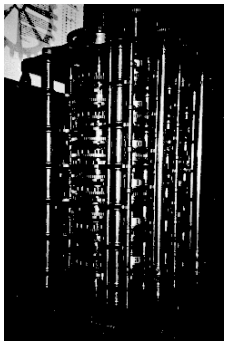
## A Brief History of Programming

- The earliest programmable computing machine was conceived by Charles Babbage (1792-1871). The first programmer was the Countess Ada Lovelace, daughter of the poet Byron.. She designed punch card programs for Babbage's Analytical Engine, which was never built.







## A Brief History of Programming

- One of the earliest electronic computers, ENIAC, was built and used back in the 40's.
- Originally, ENIAC had to be programmed by physically connecting different parts of it together. The "program" was literally "hard-wired". Reprogramming the machine required physically rewiring it. This was very time consuming.
- A man named John von Neumann, sometimes called "the father of computing", set up ENIAC so that it would retrieve "instructions" from its memory and then execute. This was the first time both the program and the data were stored in memory.
- ENIAC ran much slower in this configuration, but took only a small fraction of the time to program, since the program could now be fed in just like the data, on punched tape.
- This approach is so flexible that it has been adopted ever since. Virtually every computer built today uses what we call a "von Neumann" architecture.





## How Programs are Stored and Executed

- A CPU is capable executing a fixed set of *instructions* on data in its *registers*. Registers are tiny, fast memories contained right inside the CPU that can alter their contents based on the kind of instruction.
- There are only a handful of registers in a CPU. Registers in current machines process data in chunks of 32 bits. What people typically mean by a "32-bit processor" is that its registers process 32 bit chunks of data.
- The chunks of data processed by a CPU are often called *words*, and size of the chunks is the *word-length* of the CPU.
- The instructions in a CPU typically act on one or two registers. Some examples are:
  - load a value from primary memory into a register
  - store the value from a register into primary memory
  - add the contents of one register to another
  - compare the contents of two registers
- A *program*, at its simplest level, is a sequence of instructions for the CPU. Programs and the data they process are both stored in the primary memory.
- The CPU runs a program by fetching an instruction from the primary memory, executing the instruction, and then fetching the next instruction.
- Some instructions tell the CPU to "jump" to other parts of the program, sometimes depending on a value in the register. This allows the program to act differently depending on the data.





## First and Second Generation Languages

- The instructions used in computers are indicated by a set of bits (e.g. a byte can represent 256 different instructions).
- Early programs simply consisted a series of values fed directly into the memory. This is called *machine code*.
- Machine code is often entered as a set of hexadecimal (base 16) numbers. Each hexadecimal number represents 4 bits.
- Machine codes are also known as *first generation programming languages*.
- Machine code is very difficult for humans to understand. It was not long before people replaced the numeric values of instructions with mnemonics like "ADD" (addition) and "CMP" (compare). Programs were written that would translate these mnemonics in to the machine codes which could then be fed into the computer. These coding systems were known as *assembly languages* and fall into the category of *second generation programming languages*.
- Assembly languages made it easier for programmers to get things done, but there were still serious drawbacks.
- Certain operations were frequently performed but required a long set of instructions.
- Programmers also had to keep track of where everything was stored in memory and make sure that operations were not performed in incorrect sequences.

## Third Generation Languages

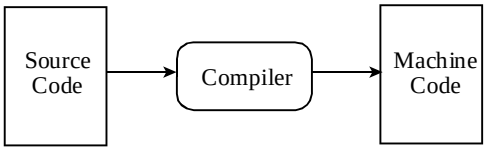
- The next leap in programming came in 1957 with FORTRAN. This language introduced the basic ideas that make up almost every language created since then. FORTRAN provided labels for memory locations and the task of deciding where in memory the things were stored was removed from the programmer. They simply referred to the labels.
- Furthermore, certain operations such a repetition and jumping around in the program were encoded in simpler terms. Now, the programmer could use one or two keywords for what had before been a long sequence of assembler instructions.
- This is a key difference. In assembly languages, there was a direct relationship between the assembler code and the machine code. In FORTRAN, a few simple keywords and labels was turned into a much larger collection of machine codes.
- FORTRAN formed the start of the *third generation programming languages* or *3GL*'s.
- Other examples of 3GL's include C, Pascal, COBOL, and many, many more.
- The question of whether any fourth generation languages exist today is open to debate. Some definitions for 4GL's have been proposed, but it's not obvious that any distinctly more abstract level has been well defined.





## Compilers

- Programming begins by writing a program in a programming language. This form of program is called *source code*.
- Computers are not capable of executing source code directly. It must be translated into machine code first.
- This translation is performed by a piece of software known as the *compiler*.



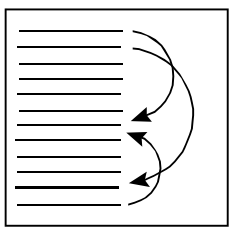

- Compilers handle the problem of where data is stored and what CPU instructions need to be executed.
- Compilers can detect errors in the *syntax* of the program. The syntax refers to the structure of the language, much like grammar in human languages.
- They can also provide warnings about source code that is likely to cause problems or be incorrect.
- However, the compiler can't detect logical errors. It is very easy to write a meaningless program that does nothing and have a compiler cheerfully accept it.





## The Software Crisis

- The development of third generation programming languages allowed the creation of larger systems than were feasible before. This was not because earlier languages were limited, but simply because the task of managing larger problems was easier.
- However, as larger and larger systems were constructed, it was found that early languages like FORTRAN and COBOL were being used to build large programs which became unmanageable. This problem became known as the *software crisis*
- The reasons were:
  - hard to find errors in the program
  - hard to keep track of how it worked
  - hard to understand the interactions between different parts
  - hard for more than one person to work on the same program
- The problem was the programs were disorganized. They consisted of long sequence of source code with lots of jumping around.

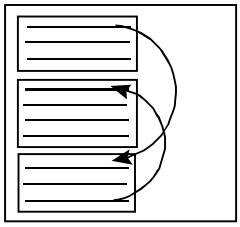






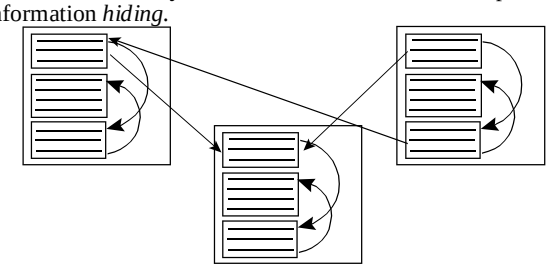
## Structured Programming

- The solution that people came up with was to break programs down into smaller chunks, and to make sure that it was easy to understand how those chunks were entered and exited.
- The idea of breaking programs down is known as *modularity*. The general idea was to create chunks of program with names so that they could dealt with separately. These chunks could *call* each other in order to achieve their goals.
- Using calls and a few standard structures for doing things like repeating sections of code meant that programmers did not need to jump around in the code as much, and that when one did, it was easy to understand what would happen.
- The approach is known as *structured programming*, and was first introduced in languages such as Algol and Pascal although elements of it pervade most modern languages..
- In structured programming, pieces of data are moved around from one part of the program to the other. Data is kept in distinct chunks, rather than all in one place, which further simplifies the problem of writing large programs.



## Modular Programming

- While structured programming solved a lot of problems, large systems were still difficult to build and maintain.
- Additionally, people were rewriting pieces of programs that already existed elsewhere. This was clearly wasteful. People wanted *reusability*.
- The idea of *modules* (or *libraries)* came into existence in languages such as Modula. Modules are even larger chunks of structured code that can call each other and be combined.
- Modules can also be written in different languages and combined later.
- A new piece of software is needed to link modules together. It is called a *linker*. After the compiler compiles each module, a linker takes all of the compiled modules and combines them together to make the final executable program.
- With modules, one need only see and learn the sections one wants and not worry about the rest. This is the concept of information *hiding*.







## Object-Oriented Programming

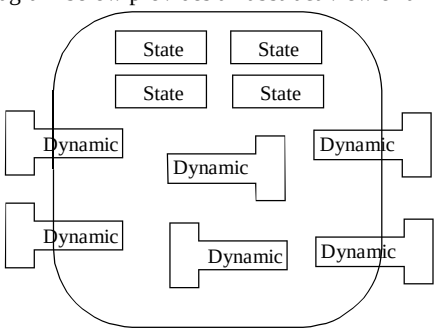
- Much of the software in the world today has been built using modular programming. However, even modular programming has encountered problems.
- The chief problem with earlier approaches was that data had to flow around through many sections of code. A lot of work was required to make sure that the data was not incorrectly processed by any part of the system. It was hard to understand the interactions of different parts of the system.
- Eventually, the concept of *object-oriented* programming (OOP) emerged. In OOP, one breaks the system down into a set of *objects*. Each object contains some data needed for the problem. The objects also have all appropriate processing attached to them.
- This means that every object *encapsulates* both the state and the dynamic for part of the problem.
- Because the object controls both the state and the dynamics, it can control how its state is changed, and ensure that invalid processing does not occur. This minimizes the risk of corrupting the data and means that if a failure occurs, we need only look at the objects involved to see where the fault occurs.
- Objects limit what can be done to them by exposing only part of their state and dynamic to other objects. The rest is hidden inside the object.
- This has another useful consequence. It means we can change the internals of an object without affecting the rest of the system.





## Object-Oriented Programming

- Another nice feature of OOP is that if we need to add more to the system, we simply create new objects. We don't necessarily need to change the existing objects.
- Because objects are nicely encapsulated, we can use them in other systems without worrying about the details of their implementation. This gives us good *reusability*.
- You can think of an object like a little machine. It has a number of mechanisms inside that you can't see and some buttons and dials on the outside that you can push and turn to make the machine do things. Some lights on the outside can tell you something about what's going on inside.
- With this view from the outside, you can safely worry only about **what** the machine is doing, not **how** it is doing it.
- The diagram below provides an abstract view of an object.





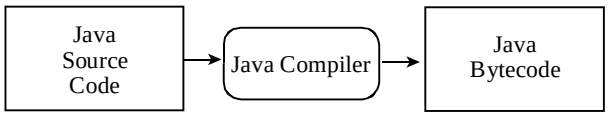


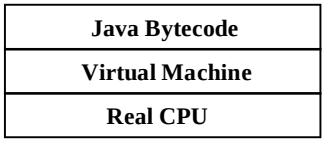

## Java: Object-Oriented and more

- Smalltalk was the first widely acknowledged object-oriented language. It uses an extremely object-oriented point of view and is not used in many large-scale, practical applications, although it is used for building prototypes of systems. Other languages, such as C++, have found wider usage in the industry.
- In the early 90's, the company, Sun Microsystems, wanted to create a language for *embedded devices*, such as cellular telephones. Such systems need very reliable and predictable software.
- Embedded systems also use a wide variety of hardware, so Sun needed a language that would perform consistently over a wide variety of *platforms*.
- A platform is a hardware/software environment in which programs are run. We frequently refer to running software *on a platform*.
- The ability to run on a wide variety of different platforms is called *platform-independence*.
- The language they started to create was called *Oak*. However, Oak failed to attract the attention they wanted. So, in order to promote the language, and to cash in on the Internet boom, they repackaged Oak as "the language for the Internet", and called it Java. By getting big Internet players like Netscape and Microsoft to add Java capabilities to their web browsers, Java became an overnight success.
- But what is it that sets Java apart from other languages?

## Java: Platform Independence

- One of Sun's main goals with Oak and Java was platform independence. Many different CPU's are used in different computers, and they do not all have the same instruction set. So Sun decided that this language would run on what was called a *virtual machine*.
- A virtual machine is a piece of software that acts like a CPU. It has an instruction set, just like an ordinary CPU. The difference is that, since it is a piece of software, we can make it behave the same on every machine. If we do this, then when we want to run Java programs on another machine, we only need to write a new version of the virtual machine for that machine. All of our other programs can stay the same.
- The Java Virtual Machine (JVM) is said to *interpret* the *Java bytecode* that forms a Java program.








## Java: Other Features

- Java was supposed to be powerful and highly reliable. It achieved this using the following features:
  - simplicity of the actual language
  - consistency of the actual language
  - uses object-oriented paradigm
  - runtime checking for certain common errors
  - exception-based error handling
  - built-in support for multi-threading
  - automatic garbage-collection of unused memory
- However, there are some costs to using Java.
  - using a virtual machine slows down execution
  - runtime error checking slows down execution
  - automatic garbage-collection can be hard to control
  - must use the object-oriented paradigm
  - a fairly large runtime environment must be *ported* to the target platform
- For the Internet, Java needed graphical capabilities. The many different operating systems on the Internet all use different windowing systems to draw graphics on the screen. Sun attempted (rather poorly at first) to create a platform independent windowing system called the Abstract Windowing Toolkit (AWT).
- Sun has provided many other *application programming interfaces* (API's) for the Java platform to handle networking, security, distributed systems, sound, 3-D graphics, internet services, and much more.





## Summary Of Important Terms and Concepts

- the "von Neumann" architecture
- machine code
- registers
- programs
- first, second, and third generation languages
- source code
- compilers
- syntax
- the "software crisis"
- structured programming
- modular programming
- information hiding
- objects and object-oriented programming (OOP)
- encapsulation
- reusability
- Java and its features
- platforms and platform-independence
- virtual machines and the Java Virtual Machine (JVM)
- Java bytecodes
- application programming inteface (API)