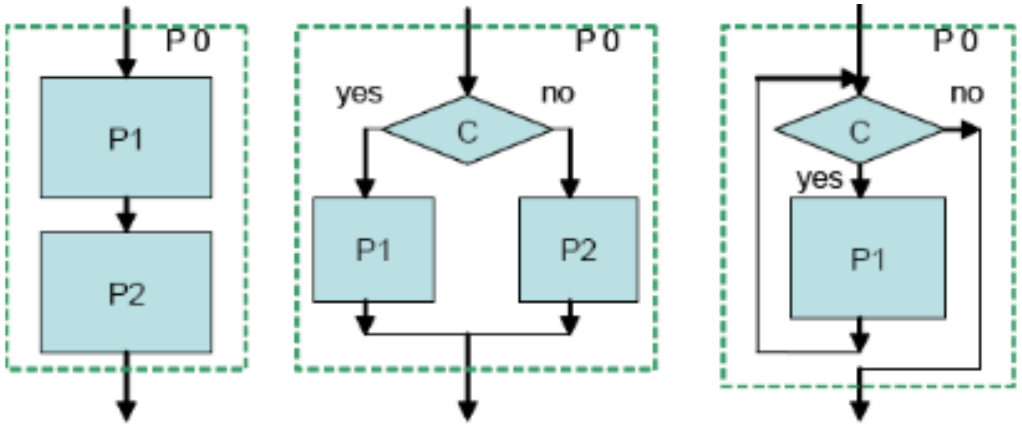
The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to



Structured Programming

Structured programming is a fundamental approach to programming aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines.



Program Design Strategies: Top-Down Versus Bottom-Up

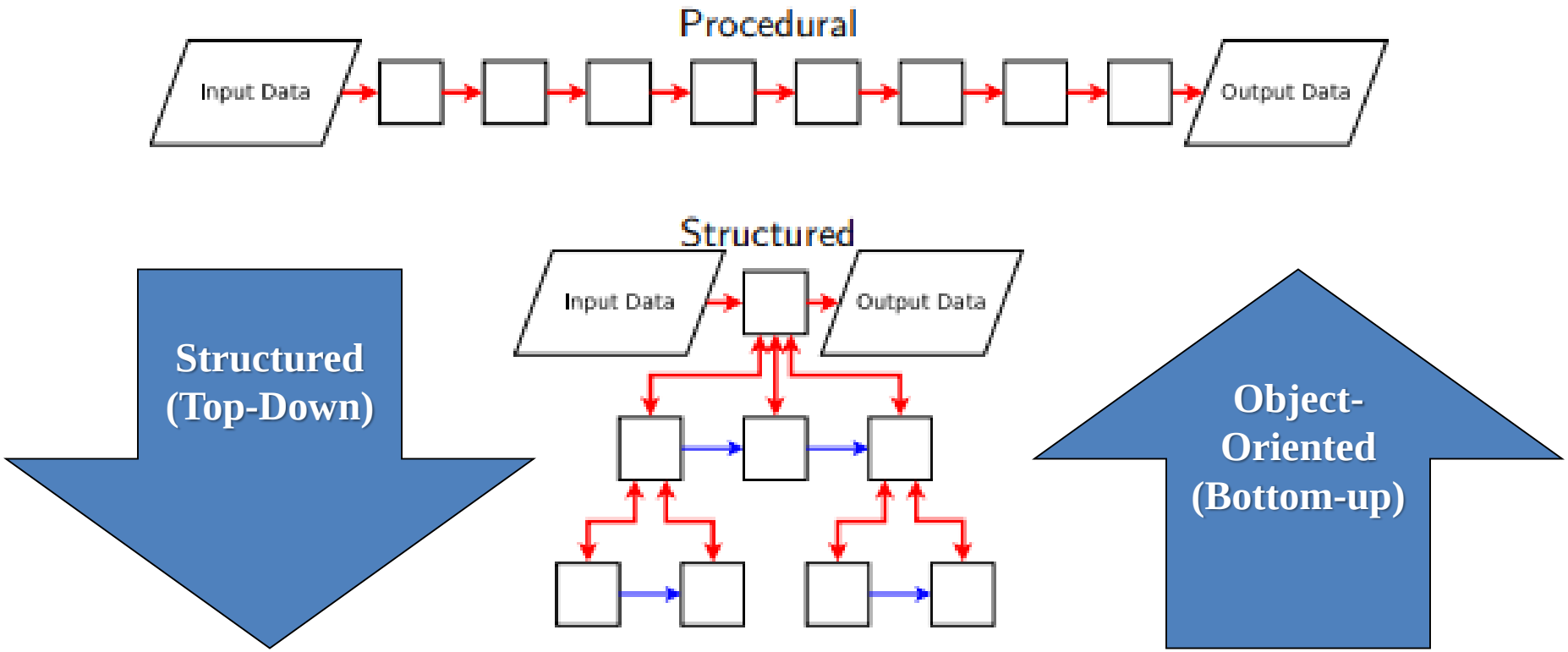
- The *top-down* approach takes a high-level definition of the problem and subdivides it into subproblems which are obvious and easy to code.
- In *bottom-up* programming, you identify lower-level elements and organize them into the overall high-level problem.
- Structured programming is naturally suited to be modular and top-down programming.
- OOP *appears* at first glance to be bottom-up since identifying objects is like identifying low-level modules.
- In reality, almost all programming is done with a combination of these fundamental strategies. In object oriented programming, you commonly subdivide the problem by identifying domain objects (top-down), refining and recombining them into the final program — bottom-up.

Note: Some OO Terminology

State: A program is described as stateful if it is designed to remember preceding events or user interactions; the remembered information is called the state of the system. (*Wikipedia*)

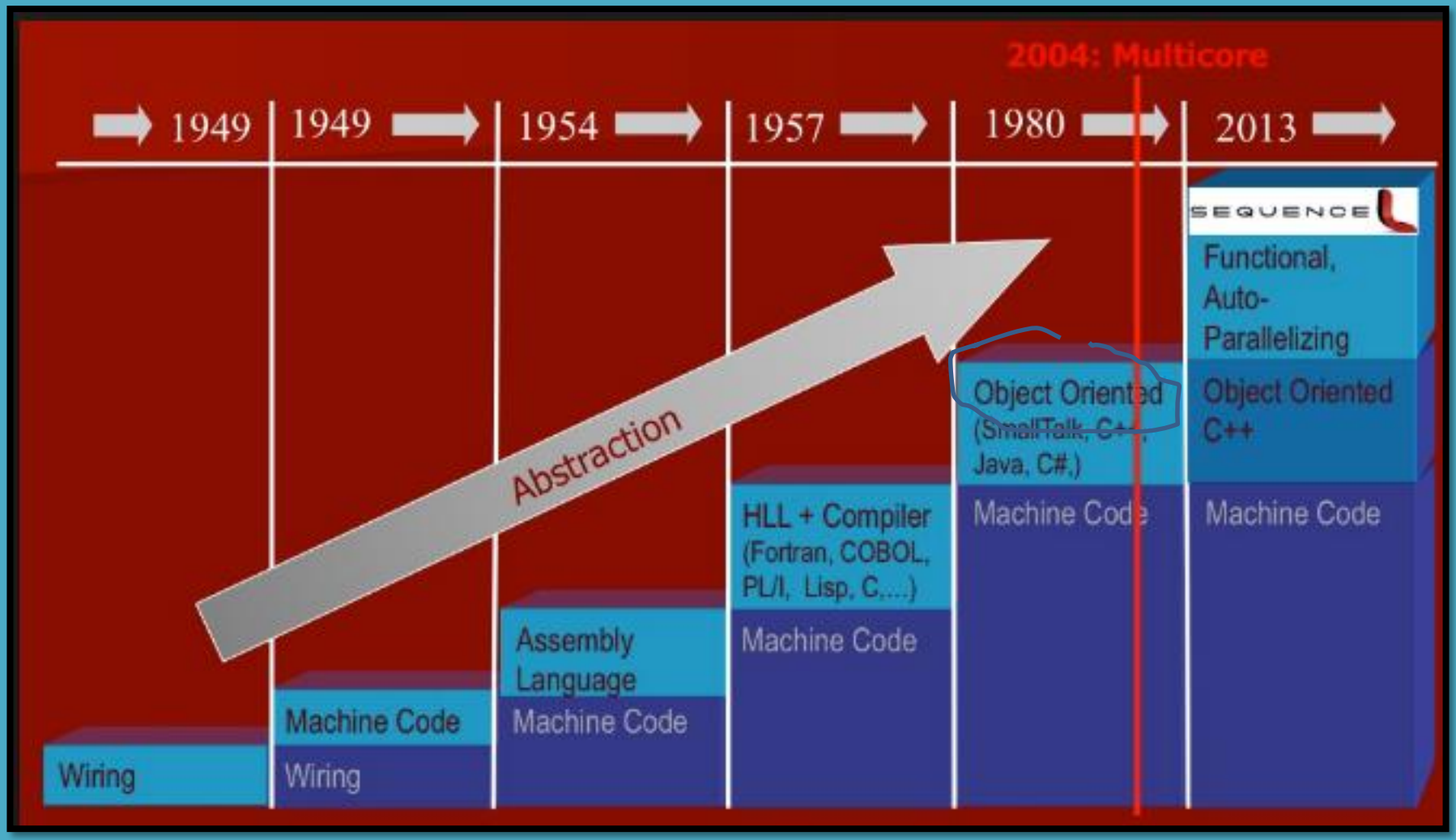
Domain: The domain of an application under development is the sphere of knowledge and activity around which the application knowledge revolves. 

Naïve Summary

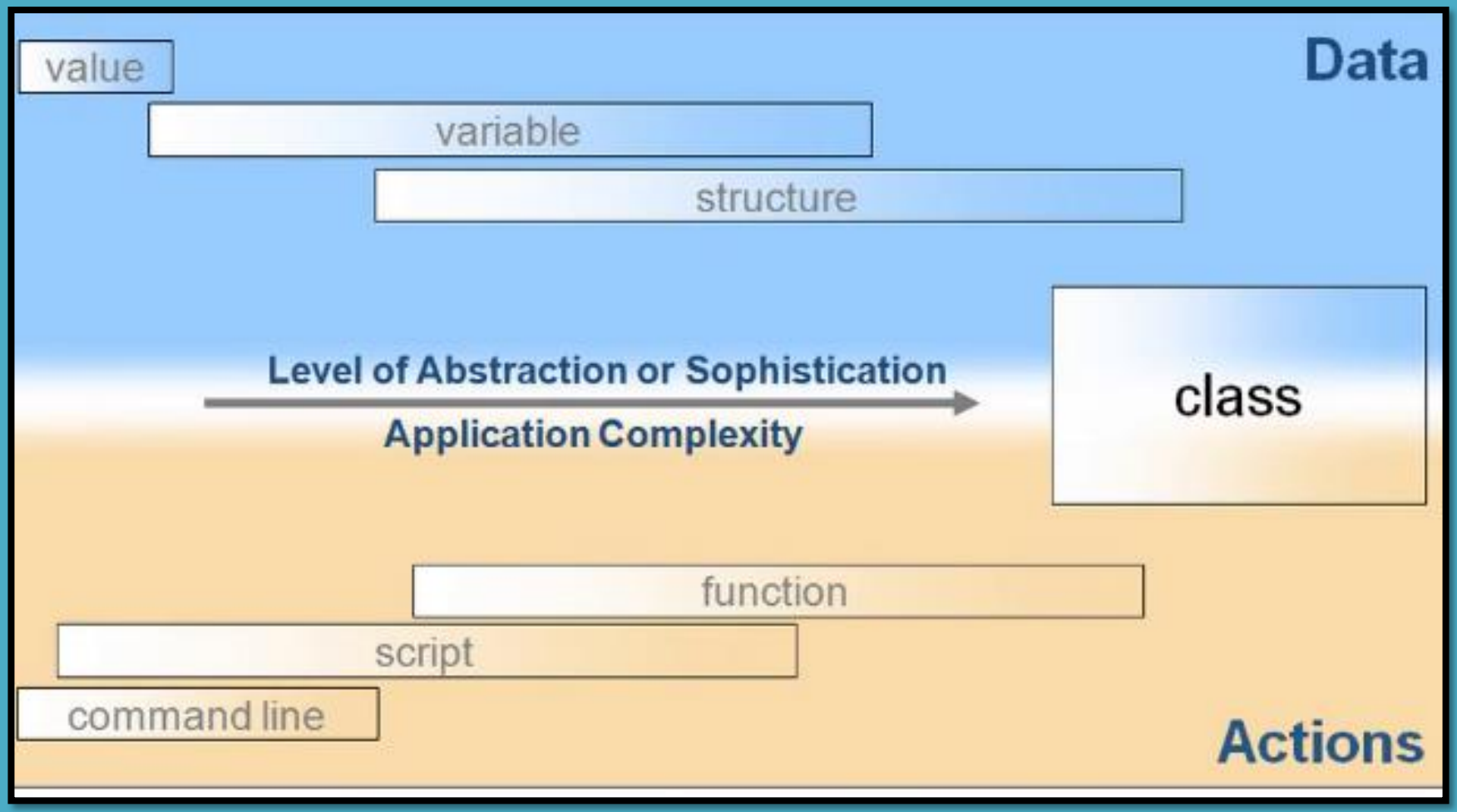


https://wiki.scinet.utoronto.ca/wiki/images/4/49/Scientific-c%2B%2B.pdf

Summary of Evolving Abstractions



Evolving Focus of Abstractions

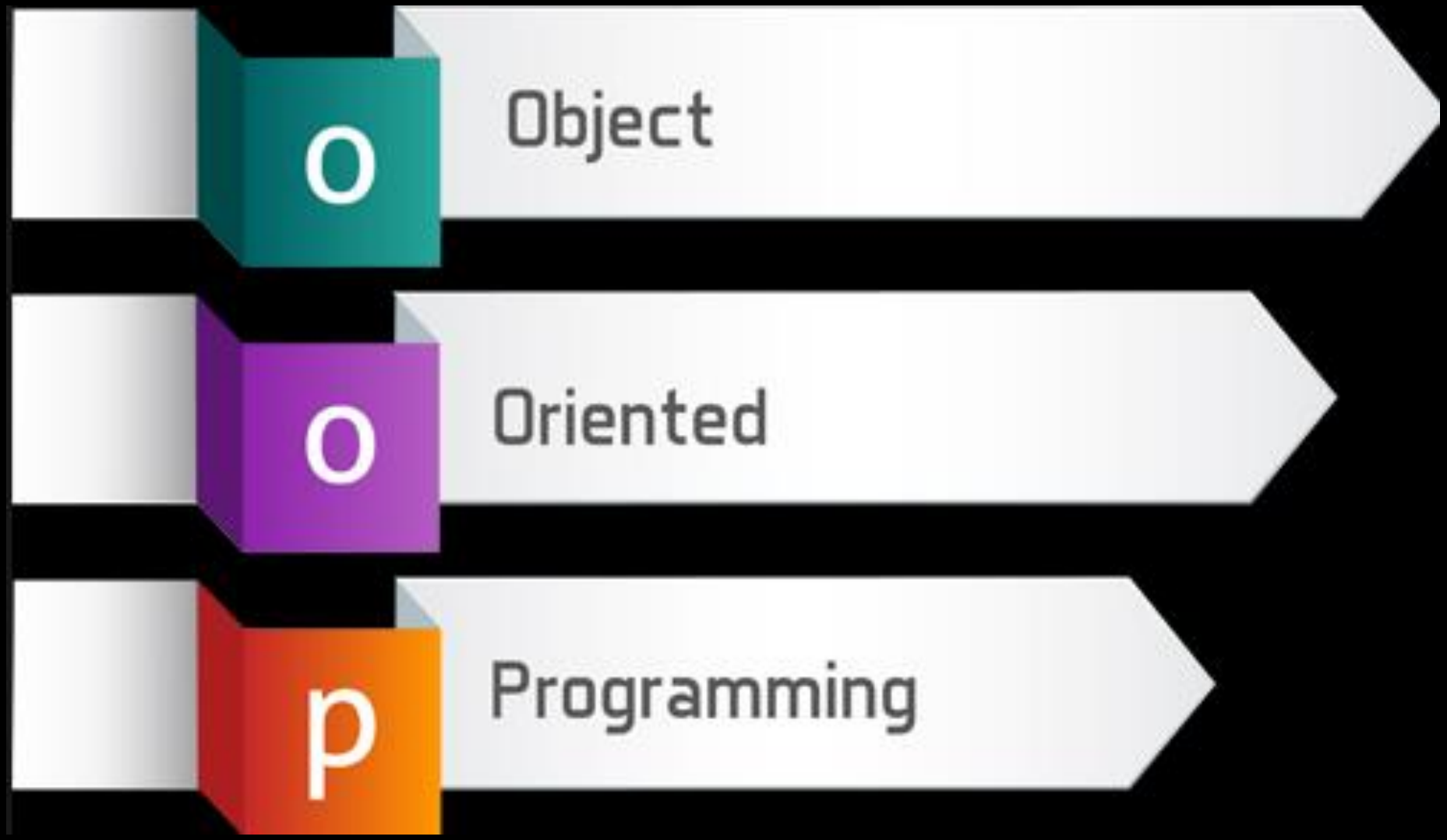


The Historical Incentive for OOP

- Again, even the structured programming approach begins to show signs of strain – weaknesses remain in the procedural/structured programming methodologies.
- In procedural programming, a program consists of data and modules/procedures that operate on the data. The two are treated as separate entities
- Also, unrelated functions and data, the basis of the procedural paradigm, provide a poor model of complex real world problems.
- OOP was inspired by a set of “good coding practices” that avoid these problems.

Preliminary Summary

- We reviewed the pertinent aspects of von Neumann computer architecture – that the fundamental computer program is **sequential** in nature.
- We discussed the importance of **abstraction**.
- We introduced **structured programming** constructs.
- We briefly introduced the two fundamental programming strategies – **top-down** and **bottom-up**.
- We reviewed the historical context which gave rise to the **object-oriented** abstraction in programming.

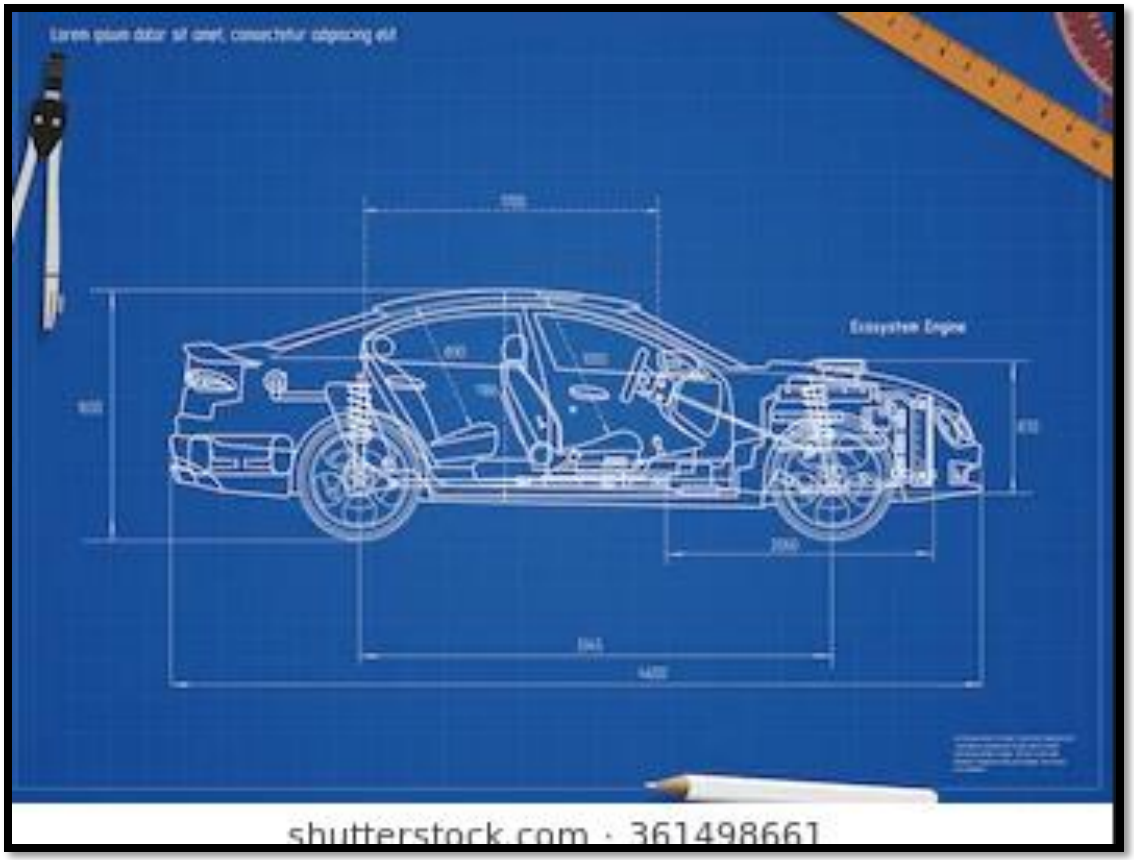

o
Object
o
Oriented
p
Programming

The Object-Oriented Approach

- In the object-oriented programming (OOP) methodology, a program is built from **objects**.
- An object is an instance of a **class**, which is an encapsulation of data (called **fields**) and the procedures (called **methods**) that manipulate them.
- In most, but not all, cases, the fields can only be accessed or modified through the methods.
- Thus, an object is like a miniature program or a self-contained component, which makes the OOP approach more modularizable and, consequently, easier to maintain, extend and model real-world situations.

The Object-Oriented Approach

A *class* is like a blueprint for a car from which various models with different attributes can be “instantiated”…



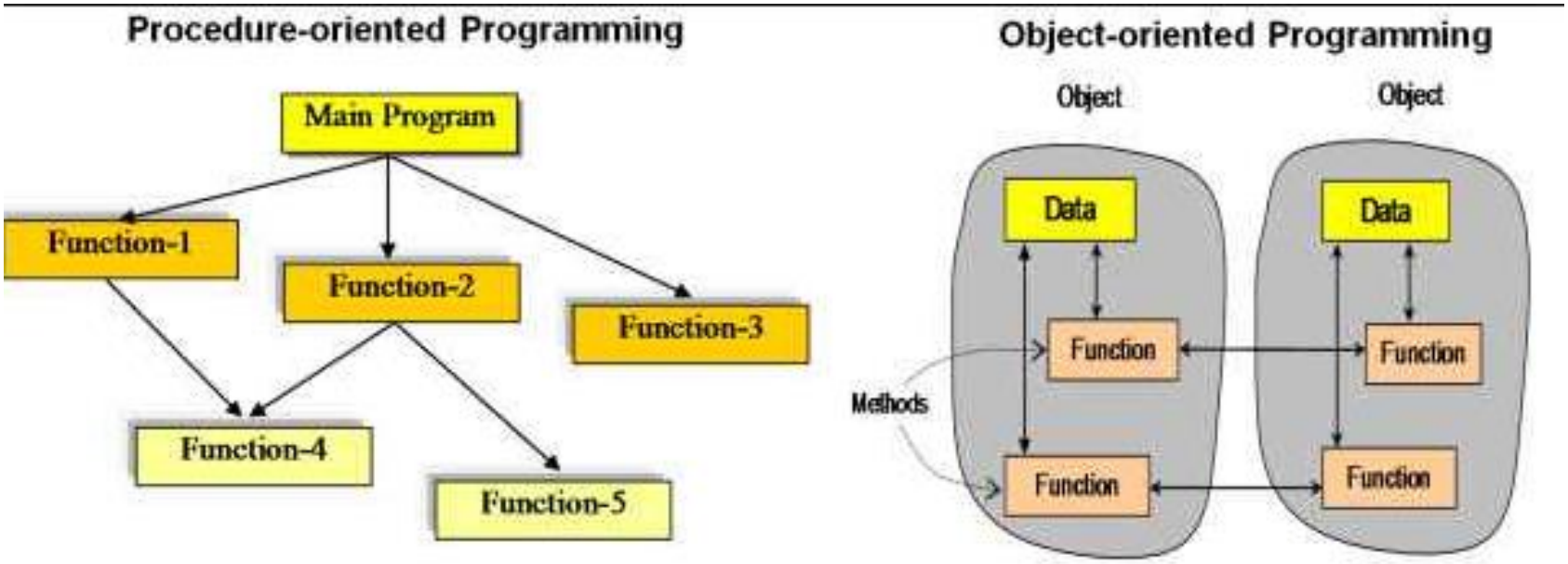
The Object-Oriented Approach

```
public class Car{
    private string _color;
    private string _model;
    private string _makeYear;
    private string _fuelType;

    public void Start(){
        ..
    }

    public void Stop(){
        ..
    }

    public void Accelerate(){
        ..
    }
}
```

DEV

PP/SP vs OOP

Object-oriented programming extends structured programming by combining data and the methods that operate upon that data into a single entity – an object.



The OO Abstraction

- In old style programming (procedural), you had:
  - data, which was completely passive, and
  - functions, which could manipulate any data.
- "Object Oriented" is actually a further abstraction along the Imperative Paradigm branch – abstracting the idea of objects beyond simple procedural thinking.
- An object contains both data and methods that manipulate that data
  - An object is *active,* not passive; it *does* things
  - An object is *responsible* for its own data
    - But: it can *expose* that data to other objects

Main Idea Underlying OOP

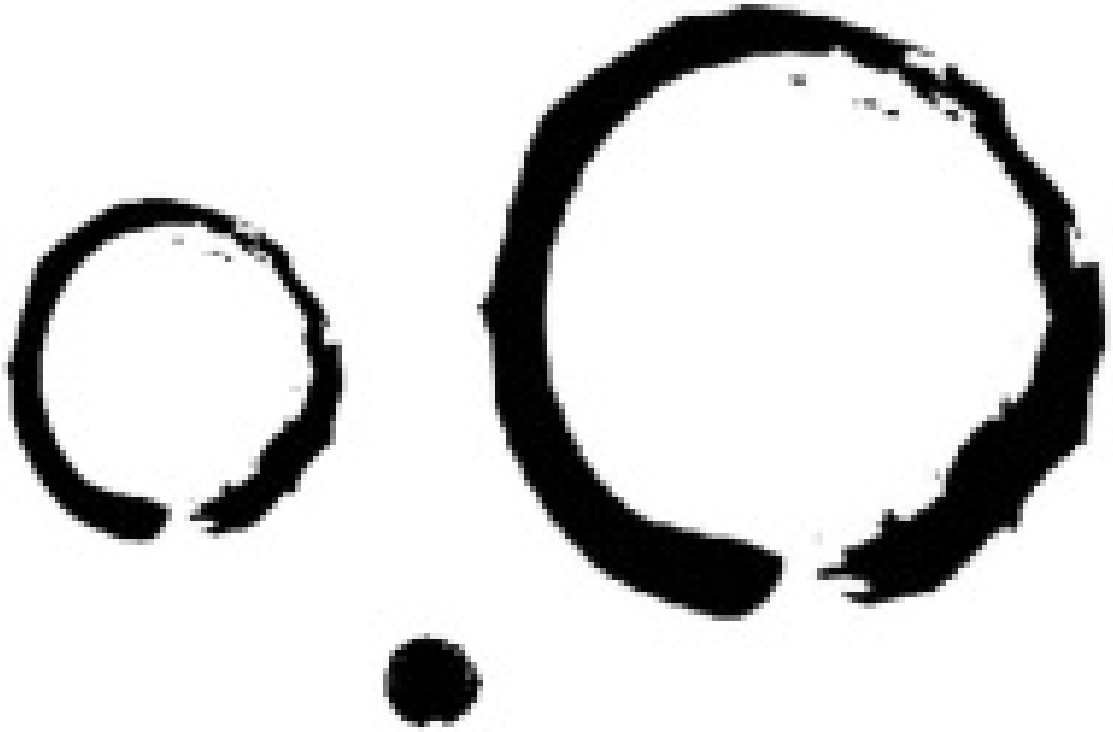
- Programmers code using “blueprints” of data models called ***classes*** which forms the basis of the following key concepts:
  - **Objects** – A unique programming entity that has *methods*, has *attributes* and can react to *events*.
  - **Methods** – Things which an object can do; the “verbs” of objects. In code, it can usually be identified by an “action” word -- *Hide*, *Show.*
- Recall that objects are nouns and methods are verbs.

Example

- The “*hand*” is a class.
- Your body has two objects of the type "*hand*", named "left hand" and "right hand".
- Their main functions are controlled or managed by a set of electrical signals sent through your shoulders (through an interface).
- So the shoulder is an interface that your body uses to interact with your hands.
- The hand is a well-architected class.
- The hand is being reused to create the left hand and the right hand by slightly changing the properties of it.

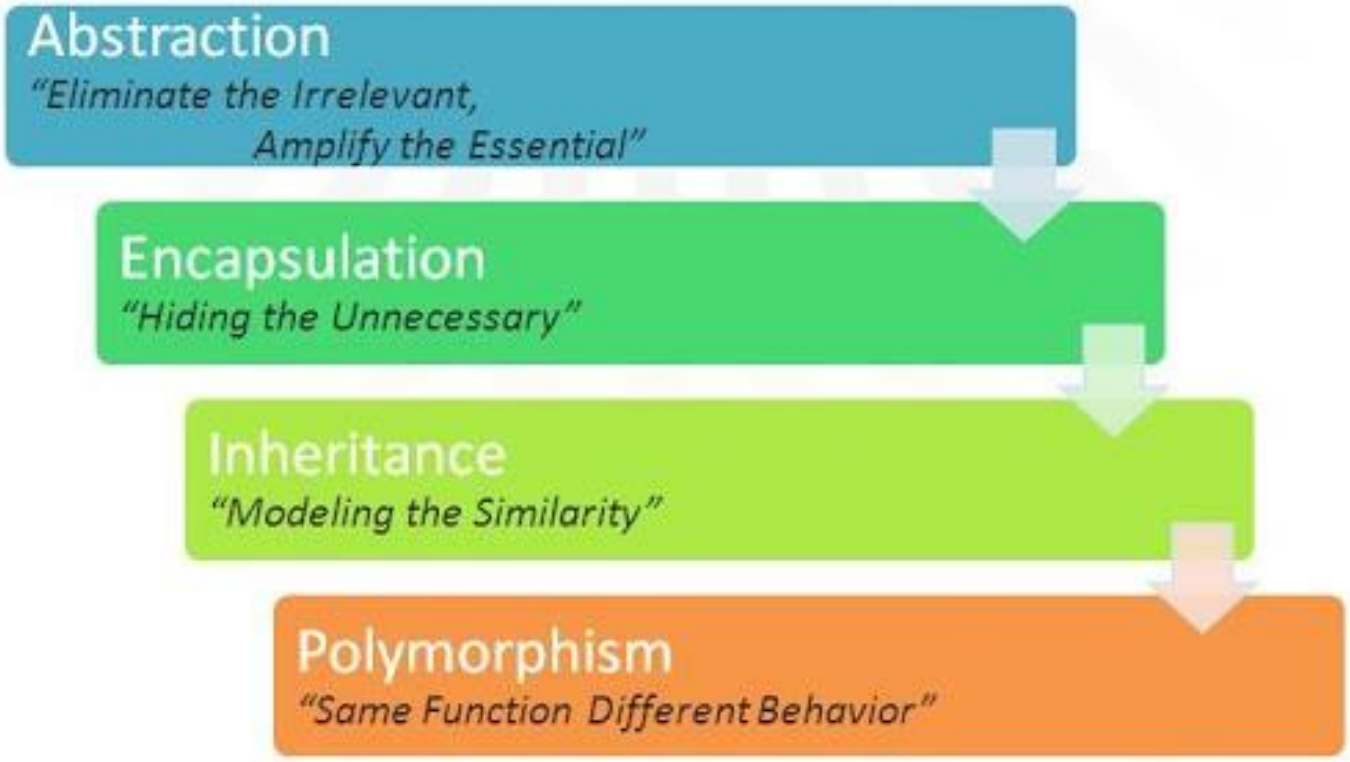
https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep

Key Elements of OO



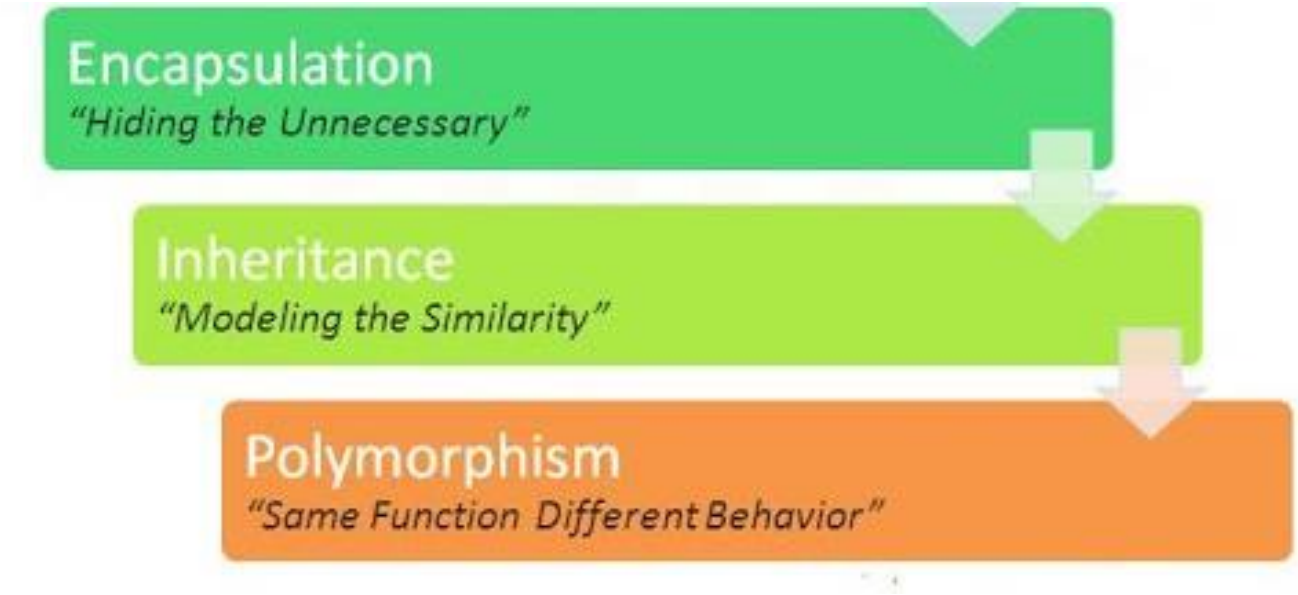
Key Elements of OO

Object-oriented methodology relies on four main characteristics that define object-oriented languages:



Key Elements of OO

Abstraction underlies all of these principles so, in effect, there are really only these three characteristics of OO which are most often referred to:



Key Elements of OO

Object-oriented methodology relies on four main characteristics that define object-oriented languages:

REMARK: This is *nearly* universally accepted by proponents of OOP.

A Clock Analogy

- We think of the clock as object with all of its familiar self-contained functionality and whose purpose is simply to tell time, hiding the details inside the box. Similarly, in programming you put all the code and data all together – that is what is meant by *encapsulation*.
- This is the practice of keeping fields within a class private, then providing access to them via public methods. It's a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide.

A Clock Analogy

- Now you've got a clock thing, you might want an alarm clock - well, once you've got all the required components together you can add things to it to make it do more - like set the alarm and make it ring – this is what is meant by *inheritance.*
- This feature allows one to build on previous work without reinventing the wheel.

A Clock Analogy

- Also, there are other clocks that look different like a grandfather clock or a digital clock. It appears different, but it's still a clock - that's called *polymorphism*.
- This OOP concept lets programmers use the same word to mean different things in different contexts.
- Technically speaking, it is the ability to redefine methods for derived classes.

What is the main difference between Inheritance and Polymorphism?

129

I was presented with this question in an end of module open book exam today and found myself lost. I was reading `Head first Java` and both definitions seemed to be exactly the same. I was just wondering what the MAIN difference was for my own piece of mind. I know there are a number of similar questions to this but, none I have seen which provide a definitive answer.

java oop inheritance polymorphism

93

share improve this question

edited Mar 11 '15 at 22:32
ROMANIA_engineer
31.5k ● 18 ● 138 ● 137

asked Jun 10 '11 at 15:00
Darren Burgess
2,381 ● 5 ● 20 ● 39

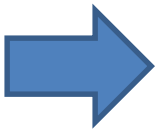
2 Somehow related to this question: Is polymorphism possible without inheritance – Edwin Dalorzo Aug 6 '12 at 13:22

2 Also related: Isn't polymorphism just a side effect of inheritance? – TMS Jun 13 '14 at 6:57

add a comment

15 Answers

active oldest votes



241

Inheritance is when a 'class' derives from an existing 'class'. So if you have a `Person` class, then you have a `Student` class that extends `Person`, `Student` *inherits* all the things that `Person` has. There are some details around the access modifiers you put on the fields/methods in Person, but that's the basic idea. For example, if you have a private field on `Person`, `Student` won't see it because its private, and private fields are not visible to subclasses.

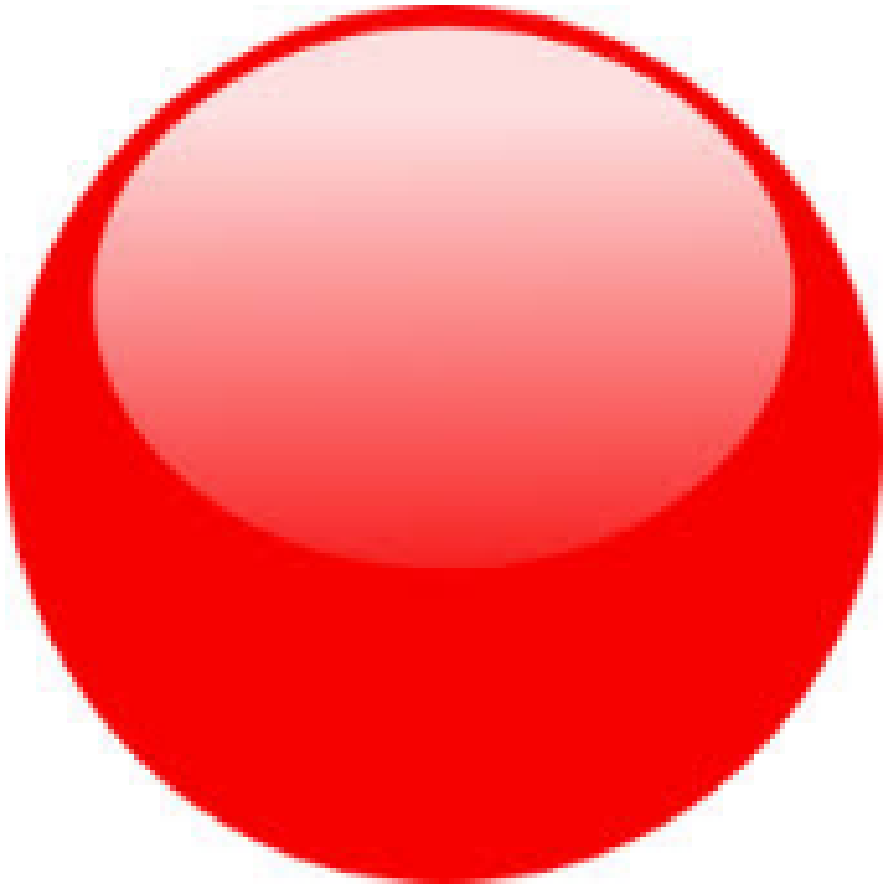
stackoverflow

Difference Between Inheritance and Polymorphism

May 11, 2016 — Leave a Comment

Inheritance allows, code reusability and the polymorphism is, the occurrence of one function with different form. The basic difference between inheritance and polymorphism is that inheritance allows the already existing code to be reused again in a program, and polymorphism provides a mechanism to dynamically decide what form of a function to be invoked.

The Dot Notation of OOP



The Dot Notation of OOP

Let us apply a particular syntactic standard to what we have learned already regarding OOP:

- Consider the following relationships:
  - Fido is a dog. During a typical day, he does various actions: he eats, runs, sleeps, etc.

http://reeborg.ca/docs/oop_py_en/oop.html

The Dot Notation of OOP

Let us apply a particular syntactic standard to what we have learned already regarding OOP:

- Consider the following relationships:
  - Fido is a dog. During a typical day, he does various actions: he eats, runs, sleeps, etc. *Here's how an object-oriented programmer might write this:*

```
Fido = Dog()
Fido.eats()
Fido.runs()
Fido.sleeps()
```


http://reeborg.ca/docs/oop_py_en/oop.html

The Dot Notation of OOP

- In addition, Fido has various qualities or attributes.
  - These are variables, like we have seen before except that they “belong” to Fido. He is tall (for a dog) and his hair is black. Here’s how the programmer might write the same things:

```
Fido.size = "tall";
Fido.hair_colour = "black";
```


http://reeborg.ca/docs/oop_py_en/oop.html

The Dot Notation of OOP

- In the object-oriented language, we have the following:

  - `Dog` is an example of a *class* of **objects**.
  - `Fido` is an **instance** (or particular object) in the Dog class.
  - `eats(), runs()` and `sleeps()` are **methods** of the Dog class; 'methods' are essentially like 'functions' which we saw before (the only difference is that they belong in a given class/object/instance).
  - `size` and `hair_colour` are attributes of a given instance/object; attributes can take any value that a "normal" variable can take.
  - The connection between the attributes or the methods with the object is indicated by a "dot" (".") written between them.

http://reeborg.ca/docs/oop_py_en/oop.html

The Dot Notation of OOP

Chaining:

- *Objects* can also have *other objects* that belong to them, each with their own *methods or attributes*:

```
Fido.tail.wags()
Fido.tail.type = "bushy";
Fido.left_front_paw.moves()
Fido.head.mouth.teeth.canine.hurts()
```


http://reeborg.ca/docs/oop_py_en/oop.html

In Summary:
The Proposed Purpose of OO

Object-Oriented Programming: Making Complicated Concepts Simple

Object-oriented (OO) programming is a programming paradigm that includes or relies on the concept of *objects*, encapsulated data structures that have properties and functions and which interact with other objects.

Objects in a program frequently represent real-world objects — for example, an ecommerce website application might have `Customer` objects, `Shopping_cart` objects, and `Product` objects. Other objects might be loosely related to real-world equivalents — like a `Payment_processor` or `Login_form`. Many other objects serve application logic and have no direct real-world parallel — objects that manage authentication, templating, request handling, or any of the other myriad features needed for a working application.

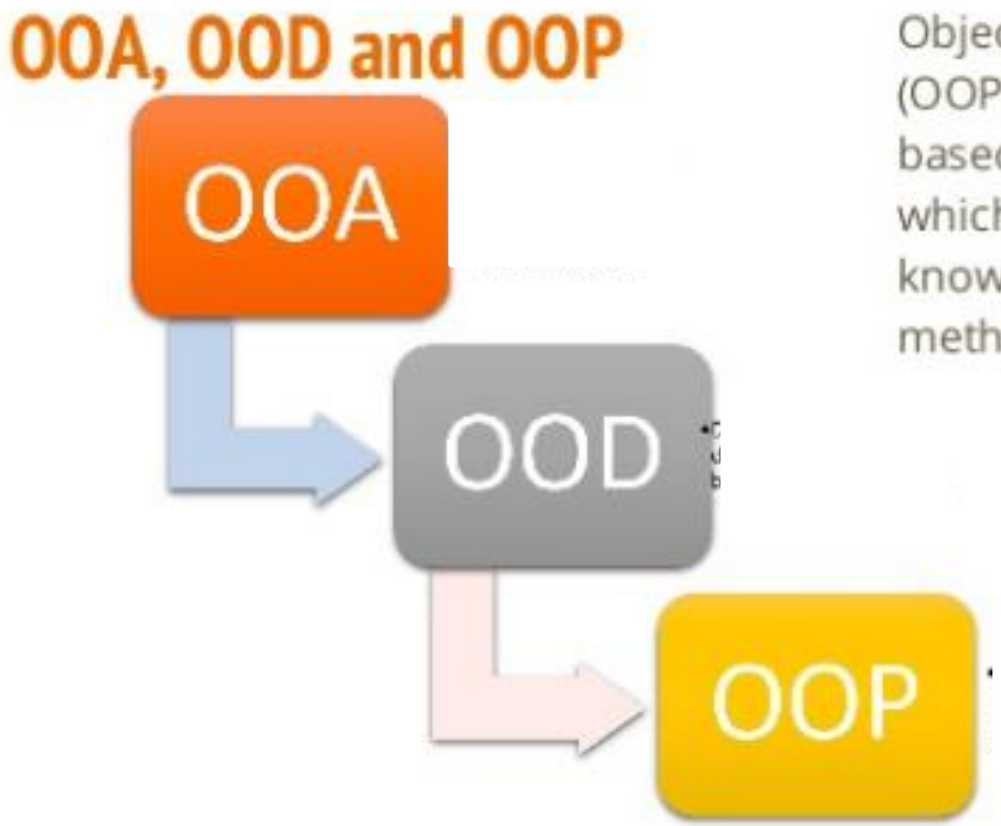
OO Scope

OBJECT-ORIENTED **ANALYSIS**: Examines the requirements of a system or a problem from the perspective of the classes and objects found in the vocabulary of the problem domain

OBJECT-ORIENTED **DESIGN**: Architectures a system as made of objects and classes, specifying their relationships (like inheritance) and interactions.

OBJECT-ORIENTED **PROGRAMMING**: A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes.

OO Scope



Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, often known as attributes; and methods.

https://www.slideshare.net/HungHoNgoc/group-8-presentationmetricsforobjectorientedsystem

Closing Note

The direct **object** is the **noun** that receives the action of the transitive verb. Typically, a direct **object** follows the verb and can be found by asking who or what received the action of the verb. Aug 23, 2013

Nouns: Direct Object - The Tongue Untied
www.grammaruntied.com/blog/?p=671

Programming is based on actions – verbs!
Be critical and don't always go along with the herd:

https://www.youtube.com/watch?v=QM1iUe6IofM

Presentation Terminated