COMP 14

Prasun Dewan[1]

1. Introduction

In this book, we will study how the computer can be programmed. To understand what that means, we must have a model of the computer. We must understand several basic computing concepts such as hardware, operating system, compiler, and interpreter. Clearly, a detailed, precise technical description of the computer is beyond the scope of this course – even an undergraduate degree is sometimes not sufficient to cover in-depth all aspects of a computer. So we will instead try to model the computer by drawing an analogy between it and a world we understand – the theater world. [2]

A Theater Model of the Computer

Consider a theater (Figure 1) staging not just plays but arbitrary "performances" such as concerts, speeches, talks, discussions, consultations, cooking lessons, and debates. Scripts may be stored in archival stores such as theater libraries, from which the performers may copy them to notebooks they carry that are faster to access during the performance. Some of these performances such as discussions are interactive, that is, involve the audience while the performance is being conducted, while others such as plays are non- interactive. In either case, we assume what the performers do is determined by a script for each performance. Moreover, in either case, special instructions may be provided at the start of or during a performance to determine which parts of the script are exercised. For instance, the script for a musical concert may include both fast and slow tunes, and the performers may be asked to focus on slow or fast tunes in different performances.

Each performance may involve one or more cooperating roles or performers, each of who is responsible for some aspect of the performance. The theater management is responsible for operating details such as determining which performances are staged, allowing audience members to see only the performances they are authorized to see, and ushering audience members into and out of performances. Stagehands help the performers execute their script.

[1] © Copyright Prasun Dewan, 2000.

[2] In the spirit of "All the world's a stage…" from As You Like It, Shakespeare.

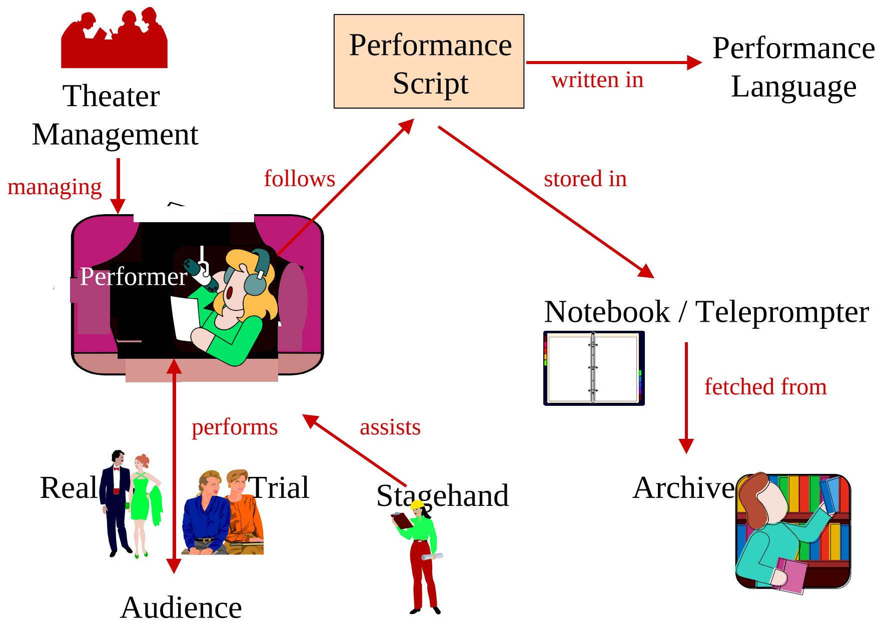

Figure 1-1 The Basic Theater World

In our theater world, scriptwriters are free to write in languages that the performers do not understand. Translators convert from the language of the scriptwriter to the language understood by the performers. A translator may do the complete translation before the performance is staged or do it sentence by sentence during the performance (Figure 2).

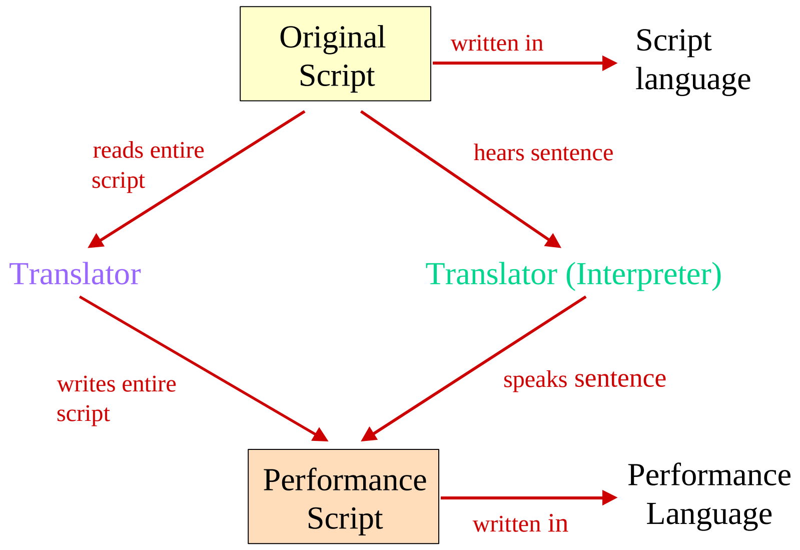

Figure 1-2 Performing a Script in a Different Language

A script may refer to reference material that the scriptwriter did not write. For instance, a state-of-the-union speech may refer to data from a report on the economy. Typically, the script will be linked to the reference material by a secretary (Figure 3). Several kinds of errors may be present in a script:

- *spelling* errors (e.g. cariage),
- *grammar* errors (such as " Me and Bobby McGee".),
- *semantic inconsistencies* among different parts of the script (such as a role called by two different names in a script, or the same name given to two different roles),
- errors having to do with the *execution* of the script (such as a performer being asked to fetch the moon!)

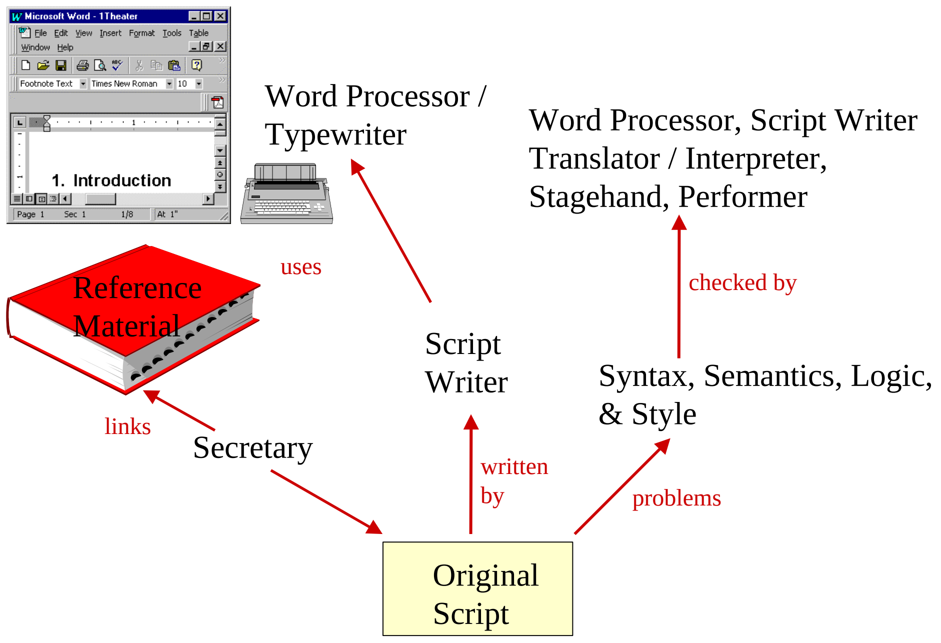
Microsoft Word - 1Theater
1. Introduction
Word Processor / Typewriter
Word Processor, Script Writer Translator / Interpreter, Stagehand, Performer
Reference Material
uses
checked by
Script Writer
Syntax, Semantics, Logic, & Style
links
Secretary
written by
problems
Original Script

Figure 1-3 More of the Theater World

Several agents may detect the errors in the script. For instance, the script writers themselves may detect all kinds of errors, word processors may detect spelling and grammar errors, translators/interpreters can detect the grammar errors and semantics inconsistencies, and the stage hands/performers would detect some of the execution errors at performance time. To reduce the likelihood of execution errors, it is important to have several trials before the final performance (Figure 1). Trials are particularly important if the performance is interactive, since unanticipated audience input may be received.

It is not enough to remove all errors in a script. It is also important to make it aesthetically pleasing by following principles of good style. For instance, though ''I and Bobby McGee'' is technically correct, we should avoid being egotistical and instead say: ''Bobby McGee and I''.

Theater Vs Computer World

This theater world has counterparts in the computer world (Figure 4, 5, 6).

Computer World	**Theater World**
Hardware	Theater
Operating System	Theater Management
Program	Performance
Processor	Performer
Instruction (e.g. add 2 to 5)	Performance action (e.g. walk 3 steps.)
Source Code	Original Script
Object Code	Performance Script
Programming Language	Script Language (e.g. German)
Machine Language	Performance Language (e.g. English)
Programmers	Script Writers
Library (of Code)	Reference Material (from Books)
Translator (Compilers/Interpreter)	Translator (Before/During Performance)
Users	Audience
Disks	Archival Storage Areas
Memory	Script performance notebook accessible to performers
Memory Page	A Notebook Page
Memory Word	A Notebook Line
Memory Address (Page Number, Word Number)	Line Identification (Page Number, Line Number)
Running a Program	Performing a Script
Interactive Program	Performance with audience participation
Non-interactive (Batch) Program	Performance with no audience participation
Program arguments	Special instructions at start of performance
Runtime	Stage-Hands
Editor	Typewriter/Wordprocessor
Editing Programs	Writing Scripts
Lexical Error	Spelling Error
Syntax Error	Grammar Error
Semantics Error	Inconsistencies in Script
Logic Error	Execution Error
Debugging	Staging trial performances
Style Principles	Style Principles

Table 1 Theater vs Computer World

The computer *hardware* provides the stage for executing programs. Programs are "performed" by *processors*, also called *central processing units (CPUs)*, which execute a variety of *instructions* such as addition, subtraction, multiplication, and division. The execution of programs is managed by the *operating system*, which allows authorized users to logon and execute the programs to which they have access.

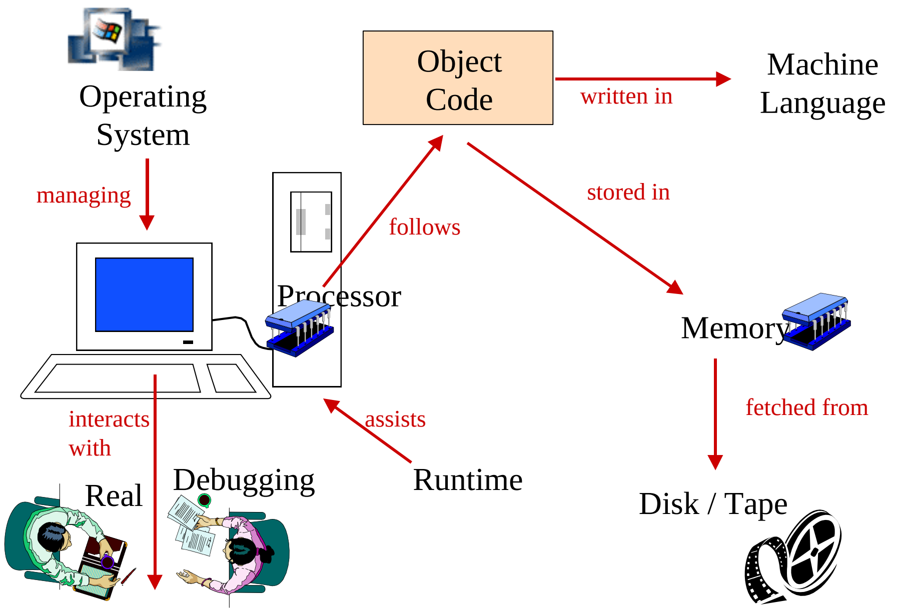

Figure 1-4 Computer Analog of Figure 1

A *program* is a script written in some *programming language*, and it must be *translated* into the *machine language* understood by the processor on which it executes. It is often combined with library code (written by some other programmer) by a *linker* before the processor executes it. The linkage can be done before the execution or *dynamically* when the library code is referenced by some instruction in the program.[3]

The *code* or script of an executing program is stored in *memory*, which is divided into a sequence of fixed-size chunks called memory *pages*, which in turn are divided into smaller, fixed-sized units called words. Each memory word has a memory *address*, and the address of the next instruction to be executed is kept in the *program counter*. Computer memory serves not only as a repository of program code but also as scratch paper for holding *data* executing programs need to compute their results.

Not all programs that we may want a computer to execute can fit simultaneously in its memory. Therefore, they tend to be stored in *disks*, which are one form of archival storage. (Tapes are another, slower but larger and more traditional, form of such storage.) Disks come in various forms – floppies, zip discs, hard discs, and compact discs. When a program is executed, the operating system copies it from disk and loads it into memory.

The results of programs are viewed by users; *interactive programs* receive input from users regarding how the computation should proceed, whereas non-interactive or batch *programs* do not offer this flexibility. In either case, parameters or *arguments* may be supplied to the program when it is executed to influence what it computes. For instance, a document name may be supplied as an argument to a word processor to indicate the document to be manipulated

The programming language implementation provides a piece of software, called the language *runtime*, which helps with program execution; for instance, gathering input from the user and reporting errors to users. It is called the runtime since it executes while the program is executing.

[3] The suffix `.dll` you see on some Microsoft files stand for *dynamically linked library*.

While a performance script may be written using a word processor, which has some knowledge of natural languages such as English and German, a program is written using an *editor*, which may have some knowledge of one or more programming languages (Figure 5).

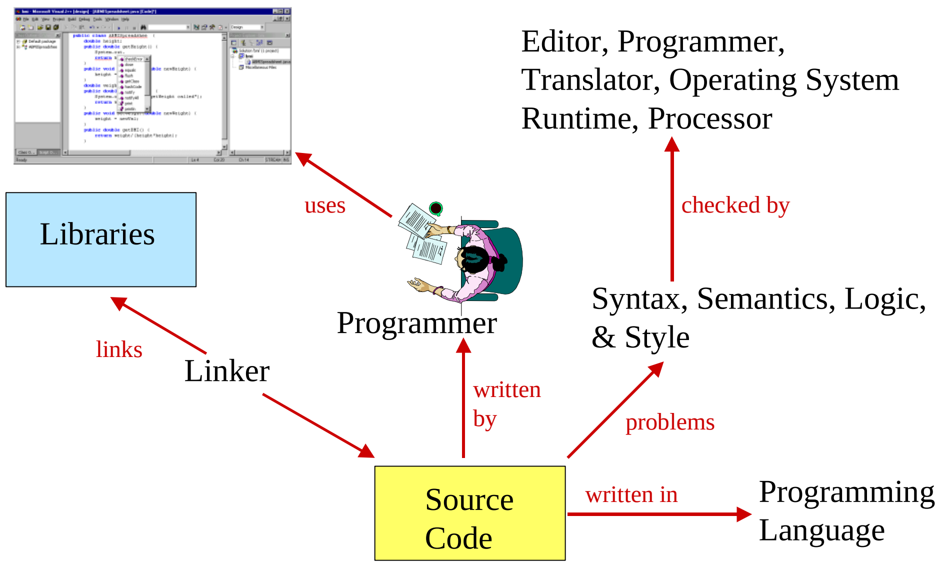

Figure 1-5 Computer Analog of Figure 2

As in the case of a performance, before a program is released, it is important to find mistakes or *bugs*,[4] which are discovered during the *debugging* phase.

Moreover, it is important to follow style principles, so that others, who may have to maintain our code, can understand and change it. Style principles can also allow software tools to understand our programs. We will see in this book an important example of such a tool, called ObjectEditor.

Thus, we can see there are many similarities between the computer and theater worlds. An important difference between them is that unlike the theater performers, the program performer, i.e. the processor, is naive (that is, it has no innate knowledge) but fast. Because it is naive, it has to follow the script and cannot improvise. As a result, it always gives the same performance for the same script and user input/arguments.

Another difference is that the language the processor understands is almost never the language a program writer uses. It understands a low-level binary machine language consisting of 1's and 0's while programmers normally write in higher-level programming languages, which are closer to natural languages such as English. Thus the role of a translator is very important in the computer world. A translator may be a *compiler* or an *interpreter* (Figure 7). A compiler does the complete translation before the program is executed, using the context of the complete source code in the translation process. An interpreter, on the other hand, does the translation during program execution, translating each instruction individually, just before it is executed. An interpreter can slow the performance down but has the advantage that it translates only part of the script that is actually performed. (Because of user input and arguments, not all parts of a program may be executed.)

---

[4] One of the early computer scientists, Grace Hopper, in the process of trying to determine why a relay was not working, traced the problem to a moth that was trapped in the relay, preventing it from closing. She filed it away in her logbook as the *bug* that caused the problem. Ever since, the word bug has come to denote in computing as a cause of an error.

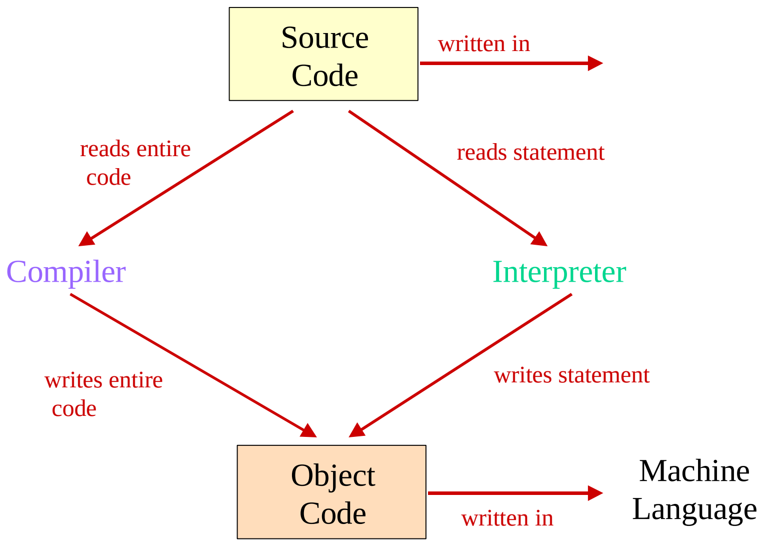

Figure 1-6 Computer Analog of Figure 3 - Program Translation

The translation process may involve both a compiler and an interpreter, with the compiler doing the major part of the translation before program execution, leaving the rest to the interpreter at execution time . This is a useful technique to support portability: The compiler translates the program into platform-independent *intermediate code*, which is then translated by a platform-specific interpreter into the machine code for the specific platform, as shown in Figure 8.

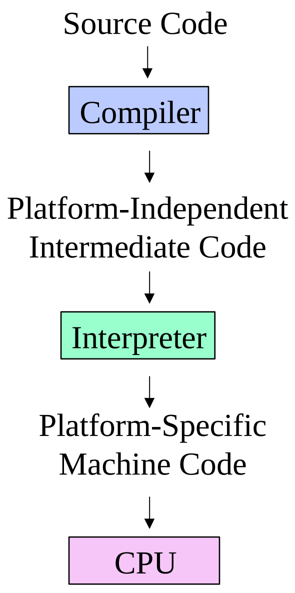
Source Code
Compiler
Platform-Independent Intermediate Code
Interpreter
Platform-Specific Machine Code
CPU

Figure 1-7: Translating using both a Compiler and Interpreter

Typically, the language used to generate intermediate code is very close to the machine language of most computers, making the task of implementing the platform-specific interpreter relatively easy. This is the approach taken in Java implementations, about which we will study in more detail later.

Because a program executes so instructions so rapidly, it is difficult to see effects of individual instructions, which is important when trying to find which one is at fault in an erroneous program. Therefore, a special tool, called the *debugger*, is often provided to trace program execution during the debugging process. A debugger allows you to *single-step* through the program, that is, suspend execution after each step of the program, thereby allowing you to see the effects of each program step. Thus, it essentially allows you to see the program execution is slow motion, so that you can better study what it is doing. It also allows you to annotate one or more program instructions as *break points.* When program execution reaches a break point, the debugger suspends the execution so that you can see the results of the program so far. Once you completely observed the effects, you can resume execution.

Why Java?

In this course, we will study programming, which means, we need to study a programming language in which we can program. Like natural languages, there are several programming languages, such as FORTRAN, BASIC, COBOL, Pascal, Lisp, C, Ada, Modula, Eiffel, Smalltalk, and C++. People brought up on different programming languages often have religious wars over which is best. We chose Java for this course because:

- It is a modern language, having the best features of most of the languages that have been developed so far. In particular, it has extensive capabilities for creating modular programs, that is, programs composed of smaller units much as a book or a play is composed of smaller units such as chapters, sections, paragraphs, and sentences. Modular programs are easy to write and understand since we focus, at any one point, on a small module of the program rather than the entire program.
- In comparison to C-based languages, it provides good error detection– an important requirement in an introductory course.
- It comes with a rich library, which embodies many of the programming principles we wish to cover in this course. Seeing concrete, useful implementations of the principles will hopefully make you follow them, just as watching a good tennis match can act as inspiration to play well!
- It provides special “reflection” facilities (which we will not actually study in this course), missing in most conventional languages, that make it possible to write teaching tools, in particular, the tool ObjectEditor, mentioned before.

Java has several other interesting features such as network support and integration with the Web, which are not relevant for an introductory programming class and thus will not be covered here.

The choice of the programming language is not that significant for this course, since the focus here is on learning programming concepts rather than some specific language. Java will simply be a means for learning programming. We will try to focus mainly on those concepts of Java that are found in other popular languages today and not address Java’s idiosyncrasies. Thus, this is a course on introductory programming and not Java.

As with natural languages, programming languages are related in that some languages have been developed from others. In particular, Java, C++ and other “object-based languages” have been developed from Smalltalk and Simula.

Moreover, like natural languages, there are dialects of programming. A dialect of a programming language can be caused by at least two reasons:

- A particular implementation adds to the vocabulary of the standard language through predefined libraries. Strictly speaking, libraries are not part of the language, just as a set of technical terms defined in some reference book is not part of the English language. The term Java Development Kit (JDK) is used for the combination of a version of the Java language and library. However, some of them are as important as the language, providing a set of system-provided tools on which the programmers come to depend. As a result, in practice, they are typically considered part of the language. Therefore, in this course, we will often use the term Java and JDK, interchangeably.

- The language writer sometimes leaves certain language features unspecified, which a particular computer system is free to fill in as it wishes; just as a script writer leaves certain performance details unspecified such as the size of the stage, the accent used to deliver lines, and the colour of the props provided by the stage hands.

Java has been designed to be portable, that is, provide the same features in all implementations. However, the portability goal remains elusive, with Microsoft and other vendors providing customized features. More important, the Java language and libraries continue to evolve rapidly, with each new version substantially differing from the previous one. In this course, we will focus mainly on fundamental concepts found in all versions of Java. However, the chapter on AWT and the tool, ObjectEditor, will assume version 1.1 of the Java Development Kit. ObjectEditor also assumes the Swing 1.1 library. Various versions of JDK for different can be downloadable from the site `http://www.javasoft.com/products/index.html`.

In addition to differences in languages and libraries, there are also differences in the Java *programming environment*s. A programming environment for a particular language is a software system that provides tools to support editing, compiling, interpreting, and debugging of programs in that language. While the behavior of the compilers and interpreters are determined by the language, a programming environment has complete freedom in the user-interfaces it provides to execute these programs, and in the kind of editors and debuggers it provides for the language.

Modeling the Computer Vs a Program

The theater analogy we studied here is independent of the programming language and environment used for developing and executing program code. Its main purpose is to explain the working of a computer, identifying its major hardware and software components. It treats a program as an indivisible black box, not dissecting its various parts, which tend to be language specific. Later we will look at analogies, based on our experiences with English, Math, and physical objects, which will explain the components of a Java program.

Summary

- In order to program a computer, it is important to have some understanding of a variety of hardware and software concepts such as CPU, operating system, program, object code, source code, libraries, compiler, interpreter, linker, memory, and disks.
- By drawing an analogy between the computer and theater world, we gain the extent of understanding we need to code and execute programs.
- We have chosen Java as the programming language because it has modern features, provides good error detection, comes with standard libraries that embody many of the style principles we will study here, and allows the writing of teaching tools, one of which we will use in this book.
- A programming language is only part of a programming environment, which also includes debuggers and editors.

Exercises

1. Define compiler, interpreter, linker, operating system, syntax error, semantics error, logic error, and style principle.