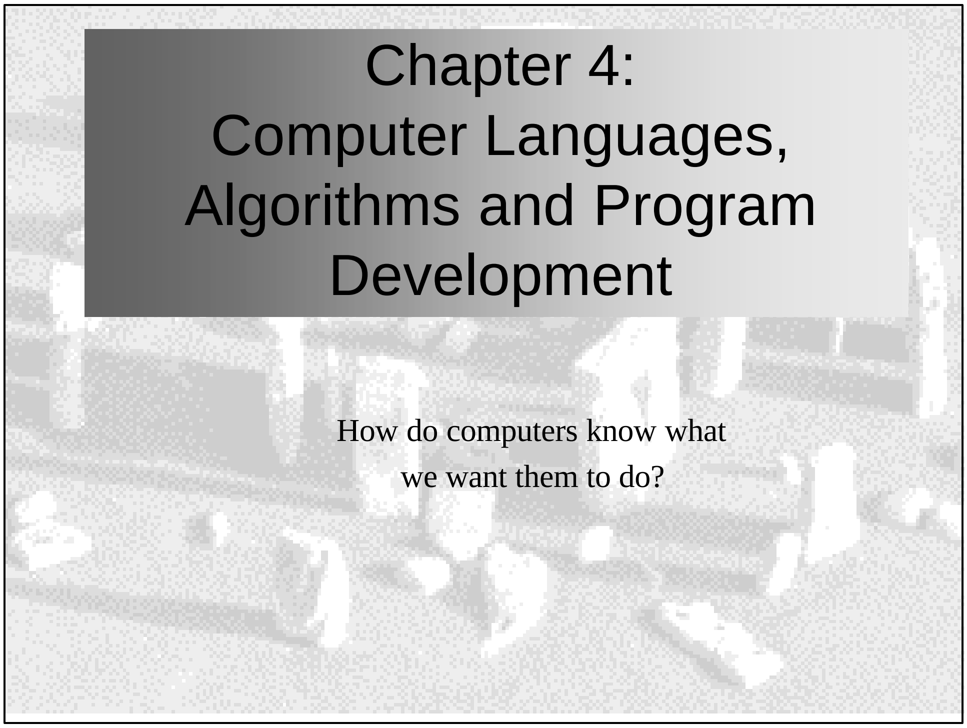
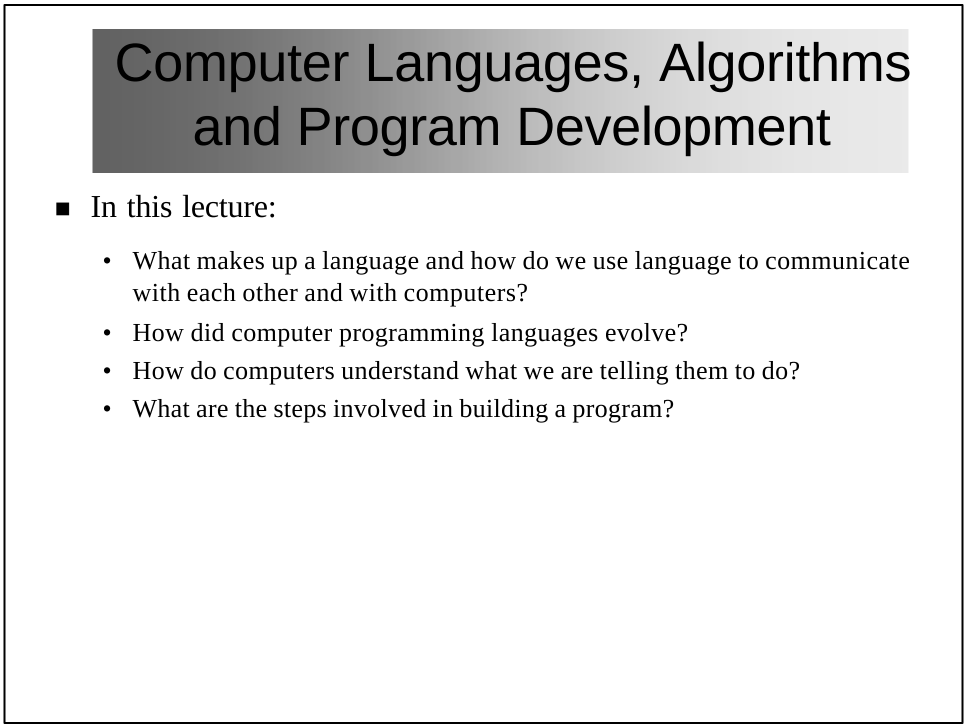


Chapter 4: Computer Languages, Algorithms and Program Development

How do computers know what we want them to do?

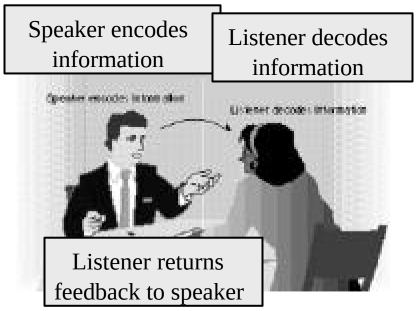


Computer Languages, Algorithms and Program Development

- In this lecture:
  - What makes up a language and how do we use language to communicate with each other and with computers?
  - How did computer programming languages evolve?
  - How do computers understand what we are telling them to do?
  - What are the steps involved in building a program?

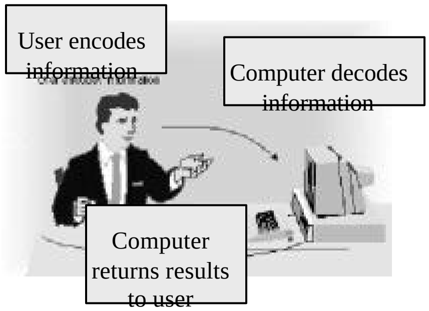
Communicating with a Computer

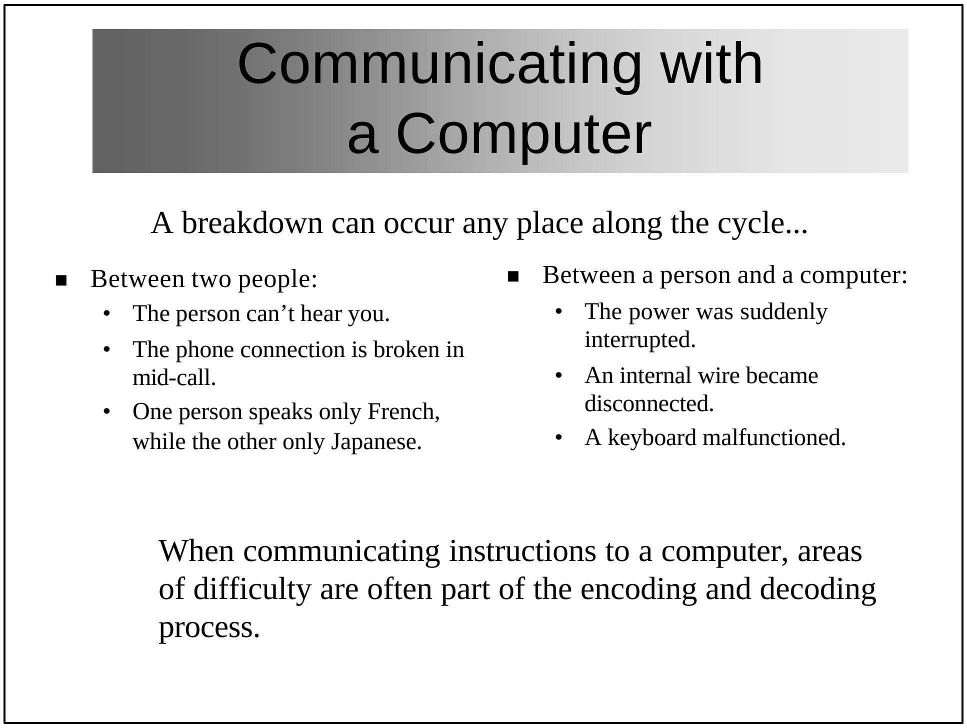
- Communication cycle
  - One complete unit of communication.
    - An idea to be sent.
    - An encoder.
    - A sender.
    - A medium.
    - A receiver.
    - A decoder.
    - A response.


Speaker encodes information
Listener decodes information
Listener returns feedback to speaker

Communicating with a Computer

- Substituting a computer for one of the people in the communication process.
  - Process is basically the same.
    - Response may be symbols on the monitor.


User encodes information
Computer decodes information
Computer returns results to user

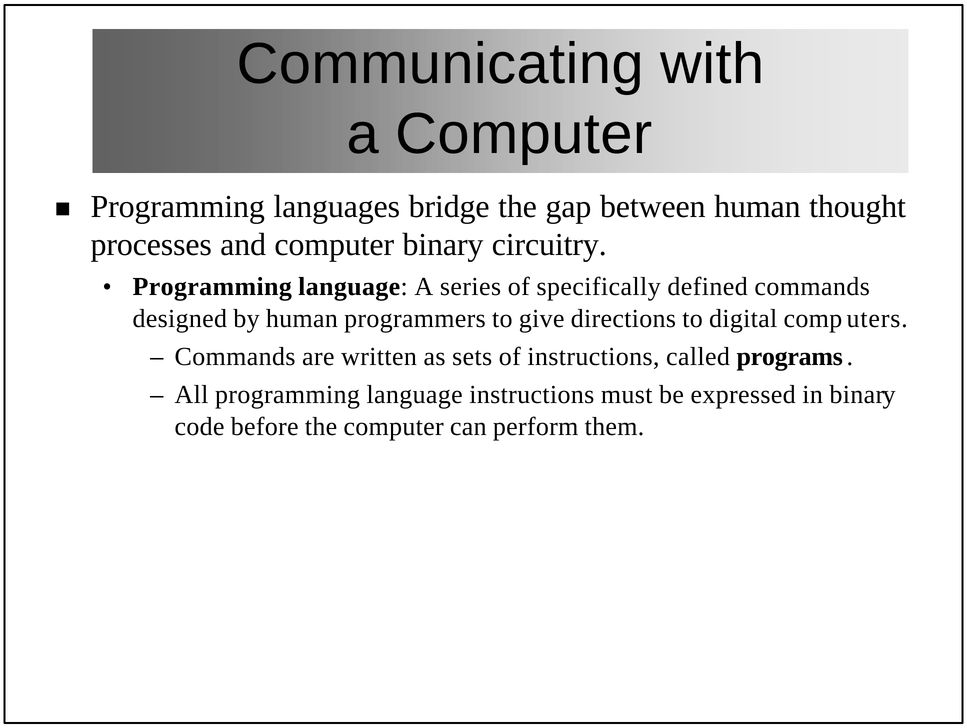


Communicating with a Computer

A breakdown can occur any place along the cycle...

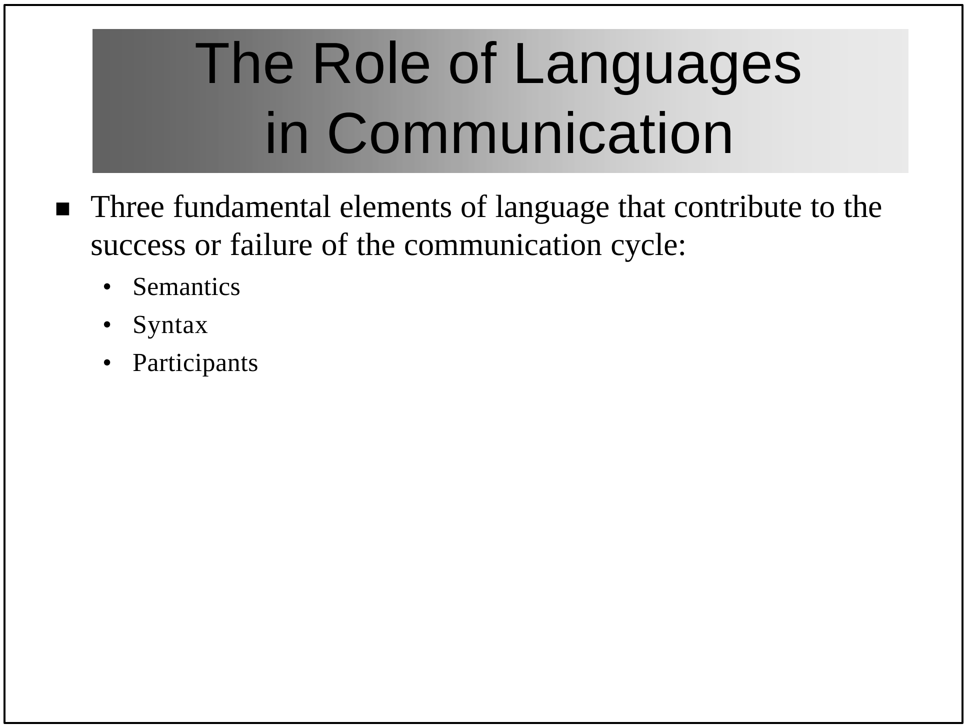
- Between two people:
  - The person can’t hear you.
  - The phone connection is broken in mid-call.
  - One person speaks only French, while the other only Japanese.
- Between a person and a computer:
  - The power was suddenly interrupted.
  - An internal wire became disconnected.
  - A keyboard malfunctioned.

When communicating instructions to a computer, areas of difficulty are often part of the encoding and decoding process.



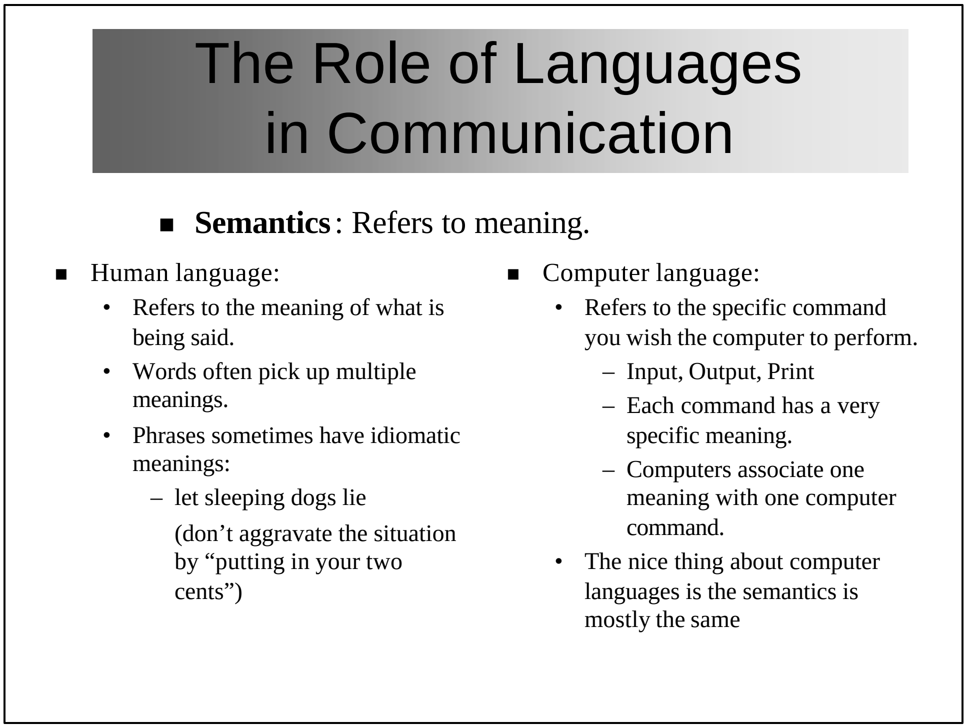
Communicating with a Computer

- Programming languages bridge the gap between human thought processes and computer binary circuitry.
  - **Programming language**: A series of specifically defined commands designed by human programmers to give directions to digital comp uters.
    - Commands are written as sets of instructions, called **programs**.
    - All programming language instructions must be expressed in binary code before the computer can perform them.



The Role of Languages in Communication

- Three fundamental elements of language that contribute to the success or failure of the communication cycle:
    - Semantics
    - Syntax
    - Participants



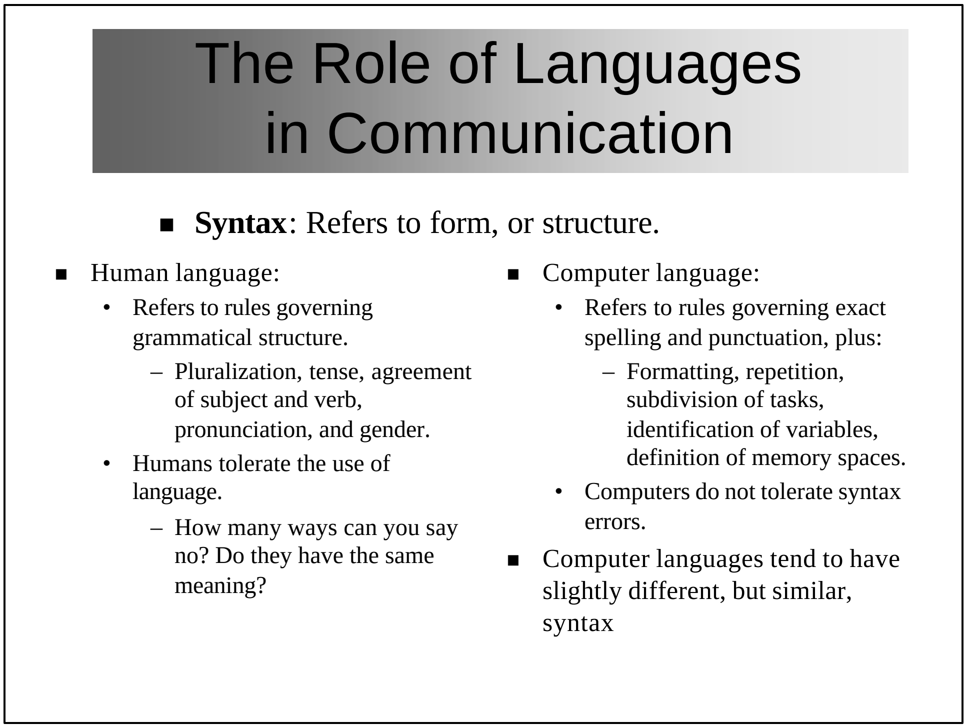
The Role of Languages in Communication

- **Semantics**: Refers to meaning.

- Human language:
    - Refers to the meaning of what is being said.
    - Words often pick up multiple meanings.
    - Phrases sometimes have idiomatic meanings:
        - let sleeping dogs lie

          (don’t aggravate the situation by “putting in your two cents”)

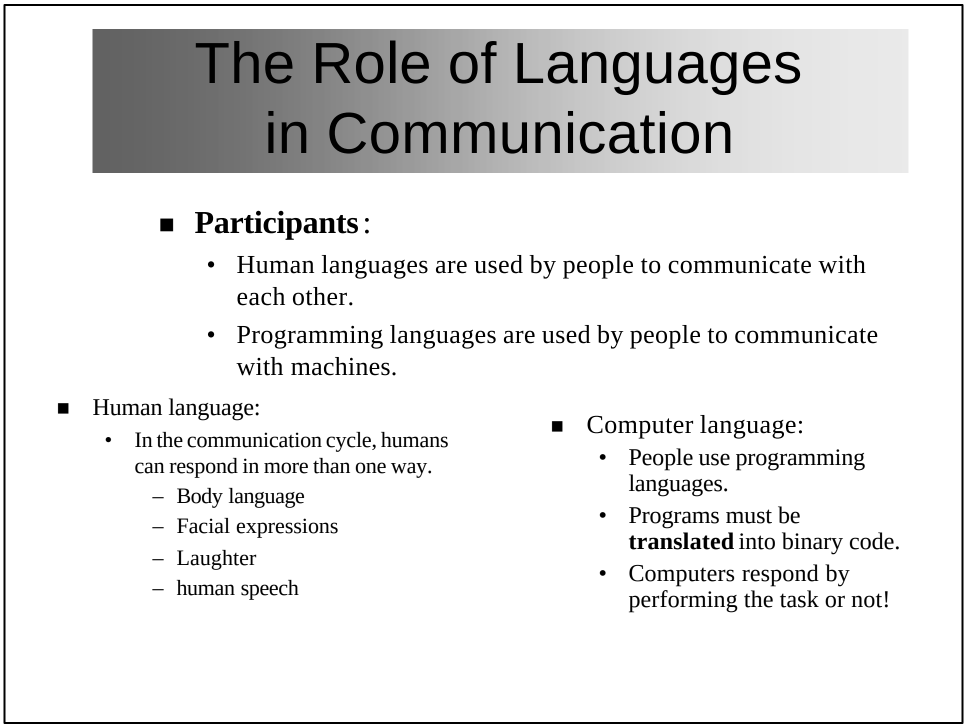
- Computer language:
    - Refers to the specific command you wish the computer to perform.
        - Input, Output, Print
        - Each command has a very specific meaning.
        - Computers associate one meaning with one computer command.
    - The nice thing about computer languages is the semantics is mostly the same



The Role of Languages in Communication

- **Syntax**: Refers to form, or structure.

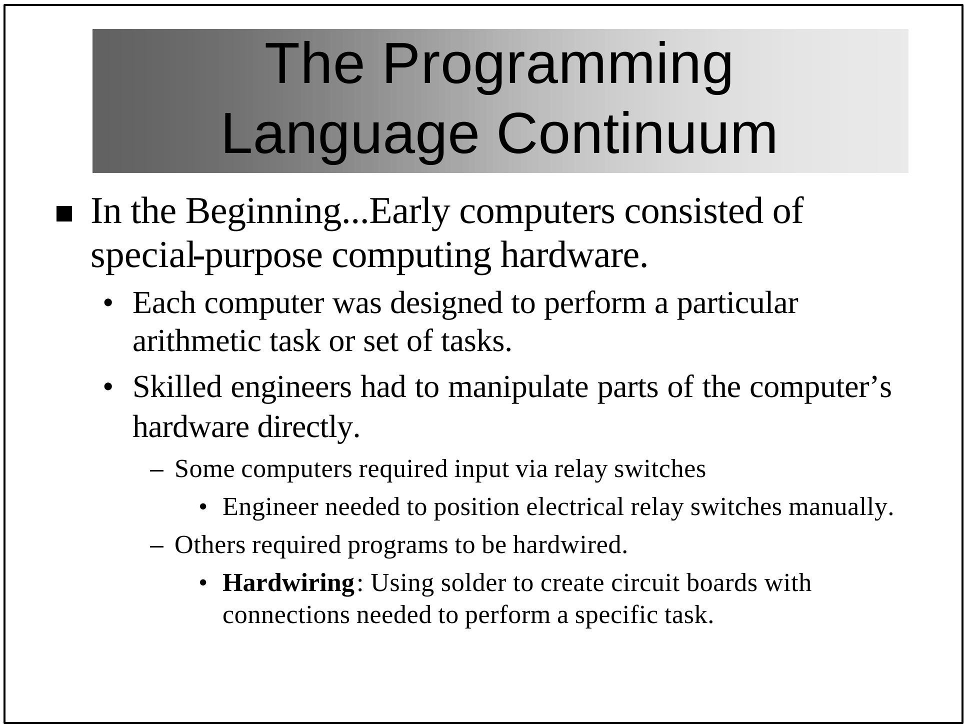
- Human language:
  - Refers to rules governing grammatical structure.
    - Pluralization, tense, agreement of subject and verb, pronunciation, and gender.
  - Humans tolerate the use of language.
    - How many ways can you say no? Do they have the same meaning?
- Computer language:
  - Refers to rules governing exact spelling and punctuation, plus:
    - Formatting, repetition, subdivision of tasks, identification of variables, definition of memory spaces.
  - Computers do not tolerate syntax errors.
- Computer languages tend to have slightly different, but similar, syntax



The Role of Languages in Communication

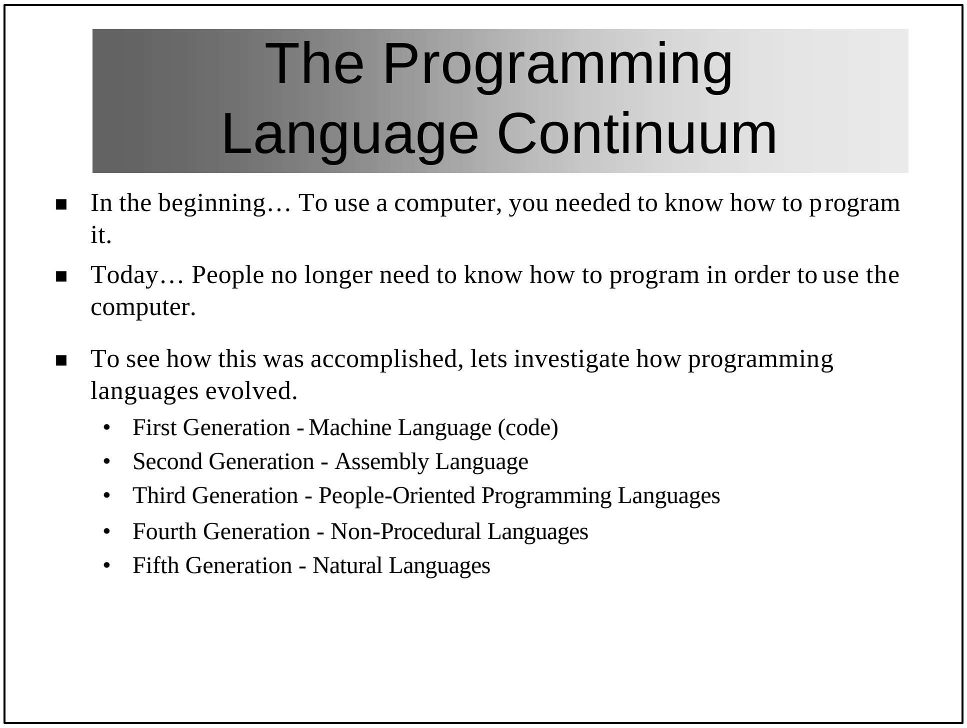
- **Participants**:
  - Human languages are used by people to communicate with each other.
  - Programming languages are used by people to communicate with machines.

- Human language:
  - In the communication cycle, humans can respond in more than one way.
    - Body language
    - Facial expressions
    - Laughter
    - human speech
- Computer language:
  - People use programming languages.
  - Programs must be **translated** into binary code.
  - Computers respond by performing the task or not!



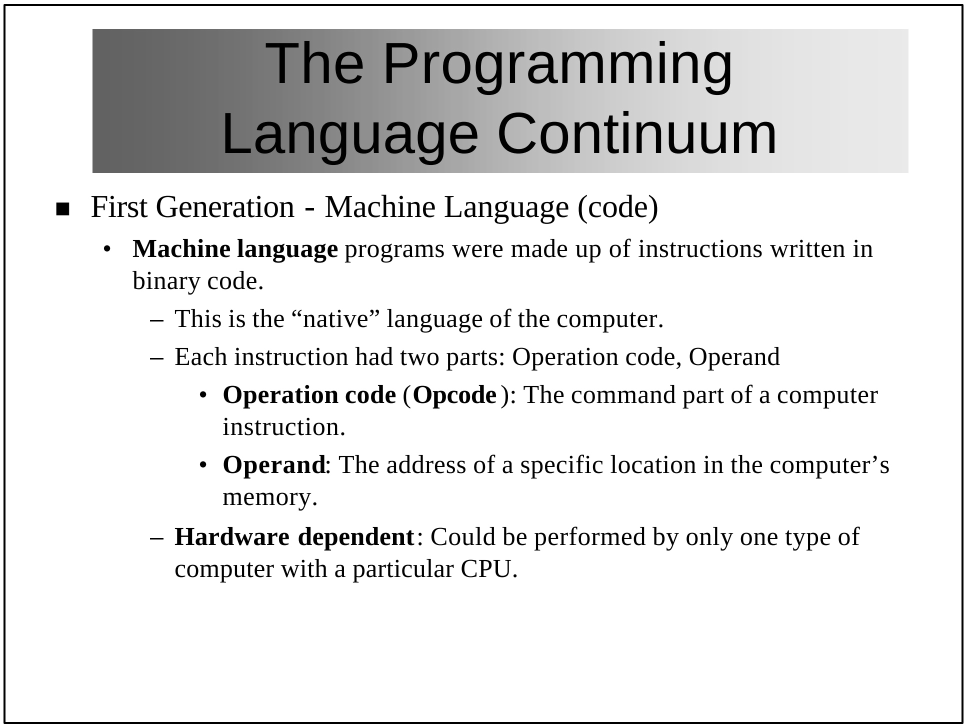
The Programming Language Continuum

- In the Beginning...Early computers consisted of special-purpose computing hardware.
  - Each computer was designed to perform a particular arithmetic task or set of tasks.
  - Skilled engineers had to manipulate parts of the computer's hardware directly.
    - Some computers required input via relay switches
      - Engineer needed to position electrical relay switches manually.
    - Others required programs to be hardwired.
      - **Hardwiring**: Using solder to create circuit boards with connections needed to perform a specific task.



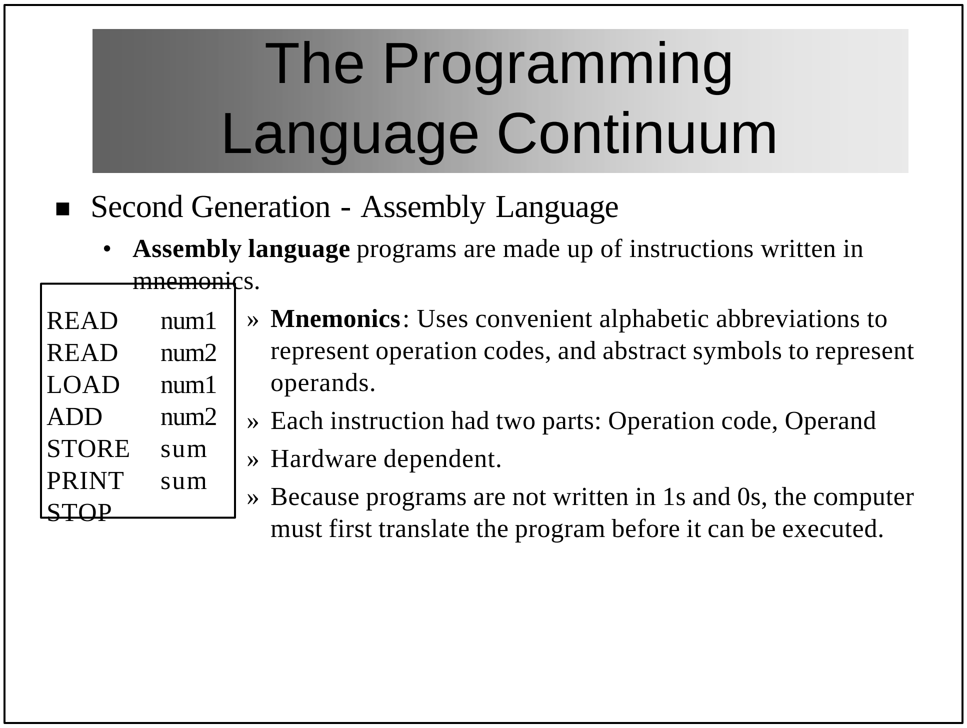
The Programming Language Continuum

- In the beginning… To use a computer, you needed to know how to program it.
- Today… People no longer need to know how to program in order to use the computer.
- To see how this was accomplished, lets investigate how programming languages evolved.
  - First Generation - Machine Language (code)
  - Second Generation - Assembly Language
  - Third Generation - People-Oriented Programming Languages
  - Fourth Generation - Non-Procedural Languages
  - Fifth Generation - Natural Languages



The Programming Language Continuum

- First Generation - Machine Language (code)
  - **Machine language** programs were made up of instructions written in binary code.
    - This is the “native” language of the computer.
    - Each instruction had two parts: Operation code, Operand
      - **Operation code** (**Opcode**): The command part of a computer instruction.
      - **Operand**: The address of a specific location in the computer’s memory.
    - **Hardware dependent**: Could be performed by only one type of computer with a particular CPU.



The Programming Language Continuum

- Second Generation - Assembly Language
  - **Assembly language** programs are made up of instructions written in mnemonics.

```
READ     num1
READ     num2
LOAD     num1
ADD      num2
STORE    sum
PRINT    sum
STOP
```

» **Mnemonics**: Uses convenient alphabetic abbreviations to represent operation codes, and abstract symbols to represent operands.

» Each instruction had two parts: Operation code, Operand

» Hardware dependent.

» Because programs are not written in 1s and 0s, the computer must first translate the program before it can be executed.

The Programming Language Continuum

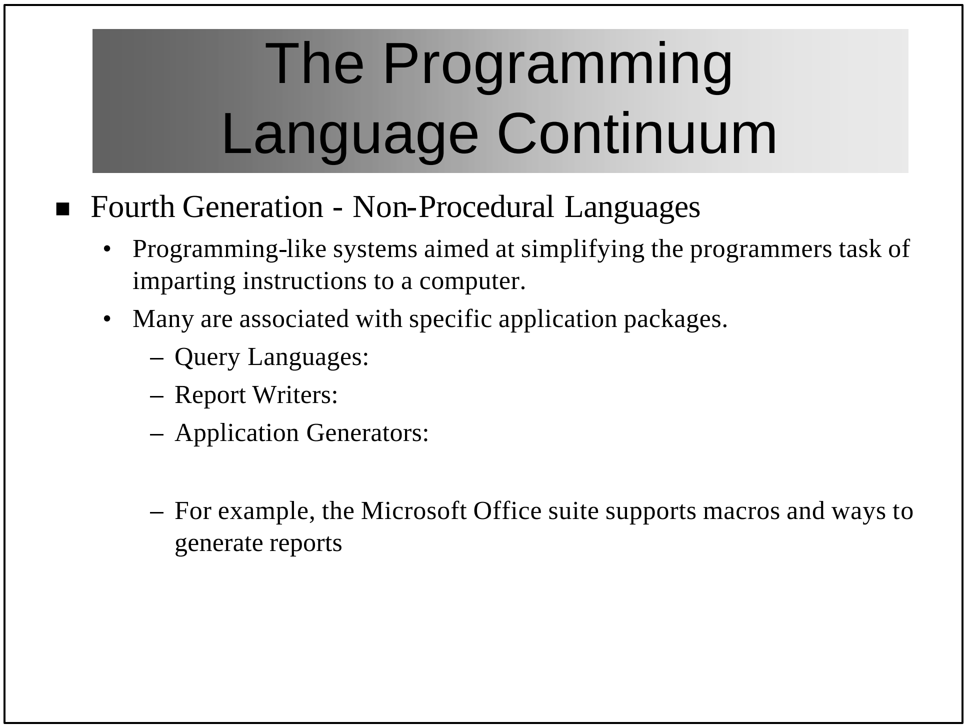
- Third Generation - People-Oriented Programs
  - Instructions in these languages are called statements.
    - **High-level languages**: Use statements that resemble English phrases combined with mathematical terms needed to express the problem or task being programmed.
    - Transportable: NOT-Hardware dependent.
    - Because programs are not written in 1s and 0s, the computer must first translate the program before it can be executed.
  - Examples: COBOL, FORTRAN, Basic (old version not new), Pascal, C

The Programming Language Continuum

- Pascal Example: Read in two numbers, add them, and print them out.

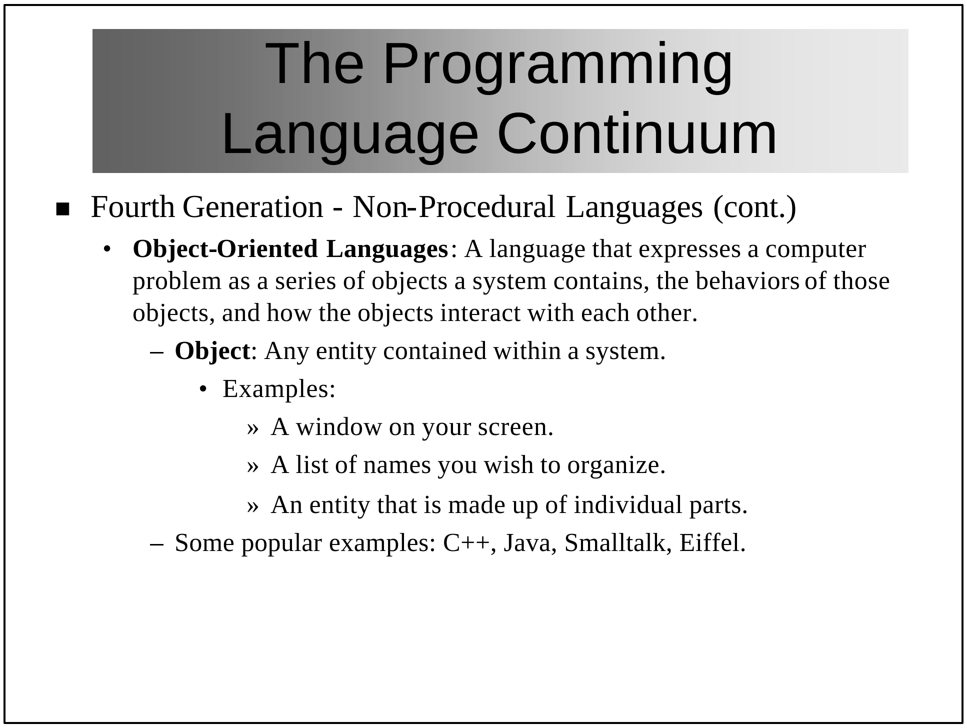
```
Program sum2(input,output);
var
  num1,num2,sum : integer;

begin
  read(num1,num2);
  sum:=num1+num2;
  writeln(sum)
end.
```



The Programming Language Continuum

- Fourth Generation - Non-Procedural Languages
  - Programming-like systems aimed at simplifying the programmers task of imparting instructions to a computer.
  - Many are associated with specific application packages.
    - Query Languages:
    - Report Writers:
    - Application Generators:
    - For example, the Microsoft Office suite supports macros and ways to generate reports



The Programming Language Continuum

- Fourth Generation - Non-Procedural Languages (cont.)
  - **Object-Oriented Languages**: A language that expresses a computer problem as a series of objects a system contains, the behaviors of those objects, and how the objects interact with each other.
    - **Object**: Any entity contained within a system.
      - Examples:
        - » A window on your screen.
        - » A list of names you wish to organize.
        - » An entity that is made up of individual parts.
    - Some popular examples: C++, Java, Smalltalk, Eiffel.

The Programming Language Continuum

- Fifth Generation - Natural Languages
  - **Natural-Language**: Languages that use ordinary conversation in one’s own language.
    - Research and experimentation toward this goal is being done.
      - Intelligent compilers are now being developed to translate natural language (spoken) programs into structured machine-coded instructions that can be executed by computers.
      - Effortless, error-free natural language programs are still some distance into the future.

Assembled, Compiled, or Interpreted Languages

- All programs must be translated before their instructions can be executed.
- Computer languages can be grouped according to which translation process is used to convert the instructions into binary code:
  - Assemblers
  - Interpreters
  - Compilers

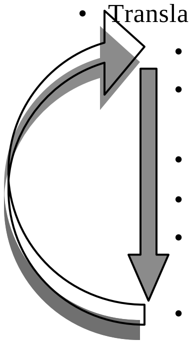
Assembled, Compiled, or Interpreted Languages

- **Assembled languages**:
  - **Assembler:** a program used to translate Assembly language programs.
  - Produces one line of binary code per original program statement.
    - The entire program is assembled before the program is sent to the computer for execution.
    - Similar to the machine code exercise we did in class
  - Example of 6502 assembly language and machine code:

– JSR SWAP	20 1C 1F
– LDA X2	A5 04
– LDY =$80	A0 80
– STY X2	49 80

Assembled, Compiled, or Interpreted Languages

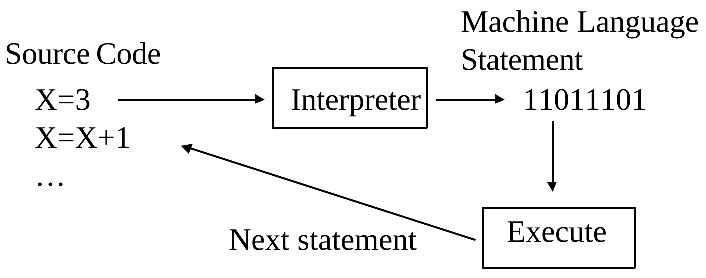
- **Interpreted Languages:**
  - **Interpreter:** A program used to translate high-level programs.
  - Translates one line of the program into binary code at a time:



  - An instruction is **fetched** from the original source code.
  - The Interpreter checks the single instruction for errors. (If an error is found, translation and execution ceases. Otherwise…)
  - The instruction is translated into binary code.
  - The binary coded instruction is **executed**.
  - The fetch and execute process repeats for the entire program.

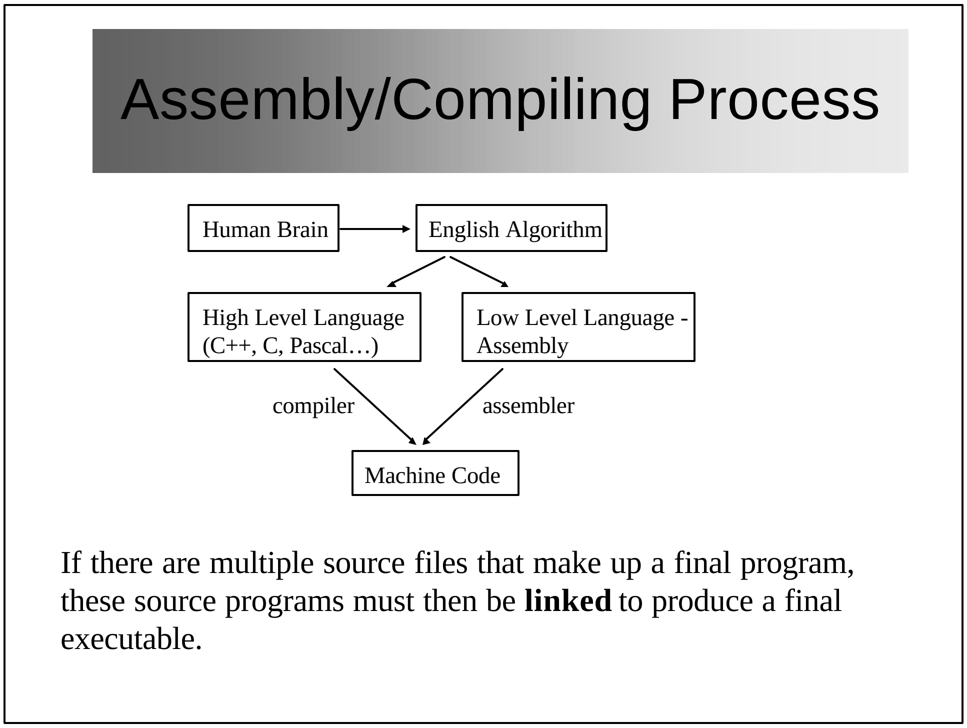
  - Examples: Lisp, Prolog, Java, JavaScript (used on Web Pages)

Interpreted Programs



Assembled, Compiled, or Interpreted Languages

- **Compiled languages**:
  - **Compiler:** a program used to translate high-level programs.
  - Translates the entire program into binary code before anything is sent to the CPU for execution.
    - The translation process for a compiled program:
      - First, the Compiler checks the entire program for syntax errors in the original **source code**.
      - Next, it translates all of the instructions into binary code.
        - Two versions of the same program exist: the original **source code** version, and the binary code version (**object code**).
      - Last, the CPU attempts execution only after the programmer requests that the program be executed.
  - Examples: C, C++, C#, Java, Pascal, Visual Basic



Assembly/Compiling Process

Human Brain → English Algorithm

High Level Language (C++, C, Pascal…)

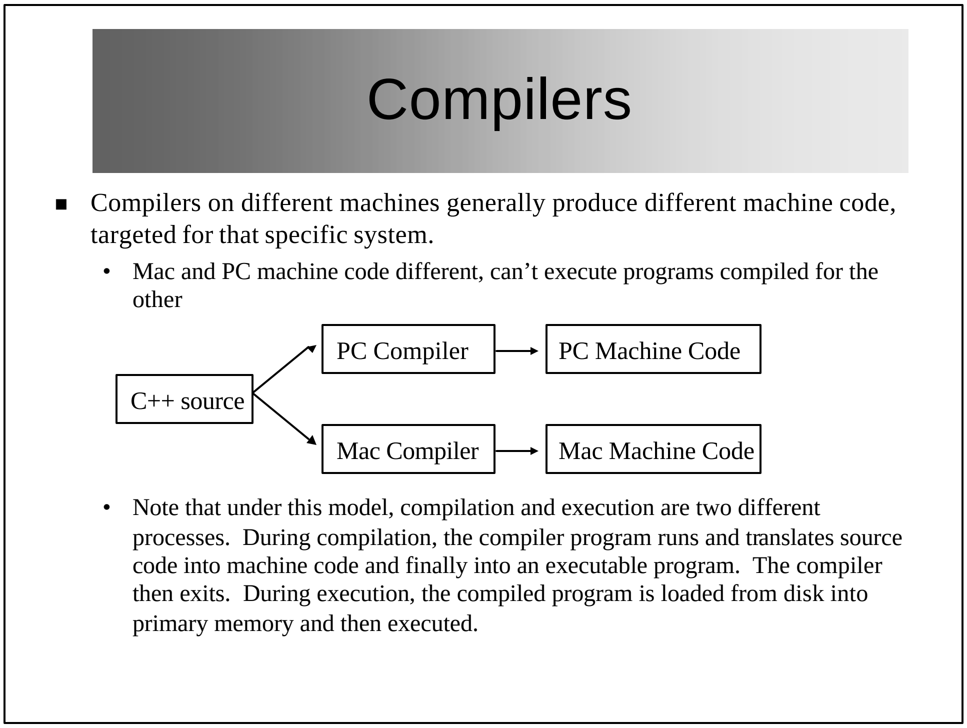
Low Level Language - Assembly

compiler

assembler

Machine Code

If there are multiple source files that make up a final program, these source programs must then be **linked** to produce a final executable.



Compilers

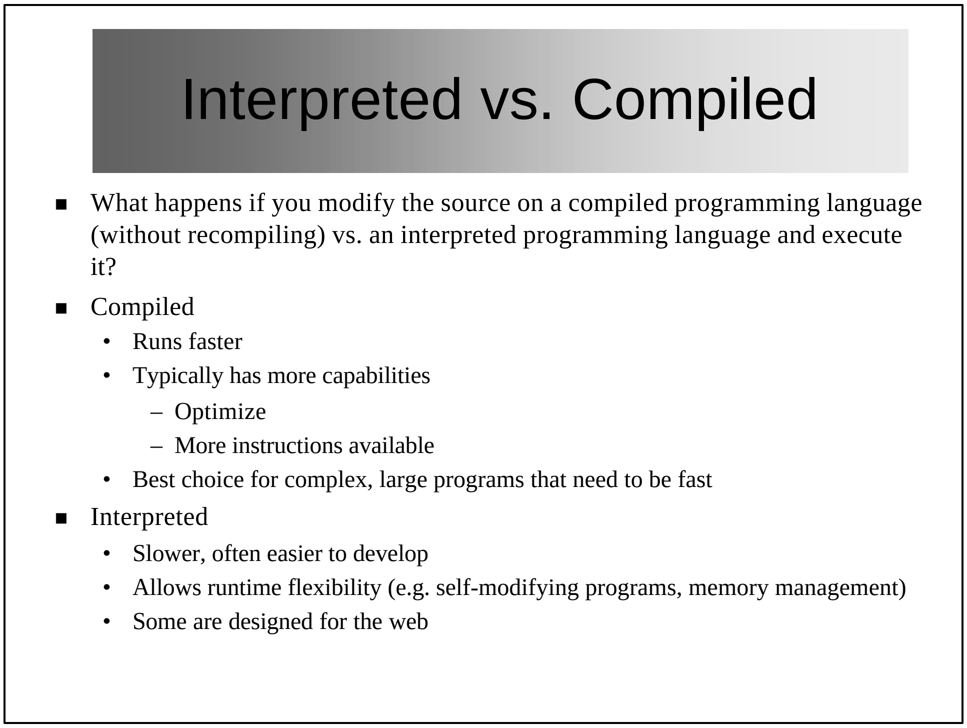
- Compilers on different machines generally produce different machine code, targeted for that specific system.
  - Mac and PC machine code different, can't execute programs compiled for the other

C++ source

PC Compiler → PC Machine Code

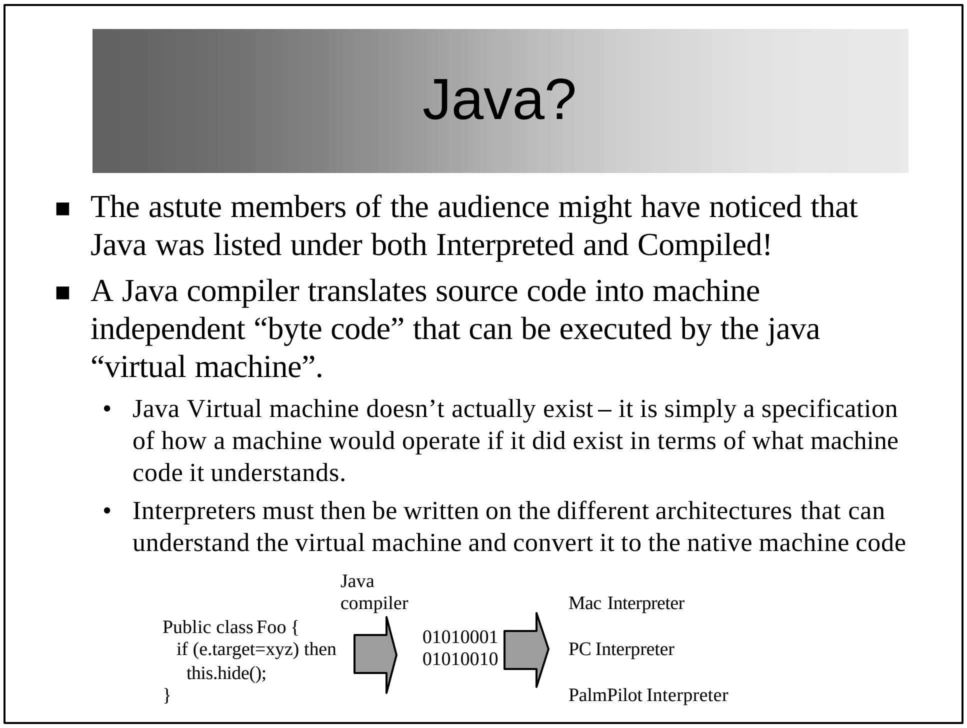
Mac Compiler → Mac Machine Code

  - Note that under this model, compilation and execution are two different processes. During compilation, the compiler program runs and translates source code into machine code and finally into an executable program. The compiler then exits. During execution, the compiled program is loaded from disk into primary memory and then executed.



Interpreted vs. Compiled

- What happens if you modify the source on a compiled programming language (without recompiling) vs. an interpreted programming language and execute it?
- Compiled
  - Runs faster
  - Typically has more capabilities
    - Optimize
    - More instructions available
  - Best choice for complex, large programs that need to be fast
- Interpreted
  - Slower, often easier to develop
  - Allows runtime flexibility (e.g. self-modifying programs, memory management)
  - Some are designed for the web



Java?

- The astute members of the audience might have noticed that Java was listed under both Interpreted and Compiled!
- A Java compiler translates source code into machine independent "byte code" that can be executed by the java "virtual machine".
  - Java Virtual machine doesn't actually exist – it is simply a specification of how a machine would operate if it did exist in terms of what machine code it understands.
  - Interpreters must then be written on the different architectures that can understand the virtual machine and convert it to the native machine code

```
Public class Foo {
   if (e.target=xyz) then
     this.hide();
}
```

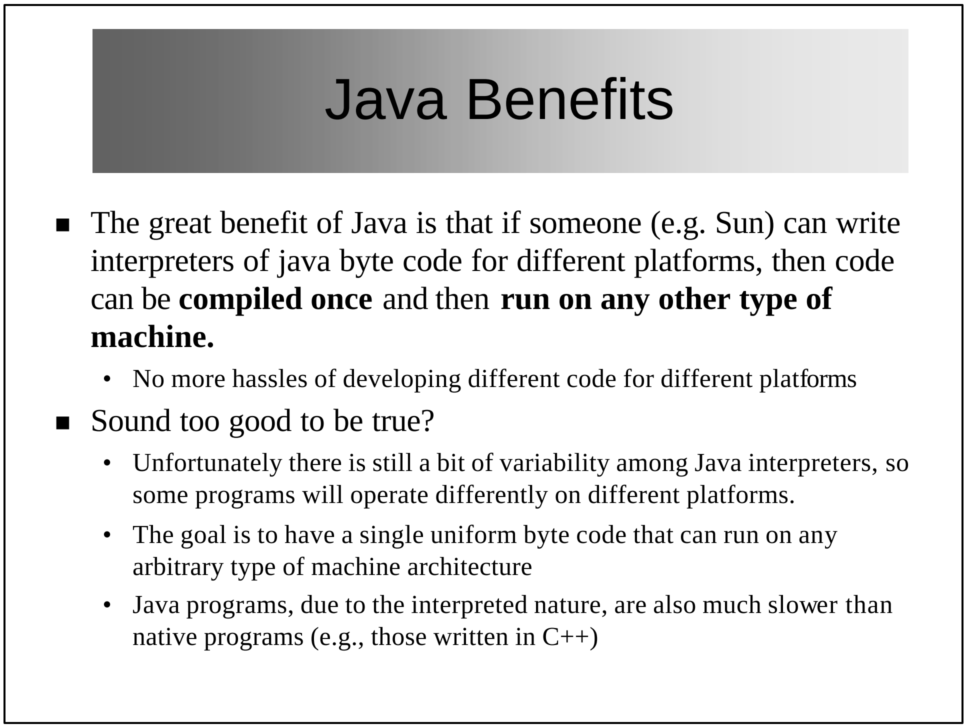
Java compiler

01010001
01010010

Mac Interpreter

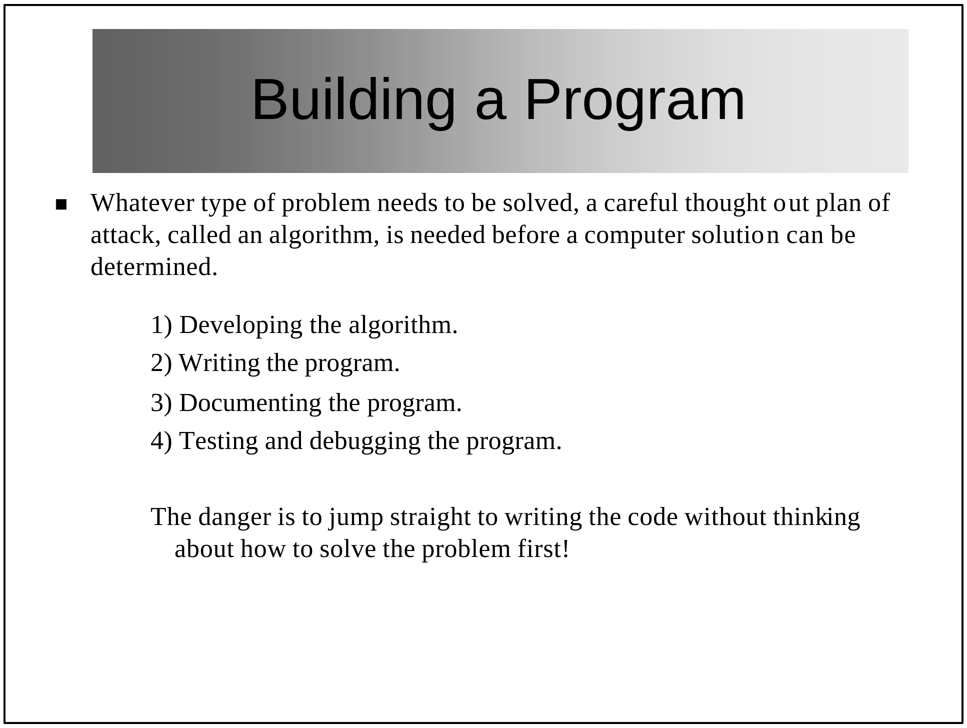
PC Interpreter

PalmPilot Interpreter



Java Benefits

- The great benefit of Java is that if someone (e.g. Sun) can write interpreters of java byte code for different platforms, then code can be **compiled once** and then **run on any other type of machine.**
  - No more hassles of developing different code for different platforms
- Sound too good to be true?
  - Unfortunately there is still a bit of variability among Java interpreters, so some programs will operate differently on different platforms.
  - The goal is to have a single uniform byte code that can run on any arbitrary type of machine architecture
  - Java programs, due to the interpreted nature, are also much slower than native programs (e.g., those written in C++)

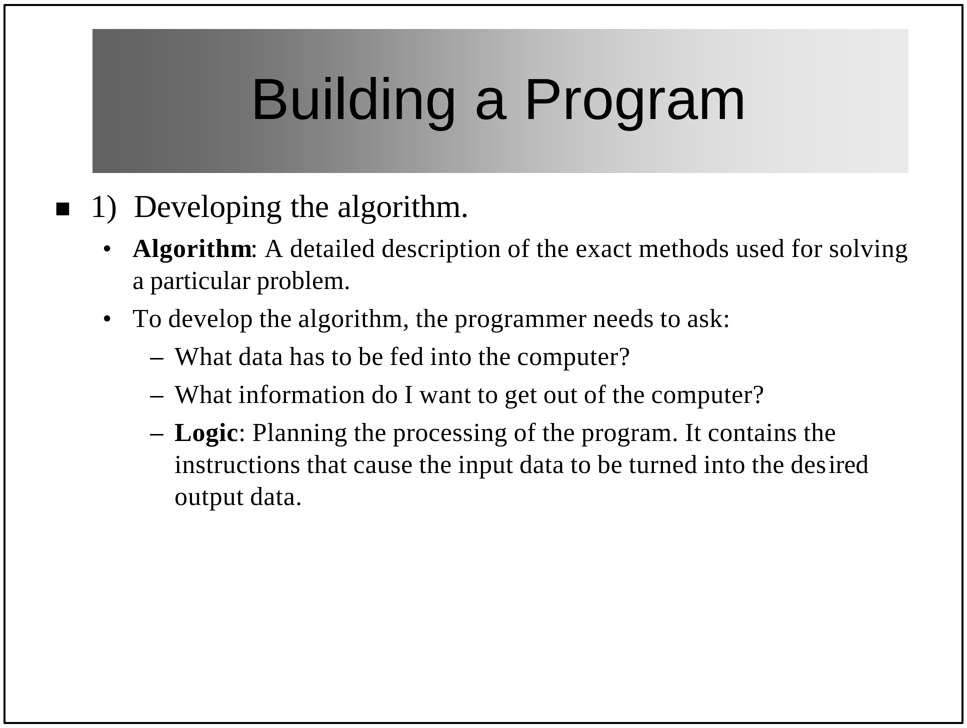


Building a Program

- Whatever type of problem needs to be solved, a careful thought out plan of attack, called an algorithm, is needed before a computer solution can be determined.

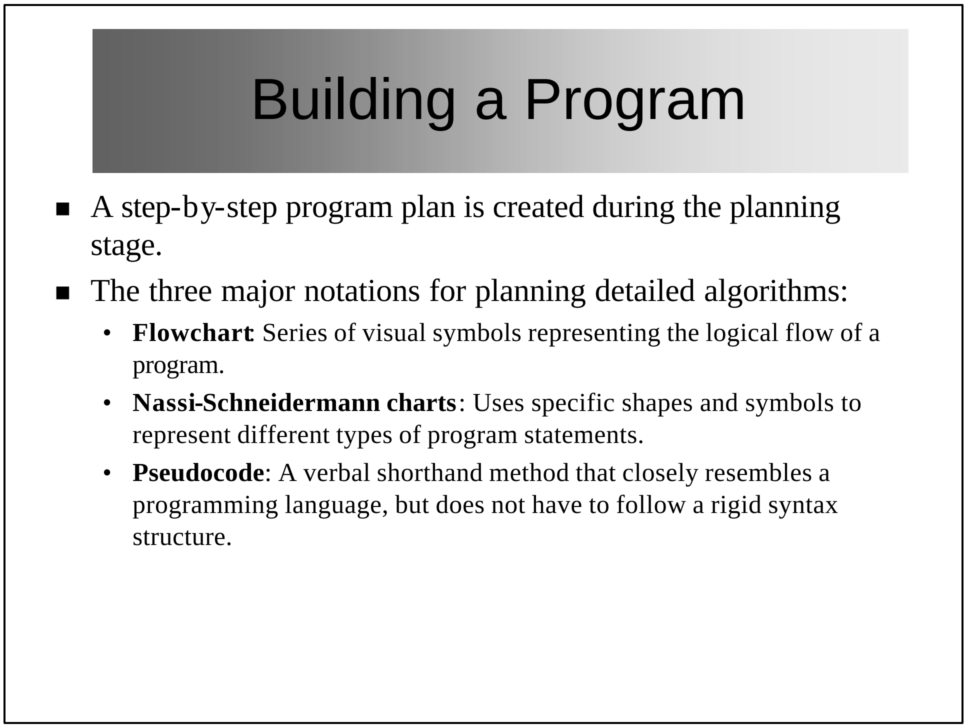
1) Developing the algorithm.
2) Writing the program.
3) Documenting the program.
4) Testing and debugging the program.

The danger is to jump straight to writing the code without thinking about how to solve the problem first!



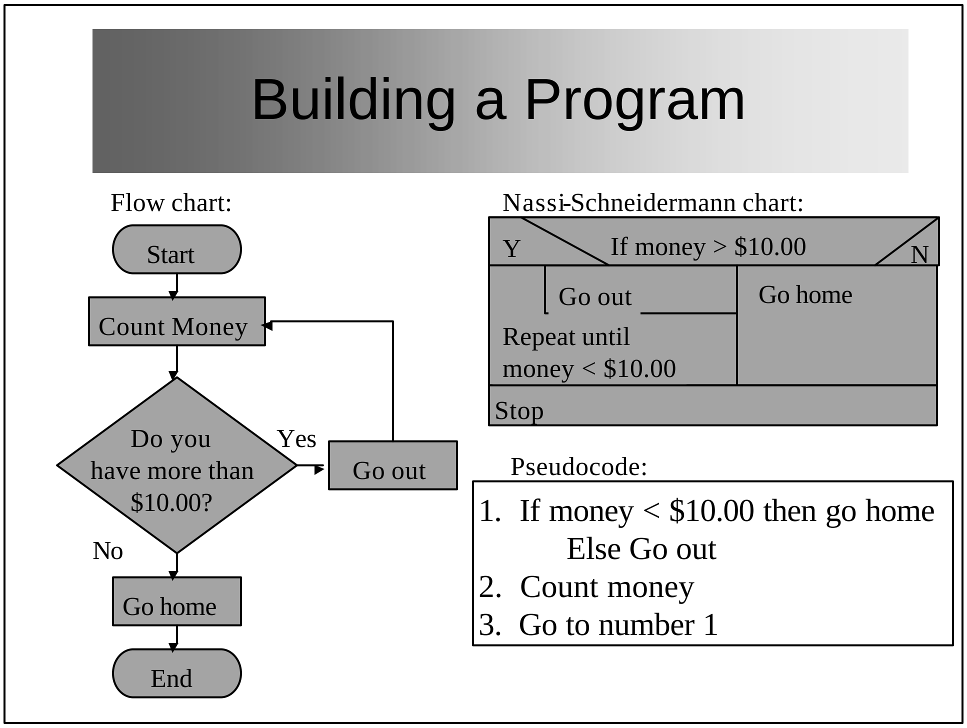
Building a Program

- 1) Developing the algorithm.
  - **Algorithm**: A detailed description of the exact methods used for solving a particular problem.
  - To develop the algorithm, the programmer needs to ask:
    - What data has to be fed into the computer?
    - What information do I want to get out of the computer?
    - **Logic**: Planning the processing of the program. It contains the instructions that cause the input data to be turned into the desired output data.



Building a Program

- A step-by-step program plan is created during the planning stage.
- The three major notations for planning detailed algorithms:
  - **Flowchart**: Series of visual symbols representing the logical flow of a program.
  - **Nassi-Schneidermann charts**: Uses specific shapes and symbols to represent different types of program statements.
  - **Pseudocode**: A verbal shorthand method that closely resembles a programming language, but does not have to follow a rigid syntax structure.



Building a Program

Flow chart:

Start

Count Money

Do you have more than $10.00?

Yes

Go out

No

Go home

End

Nassi-Schneidermann chart:

Y If money > $10.00 N

Go out

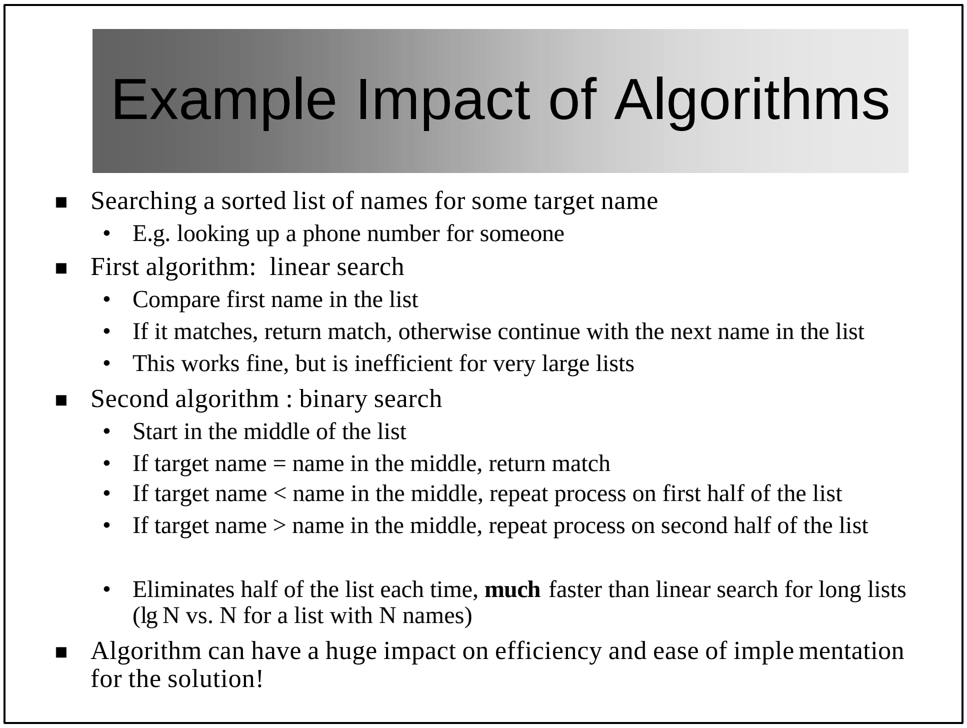
Repeat until money < $10.00

Go home

Stop

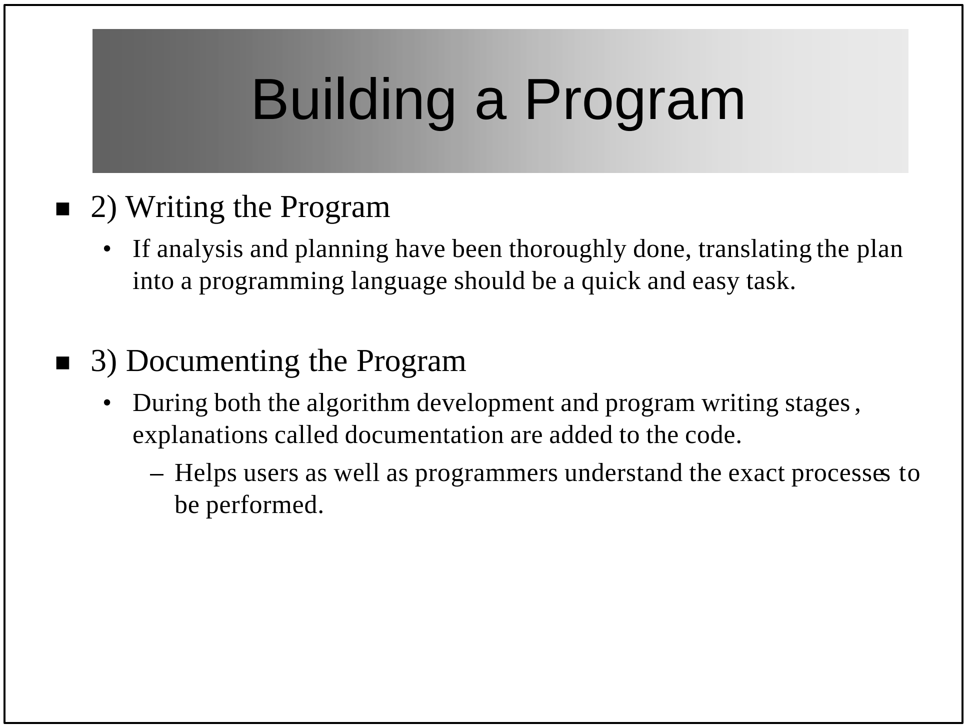
Pseudocode:

```
1. If money < $10.00 then go home
       Else Go out
2. Count money
3. Go to number 1
```



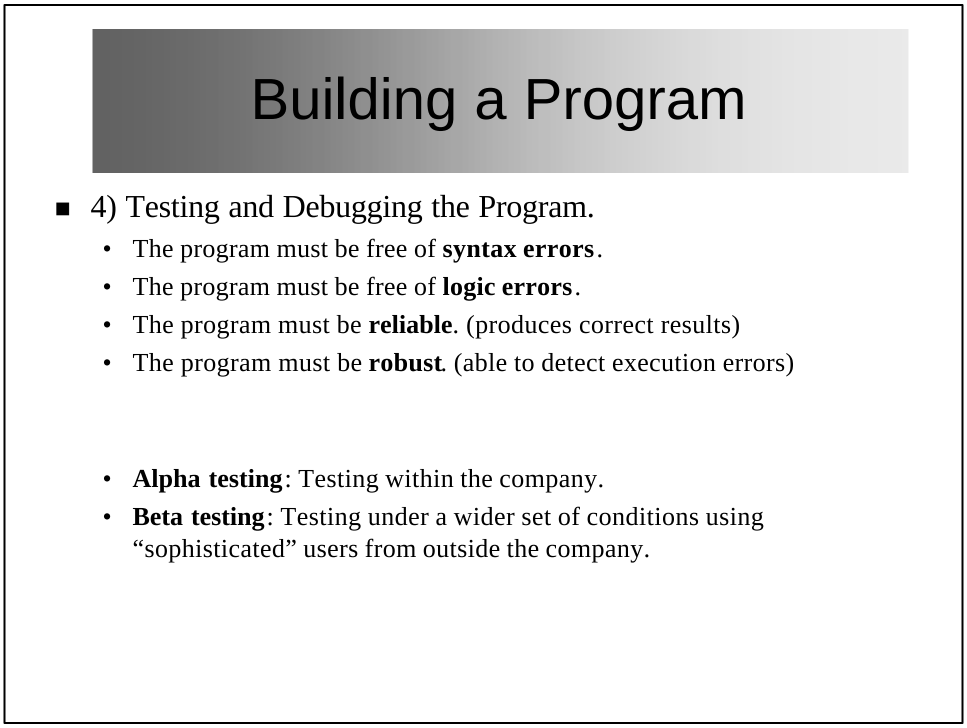
Example Impact of Algorithms

- Searching a sorted list of names for some target name
  - E.g. looking up a phone number for someone
- First algorithm: linear search
  - Compare first name in the list
  - If it matches, return match, otherwise continue with the next name in the list
  - This works fine, but is inefficient for very large lists
- Second algorithm : binary search
  - Start in the middle of the list
  - If target name = name in the middle, return match
  - If target name < name in the middle, repeat process on first half of the list
  - If target name > name in the middle, repeat process on second half of the list
  - Eliminates half of the list each time, **much** faster than linear search for long lists (lg N vs. N for a list with N names)
- Algorithm can have a huge impact on efficiency and ease of implementation for the solution!



Building a Program

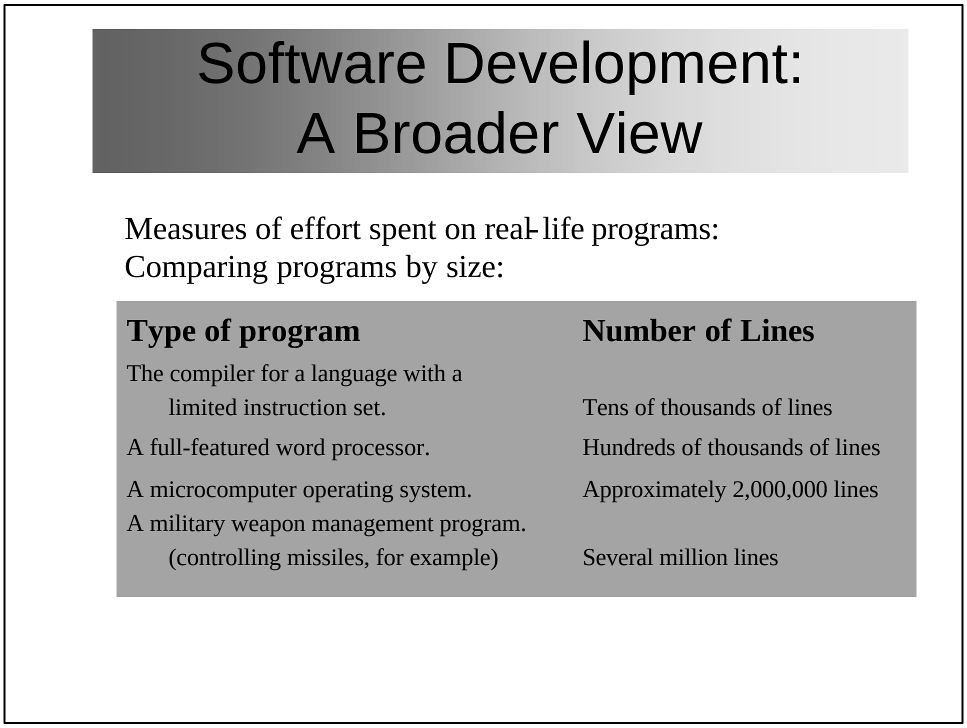
- 2) Writing the Program
  - If analysis and planning have been thoroughly done, translating the plan into a programming language should be a quick and easy task.
- 3) Documenting the Program
  - During both the algorithm development and program writing stages, explanations called documentation are added to the code.
    - – Helps users as well as programmers understand the exact processes to be performed.



Building a Program

- 4) Testing and Debugging the Program.
  - The program must be free of **syntax errors**.
  - The program must be free of **logic errors**.
  - The program must be **reliable**. (produces correct results)
  - The program must be **robust**. (able to detect execution errors)

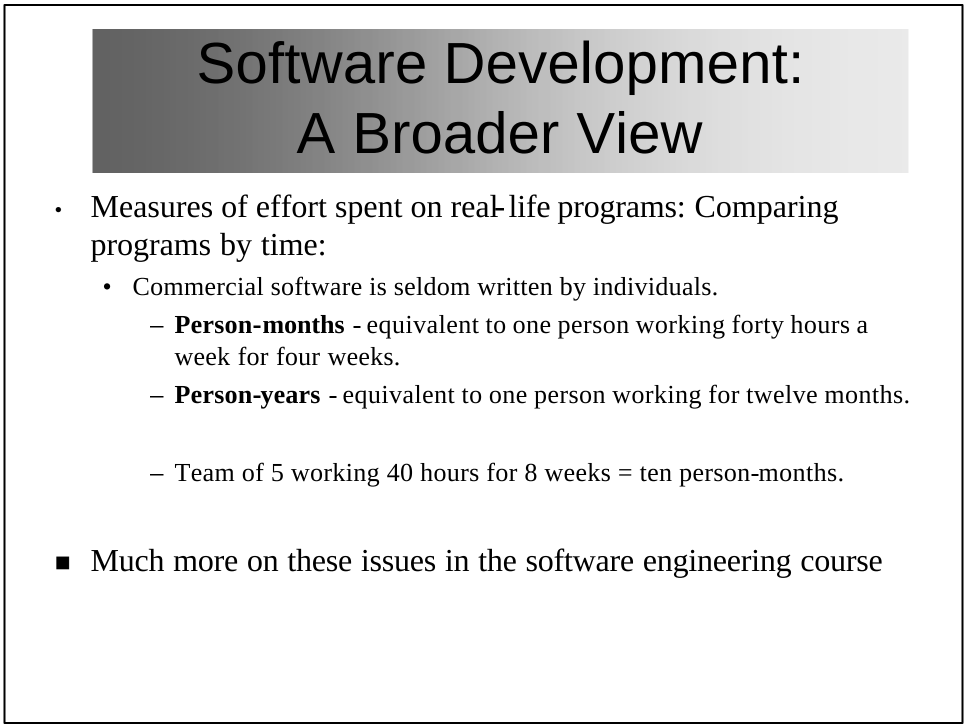
  - **Alpha testing**: Testing within the company.
  - **Beta testing**: Testing under a wider set of conditions using “sophisticated” users from outside the company.



Software Development: A Broader View

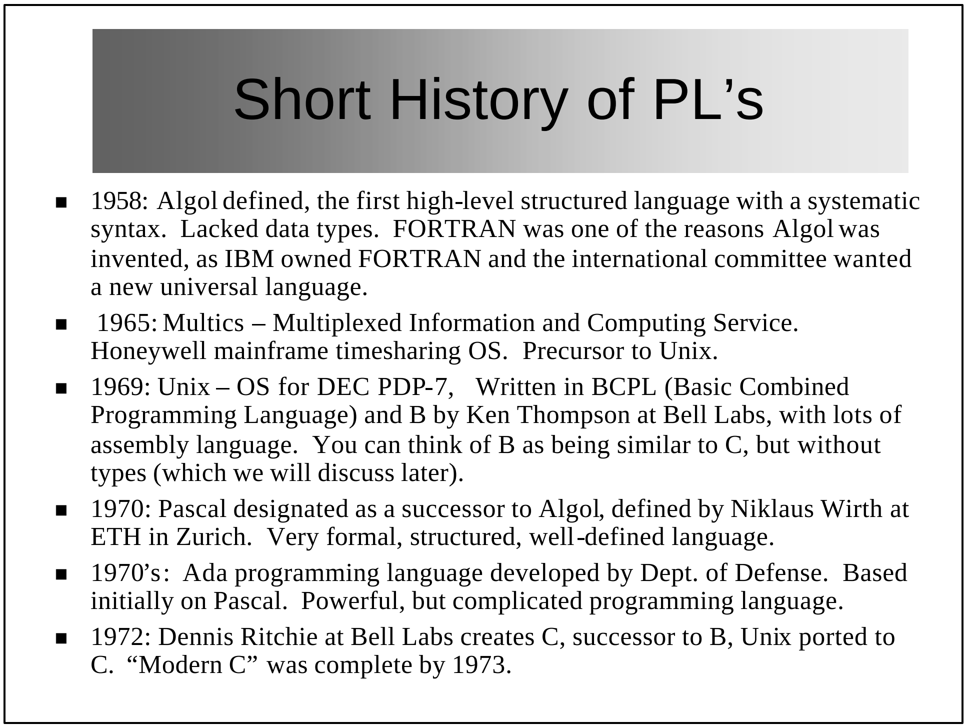
Measures of effort spent on real-life programs:
Comparing programs by size:

Type of program	Number of Lines
The compiler for a language with a limited instruction set.	Tens of thousands of lines
A full-featured word processor.	Hundreds of thousands of lines
A microcomputer operating system.	Approximately 2,000,000 lines
A military weapon management program. (controlling missiles, for example)	Several million lines



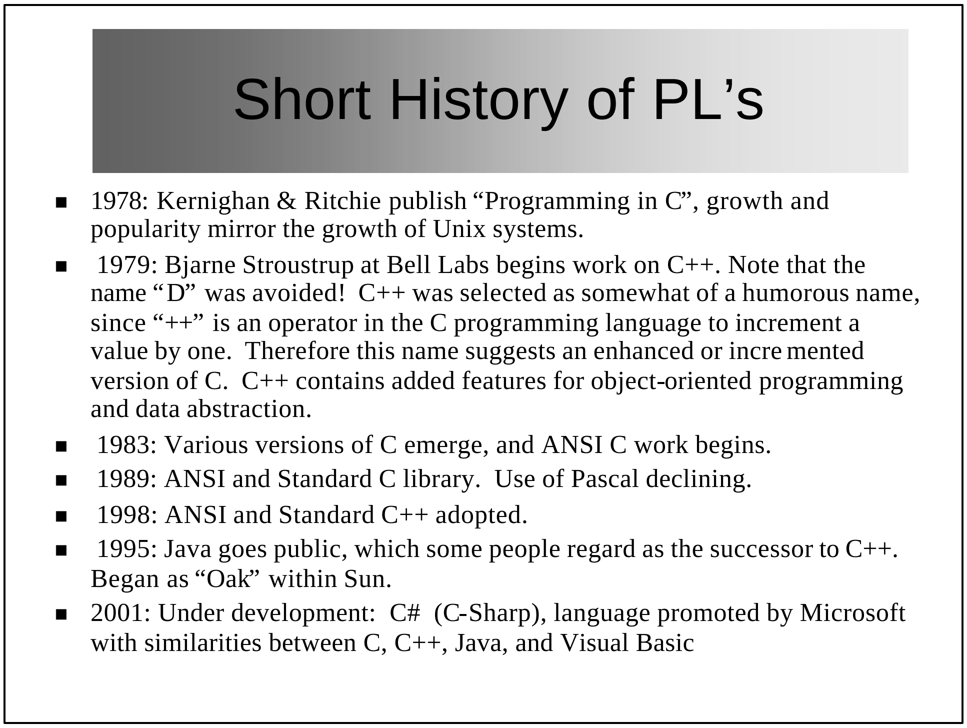
Software Development: A Broader View

- Measures of effort spent on real-life programs: Comparing programs by time:
  - Commercial software is seldom written by individuals.
    - **Person-months** - equivalent to one person working forty hours a week for four weeks.
    - **Person-years** - equivalent to one person working for twelve months.
    - Team of 5 working 40 hours for 8 weeks = ten person-months.
- Much more on these issues in the software engineering course



Short History of PL’s

- 1958: Algol defined, the first high-level structured language with a systematic syntax. Lacked data types. FORTRAN was one of the reasons Algol was invented, as IBM owned FORTRAN and the international committee wanted a new universal language.
- 1965: Multics – Multiplexed Information and Computing Service. Honeywell mainframe timesharing OS. Precursor to Unix.
- 1969: Unix – OS for DEC PDP-7, Written in BCPL (Basic Combined Programming Language) and B by Ken Thompson at Bell Labs, with lots of assembly language. You can think of B as being similar to C, but without types (which we will discuss later).
- 1970: Pascal designated as a successor to Algol, defined by Niklaus Wirth at ETH in Zurich. Very formal, structured, well-defined language.
- 1970’s: Ada programming language developed by Dept. of Defense. Based initially on Pascal. Powerful, but complicated programming language.
- 1972: Dennis Ritchie at Bell Labs creates C, successor to B, Unix ported to C. “Modern C” was complete by 1973.



Short History of PL’s

- 1978: Kernighan & Ritchie publish “Programming in C”, growth and popularity mirror the growth of Unix systems.
- 1979: Bjarne Stroustrup at Bell Labs begins work on C++. Note that the name “D” was avoided! C++ was selected as somewhat of a humorous name, since “++” is an operator in the C programming language to increment a value by one. Therefore this name suggests an enhanced or incremented version of C. C++ contains added features for object-oriented programming and data abstraction.
- 1983: Various versions of C emerge, and ANSI C work begins.
- 1989: ANSI and Standard C library. Use of Pascal declining.
- 1998: ANSI and Standard C++ adopted.
- 1995: Java goes public, which some people regard as the successor to C++. Began as “Oak” within Sun.
- 2001: Under development: C# (C-Sharp), language promoted by Microsoft with similarities between C, C++, Java, and Visual Basic