Chapter 1

Preliminaries
Chapter 1 Topics

• Reasons for Studying Concepts of Programming Languages
• Programming Domains
• Language Evaluation Criteria
• Influences on Language Design
• Language Categories
• Language Design Trade-Offs
• Implementation Methods
• Programming Environments
Reasons for Studying Concepts of Programming Languages

• Increased ability to express ideas
• Improved background for choosing appropriate languages
• Increased ability to learn new languages
• Better understanding of significance of implementation
• Better use of languages that are already known
• Overall advancement of computing
Programming Domains

- **Scientific applications**
  - Large numbers of floating point computations; use of arrays
  - Fortran
- **Business applications**
  - Produce reports, use decimal numbers and characters
  - COBOL
- **Artificial intelligence**
  - Symbols rather than numbers manipulated; use of linked lists
  - LISP
- **Systems programming**
  - Need efficiency because of continuous use
  - C
- **Web Software**
  - Eclectic collection of languages: markup (e.g., HTML), scripting (e.g., PHP), general-purpose (e.g., Java)
Language Evaluation Criteria

- **Readability**: the ease with which programs can be read and understood
- **Writability**: the ease with which a language can be used to create programs
- **Reliability**: conformance to specifications (i.e., performs to its specifications)
- **Cost**: the ultimate total cost
Evaluation Criteria: Readability

- **Overall simplicity**
  - A manageable set of features and constructs
  - Minimal feature multiplicity
  - Minimal operator overloading
- **Orthogonality**
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
- **Data types**
  - Adequate predefined data types
- **Syntax considerations**
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords
Evaluation Criteria: Writability

• Simplicity and orthogonality
  – Few constructs, a small number of primitives, a small set of rules for combining them

• Support for abstraction
  – The ability to define and use complex structures or operations in ways that allow details to be ignored

• Expressivity
  – A set of relatively convenient ways of specifying operations
  – Strength and number of operators and predefined functions
Evaluation Criteria: Reliability

• **Type checking**
  - Testing for type errors

• **Exception handling**
  - Intercept run-time errors and take corrective measures

• **Aliasing**
  - Presence of two or more distinct referencing methods for the same memory location

• **Readability and writability**
  - A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches, and hence reduced reliability
Evaluation Criteria: Cost

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Executing programs
- Reliability: poor reliability leads to high costs
- Maintaining programs
Evaluation Criteria: Others

- **Portability**
  - The ease with which programs can be moved from one implementation to another

- **Generality**
  - The applicability to a wide range of applications

- **Well-definedness**
  - The completeness and precision of the language’s official definition
Influences on Language Design

• Computer Architecture
  – Languages are developed around the prevalent computer architecture, known as the von Neumann architecture

• Program Design Methodologies
  – New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages
Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient
The von Neumann Architecture

Memory (stores both instructions and data)

Results of operations

Instructions and data

Arithmetic and logic unit

Control unit

Input and output devices

Central processing unit
The von Neumann Architecture

• Fetch–execute–cycle (on a von Neumann architecture computer)

  initialize the program counter

  repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction

  end repeat
Programming Methodologies Influences

• 1950s and early 1960s: Simple applications; worry about machine efficiency
• Late 1960s: People efficiency became important; readability, better control structures
  – structured programming
  – top–down design and step–wise refinement
• Late 1970s: Process–oriented to data–oriented
  – data abstraction
• Middle 1980s: Object–oriented programming
  – Data abstraction + inheritance + polymorphism
Language Categories

• Imperative
  - Central features are variables, assignment statements, and iteration
  - Include languages that support object-oriented programming
  - Include scripting languages
  - Include the visual languages
  - Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++

• Functional
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme, ML, F#

• Logic
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog

• Markup/programming hybrid
  - Markup languages extended to support some programming
  - Examples: JSTL, XSLT
Language Design Trade–Offs

• **Reliability vs. cost of execution**
  – Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs

• **Readability vs. writability**
  Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability

• **Writability (flexibility) vs. reliability**
  – Example: C++ pointers are powerful and very flexible but are unreliable
Implementation Methods

• Compilation
  – Programs are translated into machine language; includes JIT systems
  – Use: Large commercial applications

• Pure Interpretation
  – Programs are interpreted by another program known as an interpreter
  – Use: Small programs or when efficiency is not an issue

• Hybrid Implementation Systems
  – A compromise between compilers and pure interpreters
  – Use: Small and medium systems when efficiency is not the first concern
Layered View of Computer

The operating system and language implementation are layered over machine interface of a computer.
Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code
  - code generation: machine code is generated
The Compilation Process

1. Source program
2. Lexical analyzer
   - Lexical units
3. Syntax analyzer
   - Parse trees
4. Symbol table
5. Intermediate code generator (and semantic analyzer)
   - Intermediate code
6. Optimization (optional)
7. Code generator
   - Machine language
8. Computer
   - Input data
9. Results
Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program units and linking them to a user program
Von Neumann Bottleneck

- Connection speed between a computer’s memory and its processor determines the speed of a computer
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a bottleneck
- Known as the von Neumann bottleneck; it is the primary limiting factor in the speed of computers
Pure Interpretation

- No translation
- Easier implementation of programs (run–time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high–level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)
Pure Interpretation Process

Source program

Interpreter

Results

Input data
Hybrid Implementation Systems

• A compromise between compilers and pure interpreters
• A high-level language program is translated to an intermediate language that allows easy interpretation
• Faster than pure interpretation
• Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)
Hybrid Implementation Process
Compilation and Execution of a Java Program

Compilation

- **Parse**: Reads a set of *.java source files and maps the resulting token sequence into AST (Abstract Syntax Tree)-Nodes.
- **Enter**: Enters symbols for the definitions into the symbol table.
- **Process annotations**: If Requested, processes annotations found in the specified compilation units.
- **Attribute**: Attributes the Syntax trees. This step includes name resolution, type checking and constant folding.
- **Flow**: Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.
- **Desugar**: Rewrites the AST and translates away some syntactic sugar.
- **Generate**: Generates ‘.Class’ files.
Execution

• Class Loader:
  1) The main class is loaded into the memory by passing its “.class”
  2) All the other classes referenced in the program are loaded through the class loader.

There are two types of class loaders: primordial, and non-primordial.

Primordial class loader is embedded into all the JVMs, and is the default class loader.
A non-primordial class loader is a user-defined class loader, which can be coded in order to customize class-loading process. Non-primordial class loader, if defined, is preferred over the default one, to load classes.
Bytecode Verifier

- Variables are initialized before they are used.
- Method calls match the types of object references.
- Rules for accessing private data and methods are not violated.
- Local variable accesses fall within the runtime stack.
- The run time stack does not overflow.

If any of the above checks fails, the verifier doesn’t allow the class to be loaded.
Just-In-Time Compiler

- Convert the loaded bytecode into machine code.

- When using a JIT compiler, the hardware can execute the native code. This can lead to performance gains in the execution speed if methods are executed less frequently. Then you can let JVM interpret the same sequence of bytecode repeatedly incurring the penalty of a relatively lengthy translation process.
Just–in–Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile the intermediate language of the subprograms into machine code when they are called
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system
- In essence, JIT systems are delayed compilers
Preprocessors

• Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included

• A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros

• A well-known example: C preprocessor
  – expands `#include`, `#define`, and similar macros
Programming Environments

• A collection of tools used in software development
• UNIX
  – An older operating system and tool collection
  – Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that runs on top of UNIX
• Microsoft Visual Studio.NET
  – A large, complex visual environment
• Used to build Web applications and non-Web applications in any .NET language
• NetBeans
  – Related to Visual Studio .NET, except for applications in Java
Summary

• The study of programming languages is valuable for a number of reasons:
  – Increase our capacity to use different constructs
  – Enable us to choose languages more intelligently
  – Makes learning new languages easier
• Most important criteria for evaluating programming languages include:
  – Readability, writability, reliability, cost
• Major influences on language design have been machine architecture and software development methodologies
• The major methods of implementing programming languages are: compilation, pure interpretation, and hybrid implementation