A Brief History of Computer Programming
Software Generations and Programming Languages

Notes:
Before we dive into the Java programming language itself, let's take a look at where Java came from, on the theory that George Santayana was right when he wrote, “Those who ignore history are doomed to repeat it.”

It's almost impossible to understand why Java works the way it does without understanding a little bit about other computer programming languages—both modern and "ancient"—and how and why they were developed. In this lecture, you'll learn a little about the history of computers, and computer programming.

As you saw in previous lectures, modern computer systems (in general) consist of at least four pieces:

- A processing unit (CPU) that performs simple arithmetic and comparisons.
- Memory that can store information: both data and instructions used by the processing unit.
- Input/Output devices that allow you to load information into memory and to see the results of your calculations.
- Storage devices that allow you to keep your calculations and your programs for later reuse.

The instructions that tell the processing unit which calculations to perform, and which information to use in those calculations, is called a program. Programming is the art (or science, or even black-magic, if you prefer) of creating programs.

Let's see what programs are, and where programming came from.

Notes:
Here's something most of you are familiar with. This is a recording of Dick Dale and His Del-Tones greatest hits from 1961 to 1976. This tune is called “The Wedge”. When I was young, I'd ride my bike down to Balboa and listen to Dick and the Del-Tones playing the Rendezvous Ballroom, and, if I wanted to listen to the music at home, I'd purchase a recording that looked a lot more like this—the analog audio signals from Dale's guitar were etched into a flat vinyl disc and were played back by amplifying the sound produced by tracing the grooves with a metal phonographic stylus. Of course, each time you spun the disc, the sound changed slightly as microscopic portions of the recording were scraped away. Audio aficionados though, still swear by these gramophone records, since the spiral groove cut into the record exactly reflects the analog waveform of the original sound.

Today, of course, vinyl recordings are a thing of the past. Instead, the Compact Disc shown here uses a digital recording technology, where the original sound wave is first sampled, and then reduced to a series of 1s and 0s, just like a
computer program. These 1s and 0s are then etched into a plastic platter using a laser to create a series of microscopic raised and lowered areas, called lands and pits. Once the CD is burned, you can listen to the music by using a CD player, like my daughter’s Walkman, which uses its own laser to read the microscopic binary signals, and then reconstruct the original sound wave recorded on the disc. Once the sound is decoded, the analog output is sent to your speakers or headset.

This is similar to the way your computer works. Like your computer, the CD player gets its input in the form of binary data. Like your computer, your CD player processes its input data, and then produces output. There are two big differences between your computer and the CD player, though:

1. Your CD player can’t do anything with its input except turn it into music. The processing part of the process is hard coded. If you try to “play” your Dreamweaver CD, the CD player doesn’t know what to do with the data.

2. Your computer can’t turn your audio CD into sound like your CD player can. That’s surprising, huh? Well, in fact, your computer doesn’t really “know” how to do much of anything. It can load some data from a disc, and maybe do some rudimentary arithmetic, but certainly nothing as complex as playing music or even typing a letter. Compared to your CD player, your computer is just plain “dumb”.

To do much of anything at all, your computer relies on programs. With the right program, you can play your audio CDs, as well as create Web pages, write your term paper, and browse the Internet. Without programs, your computer is just a paperweight.
instructions along with data processed by the computer. This idea, called the stored program concept, means that you can use the same computer to play PacMan and to balance your checkbook—you simply change its program.

This stored program concept was an almost immediate success, and was the beginning of machine language. In recognition of this fact, we often say that modern computers use the von Neumann architecture.

---

**Machine Language**

- All computers "understand" a specific set of instructions
  - These instructions are called the CPU instruction set
  - Instruction set for any CPU is its machine language
  - Each CPU family speaks a different machine language

---

**Notes:**

Every computer understands a particular, specific set of instructions, what we refer to as the CPU's instruction set. Here's the instruction set for an EECS 427 processor, for instance. The individual instructions for even the newest CPUs, are really pretty rudimentary; there are instructions for fetching a byte from memory, and for storing a byte back in memory, as well as instructions for adding, subtracting, and multiplying, etc.

Since the commands in the instruction set are the only thing that the computer understand, we call the instruction set for a particular computer its machine language. These machine languages are not universal; each one is specific to a particular CPU.

Here is an Intel Pentium machine language program. The rows of numbers on the left are the memory locations (called addresses) where each instruction is stored. The columns of numbers and letters on the right are the actual machine language instructions. The instructions stored in memory are actually binary numbers, but here they are displayed in hexadecimal (base 16) notation to make them easier to read.

As you can see by looking at the illustration, (and from your experiences using Debug in previous lectures), machine language is not very much like human language. Machine language is a numeric language, because the memory inside your computer can only store numeric data. Even when you work with text (such as viewing a web page), the computer is working with binary numbers. Because of this, writing machine language programs is very slow, tedious, and error-prone. Let's take a look at that process.
Suppose your boss asks you to write an application for a large department store. Part of the program requires you to calculate the total price of all purchases, including sales tax. To write this program in machine language, you'll have to mentally perform several translation steps.

The first step is to develop a set of unambiguous instructions or procedures that correctly calculate the price. These logical instructions are called an algorithm. It's important to perform this part carefully, because if you don't identify the correct steps and the correct order, your program will produce incorrect results, just as if you'd made a mistake in the steps required for a recipe.

A correct algorithm isn't sufficient to create a correct program, however. As a machine language programmer, you then need to translate that algorithm into the individual instructions that your CPU can perform, such as loading values from memory and adding, subtracting and multiplying.

After you've completed that task, though, you're still not finished. To tell the machine to load a value from memory, or to multiply two numbers, you'll need to look up the actual binary OP-code for each instruction, and enter that into memory. Thus to write a machine-language programmer, you actually perform three different kinds of translations:

- from the original program to an algorithm that solves the problem
- from the solution algorithm into the instruction-set of the CPU
- from the instruction-set to the actual binary OP-codes that the CPU can understand.

Despite the difficulties of using machine language, the number and size of the programs written expanded dramatically in the late 1940s and early 1950s. Machine-language programs were difficult to write, but creating them was a whole lot easier, and certainly less expensive, than building new "hard-wired" computing machines or employing thousands of "human computers" performing calculations the hard way.

These early "lean-and-mean" machine language programs quickly though became very large. Soon, they were too large to easily understand and maintain. If something goes wrong in one part of a machine language program, the programmer has to create a print-out showing the values in each memory cell when the error occurred—(this is called a "core dump"). That's what you see when you use the -d command in debug.

Core dump in hand, the programmer then must translate the values stored in memory into the basic instructions that the computer can perform: adding two numbers, perhaps, or...
storing a value at a particular location. Only after the raw machine values are translated into their corresponding computer instructions is the programmer ready to unravel the problem.

Programmers soon discovered that the computer itself could be put to work performing the painstaking but tedious task of translating memory values into the corresponding mnemonic "operation code" (or op code), so that one part of the debugging chore was lightened. That discovery lead to the next generation of ancient computer languages.

**Notes:**
Remember that a program is just a set of instructions, stored in memory that the computer can obey. The computer, however, can only understand machine language, nothing else at all. That creates a big problem from our point of view, because writing machine language programs is tedious and error prone for humans beings. We have to store exactly the correct numeric instruction in exactly the correct memory location; if you are off by even one byte, your program will crash, and, if you're lucky, only lock up the computer.

Although meticulously typing numbers into memory is hard for us, it's not hard for the computer. This led early computer designers to develop assembly language—a human-readable form of machine language. Programmers still had to create an algorithm, but in the second step, assembly language programmers typed in short mnemonic instructions, like MOV AH, 3E, which tells the computer to store a value in one of the CPUs registers. Then, instead of looking up the actual machine language instruction, B43E, the programmer translates the mnemonic instruction into the actual machine language instruction, using a program called an assembler.

Both machine language and assembly language are low-level languages. This means that they are specific to a particular brand of CPU. A machine language or an assembly language program written for Intel's Pentium processor won't run at all on the Power PC CPU in a pre-intel Mac, or the Sparc inside a Sun workstation.

**Notes:**
Computers follow their programmed instructions in a literal-minded, mechanical way. You can't just tell your computer to "print the budget report"; you have to explain every single tiny step. When you program in machine or assembly language, though, things seem even worse. Something as simple as printing a sentence on the screen can take half a page of code; and often, it's the same half page of code that you've written a dozen times already.

Can't the computer be put to work remembering all of those thousands of tiny details, so you, the programmer, don't have to? You bet; that's exactly what the early assembly-language programmers did. To lessen the burden of repetition, and to
increase productivity, programmers started to create libraries of code that performed common tasks.

Along with these libraries, they also started inventing "higher-level" versions of assembly language. In these higher level languages you could:

- Combine many lines of assembly code into a single instruction, called a macro. Instead of writing 50 lines of code to print a single line of output, you'd use only one.
- Avoid having to manually translate your program into assembly or machine language; instead, you used something called an interpreter.

Instead of using an assembler, these systems used a second program running on the computer to read each "high-level instruction" and produce machine code. This second program, (the interpreter), only generated machine code when the program actually ran.

These high-level interpreters not only made programmers more productive, but they also addressed another problem that afflicted machine and assembly-language programs: portability. Early computers were very expensive and individually built—they were definitely "one of a kind" machines. Because of this, when a new computer was developed, companies often found that the programs they'd written for their previous machines would not run on new models. The interpreter provided a clever solution to this problem: create an "ideal" machine language and then simply write an interpreter for the ideal language when a new machine was released. The most popular of these virtual machine languages were Speedcode (developed by John Backus at IBM), Shortcode (developed by John Mauchley of Eniac fame), and FlowMatic (developed by Grace Hopper).

Today, Java and Microsoft's new .NET platform both use the same concept. With Java, the virtual machine language is called bytecode, and the interpreter you use to run it is called a Java Virtual Machine or JVM. Microsoft .NET's virtual machine language is called MSIL (Microsoft Intermediate Language), and its interpreter is called the CLR or Common Language Runtime. We also use more traditional interpreters for things like HTML, shell scripts (or batch files) and JavaScript.

Notes:

By the end of the 1950s, both computers and interpreters had become widely entrenched in the business community. Interpreters and virtual assembly languages such as Speedcode, and Shortcode, allowed programmers to become much more productive. These much more productive programmers did what all productive people do—they produced more stuff; in the programmers' case they produced bigger and better programs.

Well, bigger anyway.

As programs got bigger, the weaknesses of the interpreter approach became obvious. Because so much of the
A programmer named Grace Hopper, is credited with an insight that seems obvious in retrospect. Instead of translating Speedcode into machine code every time you run the program, just do the translation once. Save the translated code on disk or tape, and reuse it every time you need to run your program. This invention was called the compiler. Today, most programming languages use some form of compiler.

One way to understand the difference between an interpreter and a compiler is to think about the different ways we convert between human languages. The interpreter is kind of like the interpreters at the United Nations. The speaker's words are translated as they occur. A compiler is more like the translator of a book. The translator produces a new manuscript that is independent of the original.

Hopper's language, called Flowmatic, was the last, and greatest, of the 2nd generation languages. Hopper is also credited with discovering the first computer bug when she was a Navy Lieutenant working at Harvard in 1945. Even though this is kind of a tongue-in-cheek joke, the actual bug along with the log book is currently in the Naval Historical archives.

Notes:

Once people realized that computers could translate virtual assembly languages like Speedcode and Flowmatic into machine code, they began to wonder if, perhaps, computers could do the same for more "natural" languages. (Natural for human beings, that is.) This marked the beginning of the third generation of computer languages, called High Level Languages, or HHL.

The basic idea behind a high-level language is straightforward: instead of writing a computer program in terms that the computer uses, write it in terms of the problem to be solved like this:

Note that with high-level languages, the programmer only has to do one translation: from the problem to the logical steps needed to carry out the task. These steps, you'll recall, are called an algorithm, and for this reason, high-level languages are often called algorithmic languages.

Since different people want to solve different kinds of problems, different high-level languages were developed. Let's look quickly at the "big four".
Early high-level languages are called **procedural**
- **FORTRAN**: FORMula TRANslator
  - 1955, IBM, scientific and engineering language
- **COBOL**: COrmmon Business Oriented Language
  - 1958, business data-processing language
- **BASIC**: Beginner's All-purpose Symbolic Instruction Code
  - Developed in 1960 at Dartmouth to teach programming
- **Pascal**: developed in 1970 as a better teaching language
- **C**: 1972, AT&T, developed as a general-purpose language

**FORTRAN**, which stands for FOrmula TRANslator, was developed in 1955 at IBM, as a way to let scientists and engineers write their own programs. FORTRAN is still widely used in the scientific community, but not that much in the Computer Science world.

**COBOL**, or the COrmmon Business Oriented Language, was released in 1958 by a consortium of Computer Manufacturers, along with the Department of Defense. COBOL was created to write accounting and business applications. Although there are versions of COBOL for personal computers, most COBOL programs run on mainframes, and handle “big jobs” like generating your utility bills each month.

**BASIC**, or the Beginner's All-purpose Symbolic lnformation Code was developed in the early 1960s at Dartmouth College to teach students the fundamentals of computer programming, using the newly invented interactive time-sharing systems.

Ten years later, in 1970, Pascal was developed as a better language to teach Computer Science concepts, as well as good programming habits. For 20 years, almost all beginning programmers learned Pascal as their first “real” programming language.

In 1972, the C programming language was developed at AT&T, as part of the development of the Unix operating system, and to operate the telephone switching system infrastructure.

C is a relatively simple programming language that requires both discipline and sophistication to use safely and successfully. (A strange, but common saying, (that even ended up as a book title), is that “C gives you enough rope to shoot yourself in the foot.” Today, almost all commercial software, such as Windows, Unix, Word, and Excel, is written in the C programming language.)