

CENG204 - Programming Languages Concepts

Asst. Prof. Dr. Emre ŞATIR



Lecture 3

Preliminaries (Part 2)



Lecture 3 Topics

- Language Evaluation Criteria
 - Readability
 - Writability
 - Reliability
 - Cost
 - + Portability, Generality, Modularity, Flexibility
- Language Design Trade-Offs
- Other Influences on Language Design
 - Computer Architecture
 - Programming Design Methodologies



Language Evaluation Criteria



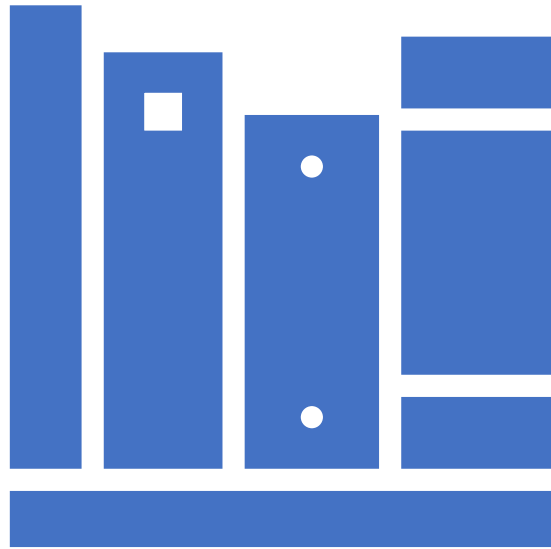


WHICH ONE?

Language Evaluation Criteria

- Evaluation criteria are needed to choose between programming languages according to the type of problem to be solved and the application area.
- Knowing the criteria that make a programming language different from others is very important for this choice.

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 under all conditions (i.e., performs to its specifications).
- **Cost:** The ultimate total cost.

+ Portability, Generality, Modularity, Flexibility.

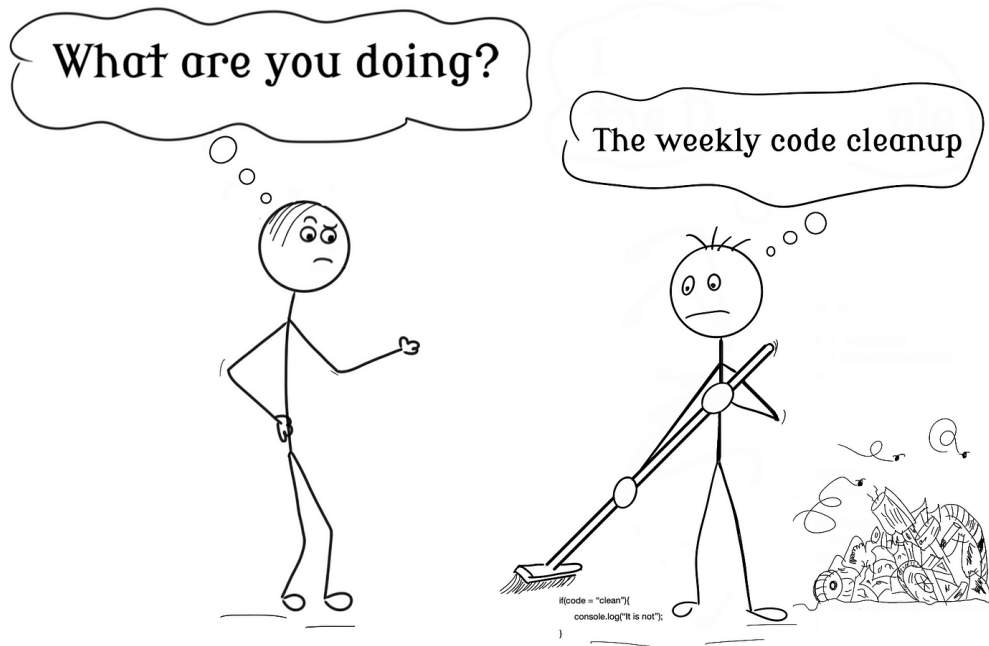
Language Evaluation Criteria

Table Language evaluation criteria and the characteristics that affect them

Characteristic	CRITERIA		
	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

*** The fourth primary criterion is cost, which is not included in the table because it is only slightly related to the other criteria and the characteristics that influence them.

Evaluation Criteria 1: Readability



- **Readability:** The ease with which programs (sourcecodes) can be “read” and “understood”.
- Readability makes updating easy.
- Readability allows many people to work together on common codes (teamwork).

*** Readability must be considered in the context of the **problem domain**. For example, if a program that describes a computation is written in a language not designed for such use, the program may be unnatural and convoluted, making it unusually difficult to read.

Evaluation Criteria 1: Readability

- The following subsections describe characteristics that contribute to the readability of a programming language:
 - Simplicity
 - Orthogonality
 - Data Types
 - Syntax Design

Characteristic 1: Simplicity



- The overall simplicity of a programming language strongly affects its readability.
- A language with **a large number of basic constructs** is more difficult to learn than one with a smaller number. Programmers who must use a large language often learn a subset of the language and ignore its other features.
- This learning pattern is sometimes used to excuse the large number of language constructs, but that argument is not valid. Readability problems occur whenever the program's author has learned a different subset from that subset with which the reader is familiar.
- So, a programming language should have a manageable set of features and constructs for simplicity.

Characteristic 1: Simplicity

- A second complicating property of a programming language is **feature multiplicity**—that is, having more than one way to accomplish a particular operation.
 - For example, in Java, a user can increment a simple integer variable in four different ways:
 - `count = count + 1`
 - `count += 1`
 - `count++`
 - `++count`
- (*** Although the last two statements have slightly different meanings from each other and from the others in some contexts, all of them have the same meaning when used as stand-alone expressions).
- A programming language should have minimal feature multiplicity for simplicity.

Characteristic 1: Simplicity

- A third potential problem is **operator overloading**, in which a single operator symbol has more than one meaning. Although this is often useful, it can lead to reduced readability if users are allowed to create their own overloading and do not do it sensibly.
- Suppose the programmer defined “+” used between single-dimensional array operands to mean the sum of all elements of both arrays. Because the usual meaning of vector addition is quite different from this, this unusual meaning could confuse both the author and the program’s readers.
- Minimal operator overloading should be preferred for simplicity.

Characteristic 2: Orthogonality

- Orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language.
- Furthermore, every possible combination of primitives should be legal and meaningful.
- Suppose a language has four primitive data types (integer, float, double, and character) and two type operators (array and pointer). If the two type operators can be applied to themselves and the four primitive data types, a large number of data structures can be defined.
- `int[5][2], float**, float*[4], etc.`

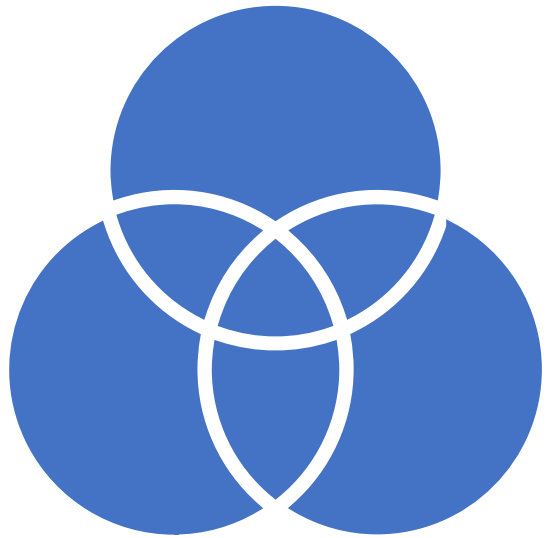
Characteristic 2: Orthogonality

- Orthogonality follows from a “symmetry” of relationships among primitives. A lack of orthogonality leads to “exceptions” to the rules of the language.
- For example, in a programming language that supports pointers, it should be possible to define a pointer to point to any specific type defined in the language.
- However, if pointers are not allowed to point to arrays, many potentially useful user-defined data structures cannot be defined.

Characteristic 2: Orthogonality

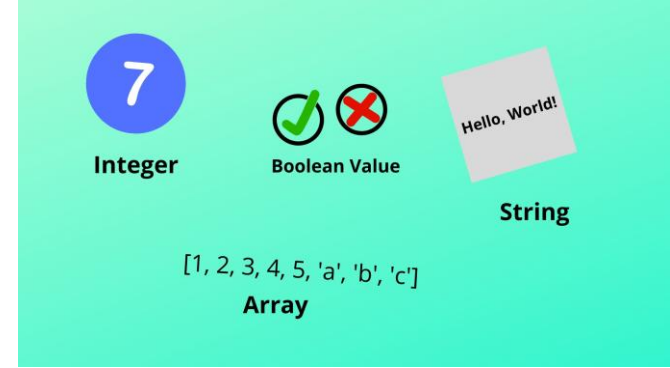
- As examples of the lack of orthogonality in a high-level language, consider the following rules and exceptions in C.
 - Although C has two kinds of structured data types, arrays and records (structs), records can be returned from functions, but arrays cannot.
 - A member of a structure cannot be a structure of the same type.
 - An array element cannot be a function.
 - Parameters are **passed by value**, unless they are arrays, in which case they are, in effect, **passed by reference**.

Simplicity and Orthogonality



- Orthogonality is closely related to simplicity: The more orthogonal the design of a language, the fewer exceptions the language rules require.
- Fewer exceptions mean a higher degree of “regularity” in the design, which makes the language easier to learn, read, and understand.
- This is a factor that increases readability.

Characteristic 3: Data Types



- The presence of adequate facilities for defining data types and data structures in a language is another significant aid to readability.
- For example, suppose a numeric type is used for an indicator flag because there is no Boolean type in the language. In such a language, we might have an assignment such as the following:
 - `timeOut = 1` (In C, this means true)
- The meaning of this statement is unclear, whereas in a language that includes Boolean types, we would have the following:
 - `timeOut = true` (For example, in Java)
- The meaning of this statement is perfectly clear.



Characteristic 4: Syntax Design

- The syntax, or form, of the elements of a language has a significant effect on the readability of programs. Following are some examples of syntactic design choices that affect readability:
 1. **Form and meaning:** Designing statements so that their appearance at least partially indicates their purpose is an obvious aid to readability. Semantics, or meaning, should follow directly from syntax, or form. A language should provide self-descriptive constructs, meaningful keywords for a good syntax design.
 - In some cases, this principle is violated by two language constructs that are identical or similar in appearance but have different meanings, depending perhaps on context. In Java, for example, the meaning of the reserved word **static** depends on the context of its appearance.

Characteristic 4: Syntax Design

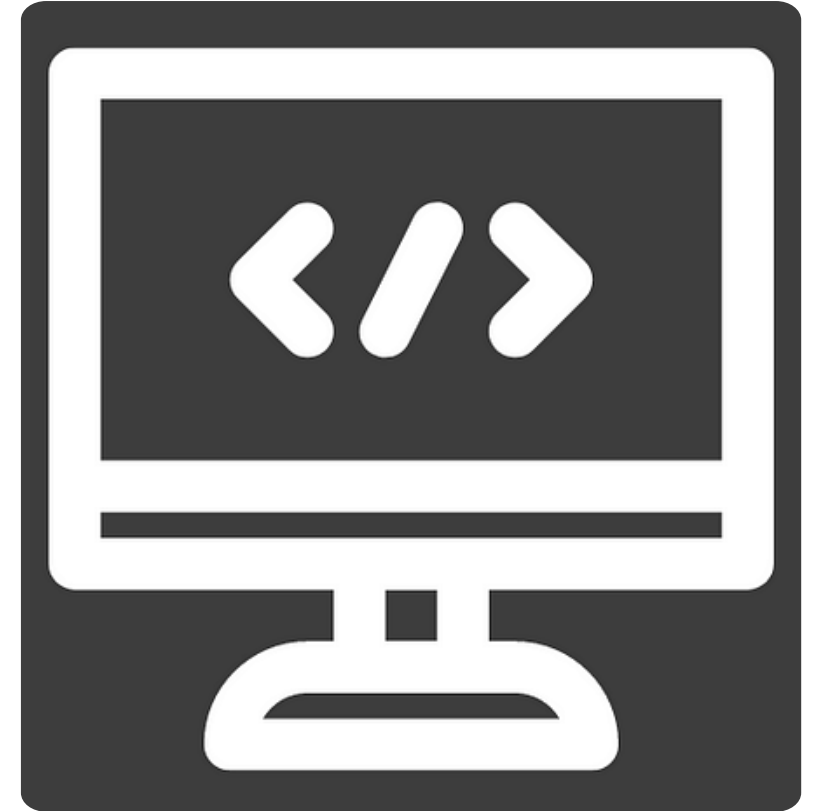
2. Special words (reserved words): Program appearance and thus program readability are strongly influenced by the forms of a language's special words (for example, “while”, “class”, and “for”).

- An issue: the method of forming compound statements, or statement groups, primarily in control constructs (Using “}” or “**end if**” at the end of an if block / Using “}” or “**end loop**” at the end of a loop block). This is an example of the conflict between simplicity that results in fewer reserved words, as in Java, and the greater readability that can result from using more reserved words, as in Ada.
- Another important issue is whether the special words of a language can be used as names for program variables. If so, the resulting programs can be very confusing. For example, in Fortran 95, special words, such as “**Do**” and “**End**”, are legal variable names, so the appearance of these words in a program may or may not connote something special.

Evaluation Criteria 2:

Writability

- **Writability:** Writability is a measure of how easily a language can be used to create programs for a chosen problem domain.
- Most of the language characteristics that affect readability also affect writability.
- *** As is the case with readability, writability must be considered in the context of the target problem domain of a language. It simply is not fair to compare the writability of two languages in the realm of a particular application when one was designed for that application and the other was not. For example, the writabilities of Visual BASIC (VB) and C are dramatically different for creating a program that has a graphical user interface (GUI), for which VB is ideal. Their writabilities are also quite different for writing systems programs, such as an operation system, for which C was designed.



Evaluation Criteria 2: **Writability**

- The following subsections describe the most important characteristics influencing the writability of a language:
 - Characteristics given up to this point (Simplicity, Orthogonality, Data Types, Syntax Design)
 - Support for Abstraction
 - Expressivity

Characteristics 1, 2, 3, 4: Simplicity, Orthogonality, Data Types, Syntax Design

- **Simplicity** reduces cognitive load, making code easier to write and understand.
 - On the other hand, for simplicity, the absence of some constructs in the programming language, may reduce expressivity and therefore writability.
- **Orthogonality** ensures that language constructs can be combined predictably, reducing special cases and increasing expressiveness.
 - On the other hand, too much orthogonality can be a detriment to writability. Errors in programs can go undetected when nearly any combination of primitives is legal. This can lead to code absurdities that cannot be discovered by the compiler.
- Well-structured **data types** provide meaningful abstractions, reducing errors and improving clarity in expressing computations.
- **Syntax Design** affects readability and ease of use (writability), as well-structured syntax, enhances code clarity and reduces syntax-related errors.

Characteristic 5:

Support for Abstraction

- In software engineering and computer science, abstraction is the process of removing or generalizing physical, spatial, or temporal details or attributes in the study of objects or systems to focus attention on details of greater importance.
- A programming language should provide this ability to define and use complex structures or operations in ways that allow details to be ignored for writability.



Characteristic 6: Expressivity

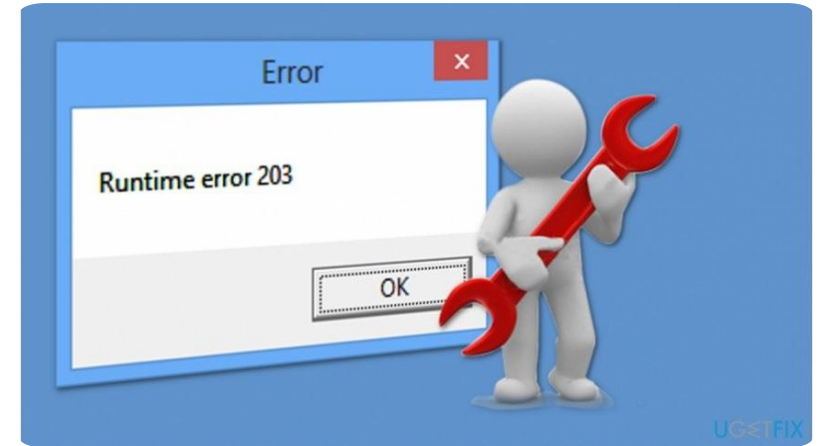


- A language that allows the person who designed the algorithm to clearly reflect the solution thought and intention is a language with high expression power.
- Expressivity in a language can refer to several different characteristics. It can be described as the strength and number of operators and predefined functions of a language.
- In a language such as APL, it means that there are very powerful operators that allow a great deal of computation to be accomplished with a very small program. More commonly, it means that a language has a set of relatively convenient ways of specifying operations.
- For example, in C, the notation `count++` is more convenient and shorter than `count = count + 1`.
- The inclusion of the “for” statement in Java makes writing counting loops easier than with the use of “while”, which is also possible.
- All of these increase the writability of a language.

Evaluation Criteria 3:

Reliability

- A program is said to be **reliable** if it performs to its specifications under all conditions.
- The following subsections describe several language features that have a significant effect on the reliability of programs in a given language:
 - Characteristics given up to this point (Simplicity, Orthogonality, Data Types, Syntax Design, Support for Abstraction, Expressivity)
 - Type Checking
 - Exception Handling
 - Restricted Aliasing



Characteristic 7: Type Checking

- **Type checking** is simply testing for type errors in a given program, either by the compiler or during program execution.
- Type checking is an important factor in language reliability. Because run-time type checking is expensive, compile-time type checking is more desirable.
- Furthermore, the earlier errors in programs are detected, the less expensive it is to make the required repairs.
- The design of Java requires checks of the types of nearly all variables and expressions at compile time. This virtually eliminates type errors at run time in Java programs.

**Type
Checking in
Compiler
Design**



Characteristic 8: Exception Handling

- The ability of a program to intercept run-time errors, take corrective measures, and then continue is an obvious aid to reliability. This language facility is called **exception handling**.
- Ada, C++, Java, and C# include extensive capabilities for exception handling, but such facilities are practically nonexistent in some widely used languages, for example C.

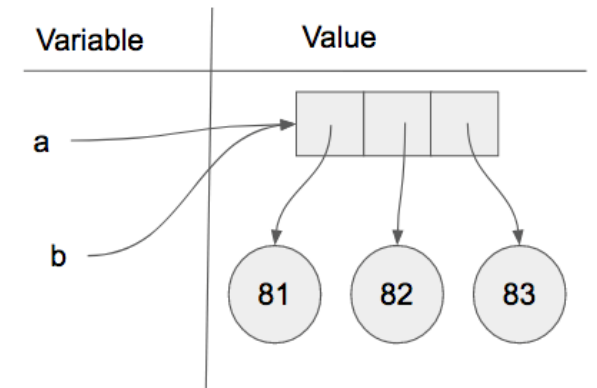
Exception Handling



```
try {  
    }  
  
catch (Exception e)  
    }
```

Characteristic 9: Restricted Aliasing

- Loosely defined, **aliasing** is having two or more distinct names in a program that can be used to access the same memory cell. (Presence of two or more distinct referencing methods for the same memory location).
- It is now generally accepted that aliasing is a dangerous feature in a programming language.
- Most programming languages allow some kind of aliasing—for example, two pointers (or references) set to point to the same variable, which is possible in most languages. In such a program, the programmer must always remember that changing the value pointed to by one of the two changes the value referenced by the other.
- Some kinds of aliasing, can be prohibited by the design of a language (They restrict alising).



Evaluation Criteria 4: Cost

- The total cost of a programming language is a function of many of its characteristics:
- **First**, there is the cost of training programmers to use the language, which is a function of the simplicity and orthogonality of the language and the experience of the programmers. Although more powerful languages are not necessarily more difficult to learn, they often are.
- **Second**, there is the cost of writing programs in the language. This is a function of the writability of the language, which depends in part on its closeness in purpose to the particular application. The original efforts to design and implement high-level languages were driven by the desire to lower the costs of creating software.

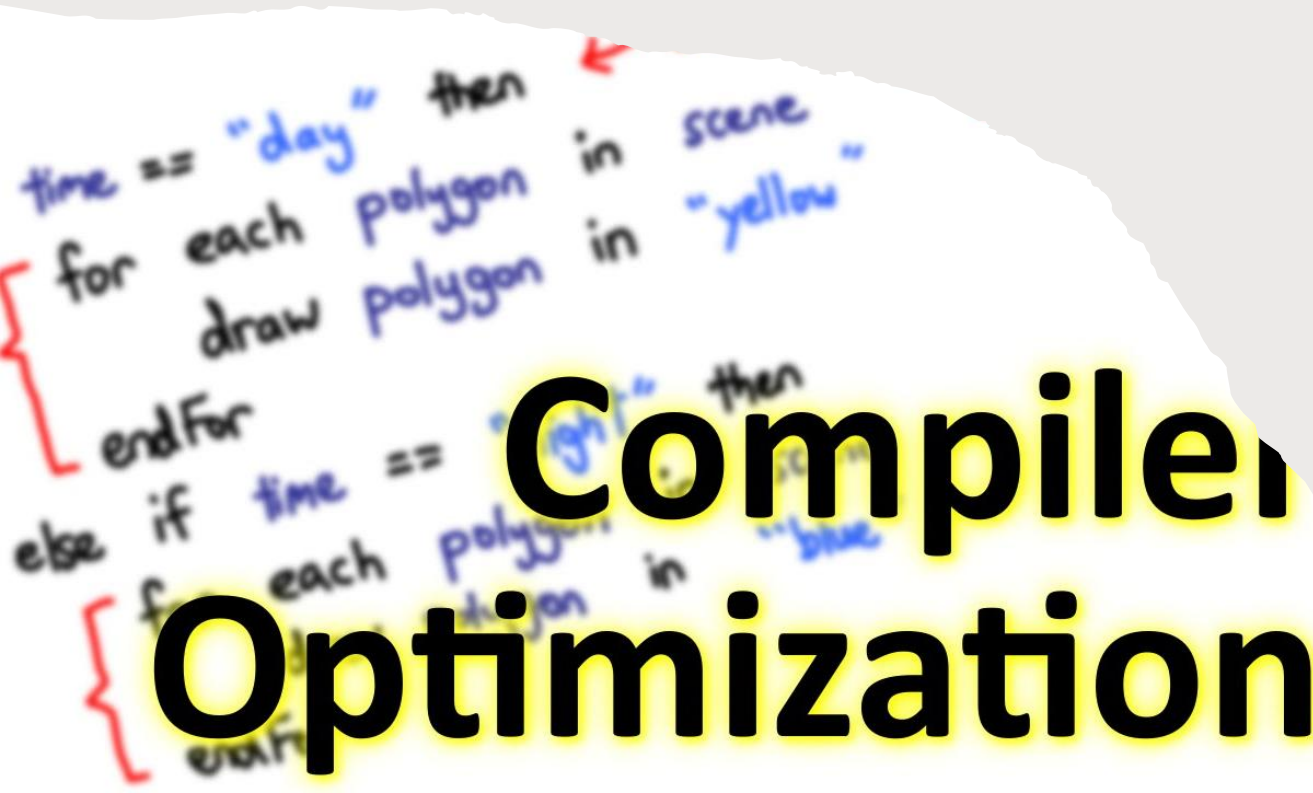


Evaluation Criteria 4: Cost

- **Third**, there is the cost of compiling programs in the language. A major impediment to the early use of Ada was the prohibitively high cost of running the first-generation Ada compilers. This problem was diminished by the appearance of improved Ada compilers.
- **Fourth**, the cost of executing programs written in a language is greatly influenced by that language's design. A language that requires many run-time type checks will prohibit fast code execution, regardless of the quality of the compiler.

Evaluation Criteria 4: Cost

- Trade-off : **compilation cost** vs **execution speed** → **Optimization**
- **Optimization** is the name given to the collection of techniques that compilers may use to decrease the size and/or increase the execution speed of the code they produce.
- If little or no optimization is done, compilation can be done much faster than if a significant effort is made to produce optimized code. The choice between the two alternatives is influenced by the environment in which the compiler will be used.
- For example, in a laboratory for beginning programming students, who often compile their programs many times during development but use little code at execution time (their programs are small, and they must execute correctly only once), little or no optimization should be done. In a production environment, where compiled programs are executed many times after development, it is better to pay the extra cost to optimize the code.



Handwritten code snippet on a torn piece of paper:

```
time == "day" then  
  for each polygon in scene  
    draw polygon in "yellow"  
  endFor  
else if time == "night" then  
  for each polygon in scene  
    draw polygon in "blue"  
  endFor  
endIf
```

The code is written in blue ink. Red curly braces are drawn on the left side, grouping the 'day' and 'night' blocks. A red arrow points to the 'scene' variable in the first 'for' loop.

Compiler Optimization

Evaluation Criteria 4: Cost

- The **fifth** factor in the cost of a language is the cost of the language implementation system. One of the factors that explains the rapid acceptance of Java is that free compiler/interpreter systems became available for it soon after its design was released. A language whose implementation system is either expensive or runs only on expensive hardware will have a much smaller chance of becoming widely used.
- **Sixth**, there is the cost of poor reliability. Poor reliability leads to high costs. If the software fails in a critical system, such as a nuclear power plant or an X-ray machine for medical use, the cost could be very high. The failures of noncritical systems can also be very expensive in terms of lost future business or lawsuits over defective software systems.

Evaluation Criteria 4: Cost

- The final consideration is the cost of maintaining programs, which includes both corrections and modifications to add new functionality.
- The cost of software maintenance depends on a number of language characteristics, primarily readability.
- Because maintenance is often done by individuals other than the original author of the software, poor readability can make the task extremely challenging.
- The importance of software maintainability cannot be overstated. It has been estimated that for large software systems with relatively long lifetimes, maintenance costs can be as high as two to four times as much as development costs.



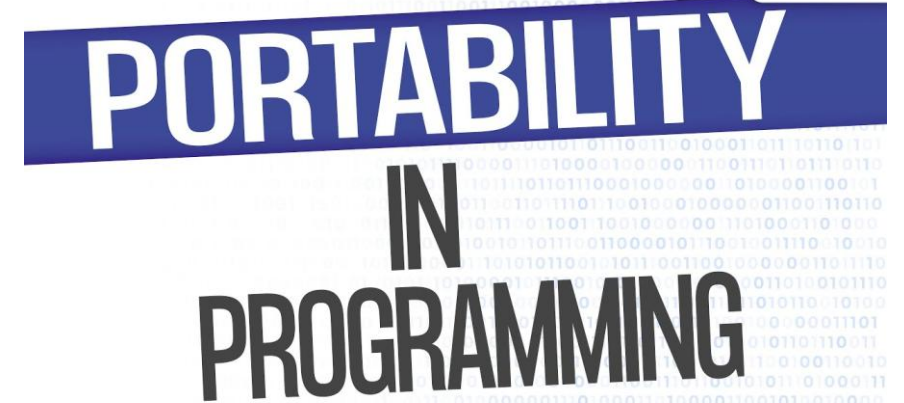
EVALUATION CRITERIA EXPLAINED

Evaluation
Criteria: The
Others

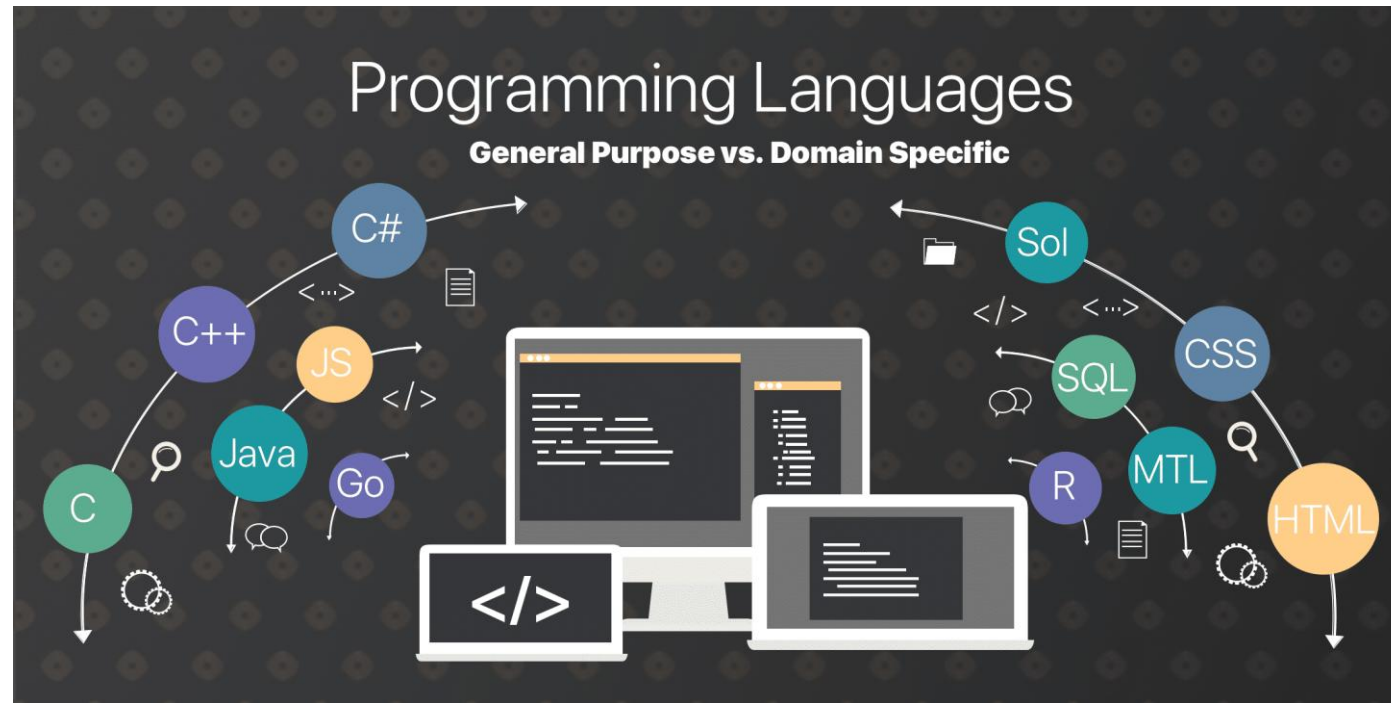
- Portability
- Generality
- Modularity
- Flexibility

Evaluation Criteria 5: Portability

- **Portability:** The ease with which programs can be moved from one implementation to another.
- The term portability here is used for source code.
- Portability is the ability to compile and run source code written in a programming language on other systems without any problems.
- As the level of languages decreases, portability decreases.
- Portability is most strongly influenced by the degree of standardization of the language. Some languages are not standardized at all, making programs in these languages very difficult to move from one implementation to another.
- **Standardization** is a time-consuming and difficult process. A committee began work on producing a standard version of C++ in 1989. It was approved in 1998.



PORTABILITY
IN
PROGRAMMING



Evaluation Criteria 6: Generality

- **Generality:** The applicability to a wide range of applications (domain independence).
- For example, while COBOL is an effective language in commercial applications, it is not preferred in engineering areas.
- Pascal, Basic, C, Java, C# can be qualified as “general-purpose languages”.

Evaluation Criteria 7: Modularity

- **Modularity:** The modularity capability is that a programming language supports writing the program in chunks.
- Thanks to the modularity, the code of the program becomes smaller because the most repetitive operations are put in modules.
- In addition, the code becomes more understandable thanks to subprograms.
- **Subprogramming** is a factor that facilitates testing, maintenance, and updating of source code.
- General purpose codes can be written with subprogramming and these codes can be used in more than one project. This is called “**reusability**”.





Evaluation Criteria 8: Flexibility

- **Flexibility:** The programming language should not restrict the programmer.
- Many operations in flexible languages are free for the programmer.
- While this feature will empower an experienced user, it will increase the risk of an inexperienced user making mistakes.
- C is a very flexible language. For example, in C language, an integer value can be assigned to a character and also a character value can be assigned to an integer (ASCII codes).

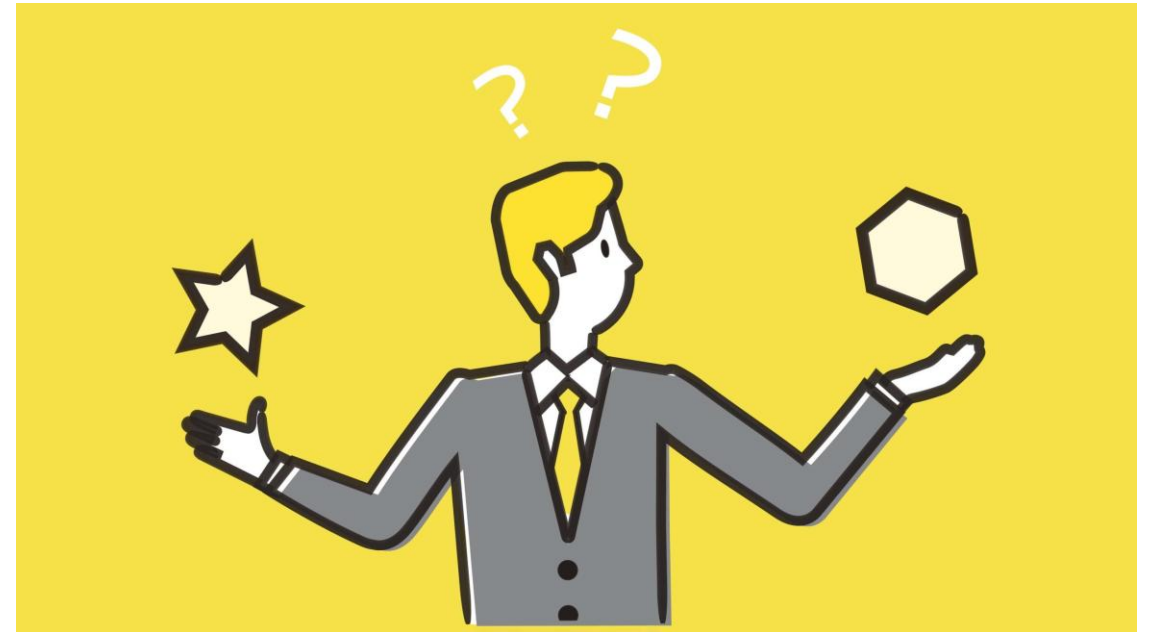


Language Design Trade-Offs



Language Design Trade-Offs

- The programming language evaluation criteria described before provide a framework for language design.
- Unfortunately, that framework is self-contradictory. In his insightful paper on language design, Hoare (1973) stated that “there are so many important but conflicting criteria, that their reconciliation and satisfaction is a major engineering task.”
- Designing a programming language is primarily an engineering feat, in which a long list of trade-offs must be made among features, constructs, and capabilities.



Language Design Trade-Offs

- **Reliability vs Cost of Execution**
- For example, the Java language definition demands that all references to array elements be checked to ensure that the index or indices are in their legal ranges. This step adds a great deal to the cost of execution of Java programs that contain large numbers of references to array elements.
- C does not require index range checking, so C programs execute faster than semantically equivalent Java programs, although Java programs are more reliable.
- The designers of Java traded execution efficiency for reliability.

An Example C Program

```
#include <stdio.h>

int main()
{
    int myArray[10];

    int i;

    for(i=0; i<10; i++)
        myArray[i] = i * 2;

    myArray[10] = 20;

    for(i=0; i<10; i++)
        printf("%d ", myArray[i]);

    return 0;
}
```

➔ error(s), 0 warning(s) (compiling)
➔ 0 2 4 6 8 10 12 14 16 18 (running)
Process returned 0 (0x0)

An Example Java Program

```
public class ExampleJavaClass
{
    public static void main(String[] args)
    {
        int[] myArray = new int[10];

        for(int i=0; i<10; i++)
            myArray[i] = i * 2;

        myArray[10] = 20;

        for(int i=0; i<10; i++)
            System.out.print(myArray[i] + " ");
    }
}
```

→ BUILD SUCCESSFUL (compiling)

→ Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
at ExampleJavaClass.main(ExampleJavaClass.java:10)
C:\Users\emre\AppData\Local\NetBeans\Cache\8.2\executor-
snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds) (running)

Language Design Trade-Offs

- **Writability/Expressivity vs Readability**
- For 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.
- One result of this high degree of expressivity is that, for applications involving many array operations, APL is very writable, but APL programs have very poor readability. A compact and concise expression has a certain mathematical beauty, but it is difficult for anyone other than the programmer to understand.
- The designer of APL traded readability for writability.

Language Design Trade-Offs

- **Writability/Flexibility vs Reliability**
- For example, the pointers of C++ can be manipulated in a variety of ways, which supports highly flexible addressing of data.
- C++ pointers are powerful and very flexible but are unreliable.
- Because of the potential reliability problems with pointers, they are not included in Java.



Other Influences on Language Design



Other Influences on Language Design

- In addition to those factors described before, several other factors influence the basic design of programming languages.
- The most important of these are:
 - Computer Architecture
 - Programming Design Methodologies

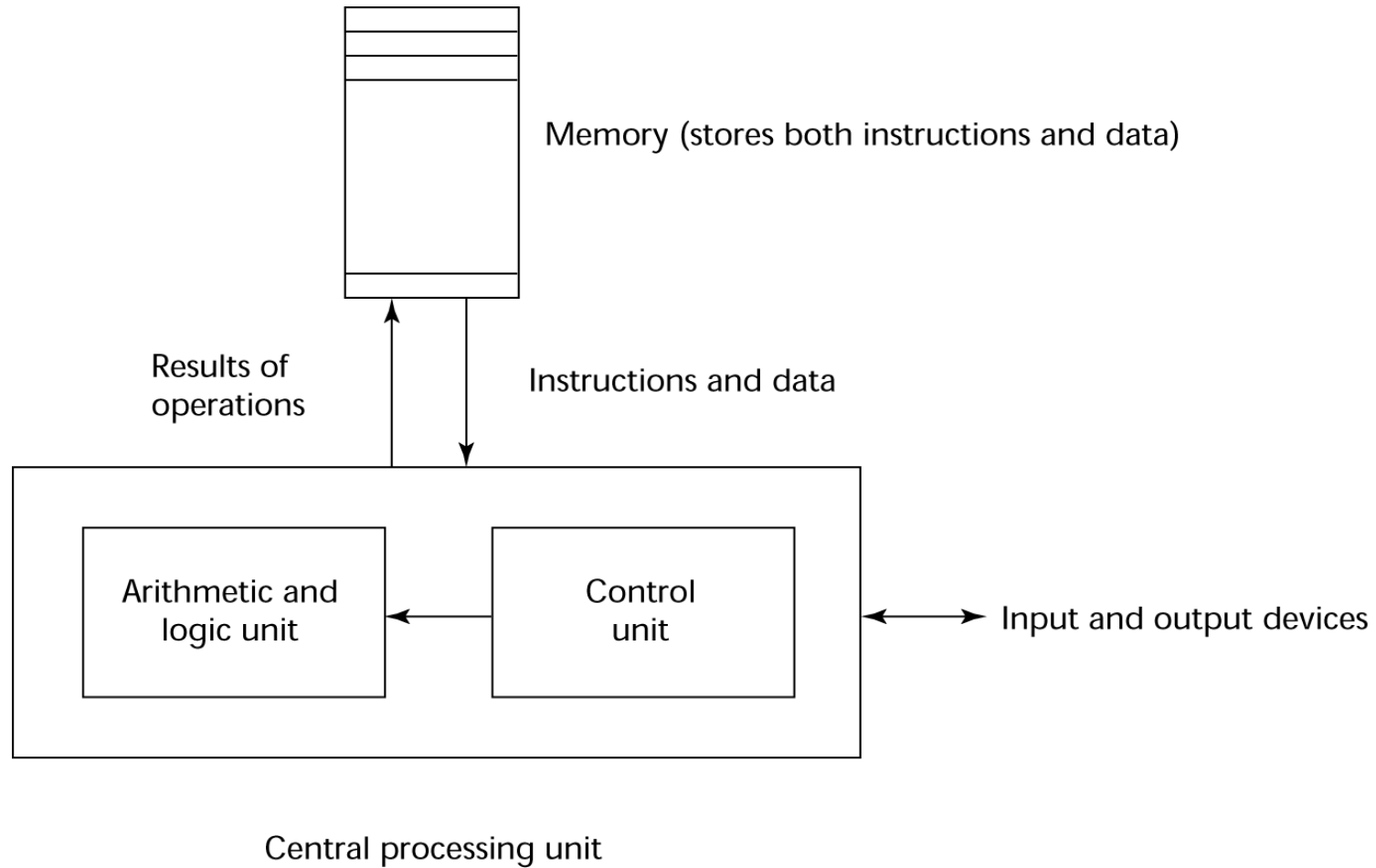


Computer Architecture Influence

- The basic architecture of computers has had a profound effect on language design.
- Most of the popular languages of the past 60 years have been designed around the prevalent computer architecture, called the **von Neumann architecture**, after one of its originators, **John von Neumann** (Imperative languages are most dominant, because of von Neumann computers).
- In a von Neumann computer, both data and programs are stored in the same memory. The central processing unit (CPU), which executes instructions, is separate from the memory. Therefore, instructions and data must be transmitted, or pipelined, from memory to the CPU. Results of operations in the CPU must be moved back to memory. Nearly all digital computers built since the 1940s have been based on the von Neumann architecture.



The von Neumann Architecture



The von Neumann Architecture

- The execution of a machine code program on a von Neumann architecture computer occurs in a process called the **fetch-decode-execute** cycle.
- As stated earlier, programs reside in memory but are executed in the CPU. Each instruction to be executed must be moved from memory to the processor.
- The address of the next instruction to be executed is maintained in a register called the **program counter**.
- The fetch-decode-execute cycle can be simply described by the following algorithm:

The fetch- decode-execute Cycle Algorithm

- initialize the program counter
- **repeat** forever
 - fetch the instruction pointed to by the program counter
 - increment the program counter to point at the next instruction
 - decode the instruction
 - execute the instruction
- **end repeat**



Computer Architecture Influence

- Because of the von Neumann architecture, the central features of imperative languages are **variables**, which model the memory cells; **assignment statements**, which are based on the pipng operation; and the **iterative form of repetition**, which is the most efficient way to implement repetition on this architecture.
- Operands in expressions are piped from memory to the CPU, and the result of evaluating the expression is pipd back to the memory cell represented by the left side of the assignment.
- ***Iteration is fast on von Neumann computers*** because instructions are stored in adjacent cells of memory and repeating the execution of a section of code requires only a branch instruction.

Computer Architecture Influence

- As stated earlier, a **functional language** is one in which the primary means of computation is applying functions to given parameters.
- Programming can be done in a functional language without the kind of variables that are used in imperative languages, without assignment statements, and without iteration.
- Although many computer scientists have expounded on the myriad benefits of functional languages, such as Scheme, it is unlikely that they will displace the imperative languages until a non-von Neumann computer is designed that allows efficient execution of programs in functional languages.

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).
- Late 1970s: Process-oriented to data-oriented. Simply put, data-oriented methods emphasize data design, focusing on the use of abstract data types to solve problems.
- Middle 1980s: Object-oriented programming
 - Data abstraction + inheritance + dynamic binding/polymorphism
- All of these evolutionary steps in software development methodologies led to new language constructs to support them.

