INTRODUCTION TO COMPILERS

Definition: A compiler is a program that translates high-level source code into machine code.

Phases of a Compiler:

- **Front-End**: Focuses on analyzing and understanding the code.
- **Back-End**: Focuses on optimization and generating executable code.

Importance: Helps convert human-readable code into efficient instructions for computers

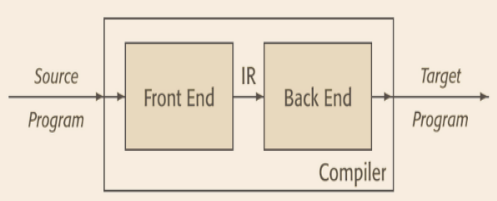

Fig 1. Two phase compiler Block diagram

THE FRONT-END OF A COMPILER:

Main Components:

- Scanner (Lexical Analyzer): Converts source code into tokens.
- Parser (Syntax Analyzer): Checks the grammar and structure of tokens.
- Semantic Analyzer: Ensures the meaning of code is valid (e.g., type checking).

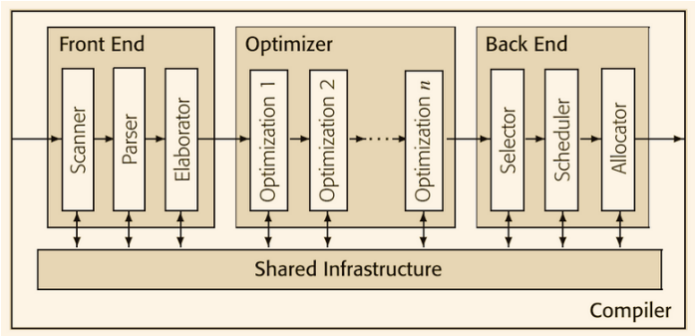

Fig 2. Detailed 3 phase compiler Block diagram.

TOKENS – DEFINITION AND CATEGORIES

Definition: Tokens are the smallest units of a program, generated during lexical analysis.

Categories of Tokens:

- Keywords (Reserved words like if, else).
- Identifiers (Variable or function names like x, y).
- Literals (Fixed values like 42, "Hello").
- Operators (+, -, =).
- Separators/Delimiters ({, }, ;).
- Comments (Ignored by the compiler).
- Whitespace (Used for formatting).

A) KEYWORDS

Definition: Keywords are reserved, predefined syntax(words) that holds a special meaning and are not used as identifiers in the computer programming.

a) Python Keywords

There are 35 keywords in Python that can't be used for variable/ function names. They serve to specify the language's structure and grammar. Key Categories of Python Keywords:

- Boolean and Special Literals: False, None, True.
- Logical Operators: not, or, and.
- Control Flow: if, elif, else, while, for, continue, break, return, try, except, finally.
- Function/Class Definitions: def, class, lambda.
- Exception Handling: raise, assert.
- Importing and Scope Management: import, from, global, nonlocal.
- Asynchronous Programming: async, await.

Table 1. Token Representation of Python Keywords

Category	Keyword Identified	Token Representation
Asynchronous Programming	async	(keyword, "async")
	await	(keyword, "await")
Boolean and Special Literals	False	(keyword, "False")
	None	(keyword, "None")
	True	(keyword, "True")
Context Management	with	(keyword, "with")
Control Flow	break	(keyword, "break")
	continue	(keyword, "continue")
	elif	(keyword, "elif")
	else	(keyword, "else")
	for	(keyword, "for")
	if	(keyword, "if")
	return	(keyword, "return")
	try	(keyword, "try")
	while	(keyword, "while")
Deletion	del	(keyword, "del")
Exception Handling	assert	(keyword, "assert")
	except	(keyword, "except")
	finally	(keyword, "finally")
	raise	(keyword, "raise")
Function/Class Definitions	class	(keyword, "class")
	def	(keyword, "def")
	lambda	(keyword, "lambda")
Generator Function	yield	(keyword, "yield")

Importing and Scope Management	from	(keyword, "from")
	global	(keyword, "global")
	import	(keyword, "import")
	nonlocal	(keyword, "nonlocal")
Keyword for Aliases/Contexts	as	(keyword, "as")
Logical Operators	and	(keyword, "and")
	in	(keyword, "in")
	is	(keyword, "is")
	not	(keyword, "not")
	or	(keyword, "or")
Null Operation	pass	(keyword, "pass")

Python Code/ Program:

```
class Person:
    def __init__(self, name, age):
        self.name = name  # 'name' is an identifier, 'self' is a special identifier
        self.age = age    # 'age' is an identifier
    def greet(self):
        if self.name:
            return f"Hello, {self.name}"
        else:
            return "Hello, World!"
# Creating an instance of the class
person = Person("Alice", 30)
print(person.greet())
```

Keywords: class, def, if, else, return

b) Java Keywords

Java has a set of **51 keywords** that also have reserved meanings. Key Categories of Java Keywords:

- Control Flow: if, else, while, for, switch, continue, break.
- Modifiers: private, protected, public, static, final, abstract.
- Data Types: int, char, boolean, double, float.
- Class and Object: class, interface, extends, implements.
- Exception Handling: try, catch, finally, throw, throws.
- Threading: synchronized, volatile.
- Access Modifiers: public, private, protected.

Table 2. Token Representation of Java Keywords

Category	Keyword Identified	Token Representation
Access Modifiers	private	(keyword, "private")
	protected	(keyword, "protected")

	public	(keyword, "public")
Class and Object	class	(keyword, "class")
	enum	(keyword, "enum")
	extends	(keyword, "extends")
	implements	(keyword, "implements")
	import	(keyword, "import")
	instanceof	(keyword, "instanceof")
	interface	(keyword, "interface")
	new	(keyword, "new")
	package	(keyword, "package")
	super	(keyword, "super")
	this	(keyword, "this")
Control Flow	break	(keyword, "break")
	case	(keyword, "case")
	continue	(keyword, "continue")
	default	(keyword, "default")
	do	(keyword, "do")
	else	(keyword, "else")
	for	(keyword, "for")
	goto	(keyword, "goto")
	if	(keyword, "if")
	return	(keyword, "return")
	switch	(keyword, "switch")
	while	(keyword, "while")
Data Types	boolean	(keyword, "boolean")
	byte	(keyword, "byte")
	char	(keyword, "char")
	double	(keyword, "double")
	float	(keyword, "float")
	int	(keyword, "int")
	long	(keyword, "long")
	null	(keyword, "null")
	short	(keyword, "short")
	void	(keyword, "void")
Exception Handling	catch	(keyword, "catch")
	finally	(keyword, "finally")
	throw	(keyword, "throw")
	throws	(keyword, "throws")
	try	(keyword, "try")
Modifiers	abstract	(keyword, "abstract")
	assert	(keyword, "assert")

	const	(keyword, "const")
	final	(keyword, "final")
	native	(keyword, "native")
	static	(keyword, "static")
	strictfp	(keyword, "strictfp")
	transient	(keyword, "transient")
Threading	synchronized	(keyword, "synchronized")
	volatile	(keyword, "volatile")

Java Code/ Program:

```
public class Person {
    private String name; // 'name' is an identifier
    private int age;     // 'age' is an identifier

    public Person(String name, int age) { // 'name' and 'age' are identifiers
        this.name = name; // 'this' is a special keyword
        this.age = age;
    }

    public String greet() {
        if (this.name != null) {
            // 'if' and 'return' are keywords
            return "Hello, " + this.name;
        } else {
            return "Hello, World!";
        }
    }
    public static void main(String[] args) {
        Person person = new Person("Alice", 30); // 'Person' and 'person' are identifiers
        System.out.println(person.greet());
    }
}
```

Keywords: public, class, String, private, int, if, else, return, static, void, new

B) IDENTIFIERS

Definition: Variables that are used to hold a value(s) in the programming language is referred to as Identifiers. These are symbols used to treat a particular value and can be called/ used whenever and wherever it is needed, provided that the call should be within the scope of that variable.

a) SIMILARITIES BETWEEN JAVA AND PYTHON

Both Java and Python have several common rules for identifiers:

- Starting Character:
  - Identifiers must begin with a letters/ character (A-Z, a-z) or underscore (_).
  - Identifiers cannot start with a digit/ number (0-9).
- Subsequent Characters: From the second character in an identifier, letters, digits, or underscores can be used.
- Case Sensitivity: Yes, Identifiers are case-sensitive.
- Length: Both languages allow identifiers of arbitrary length (though there may be practical limits).
- Reserved Words: Identifiers cannot be keywords or reserved words in the language.
- Unicode Characters: Both Java and Python support Unicode characters in identifiers, meaning you can use letters from various languages (e.g., ñ, 你好, etc.).

b) DIFFERENCES BETWEEN JAVA AND PYTHON

- Starting Characters:
  - Java: Allows identifiers to start with letters (a-z, A-Z), underscores (_), or dollar signs ($).
  - Python: Identifiers can only start with letters (a-z, A-Z) or underscores (_). Dollar sign ($) is not allowed in Python identifiers.
- Use of the Dollar Sign ($):
  - Java: The dollar sign ($) can be used in identifiers naming.
  - Python: The dollar sign ($) can't be used in identifiers.
- Special Meaning of Underscore (_)
  - Java: The underscore _ has no special meaning beyond being a valid character in identifiers.
  - Python: The underscore _ has special meaning:
    - Single leading underscore (_var): A convention for weak internal use (not enforced by the interpreter).
    - Double leading underscore (__var): Used to trigger name mangling (used for private variables).
    - Double leading and trailing underscores (init): Reserved for special methods in Python (e.g., constructors, operator overloads).
- Static vs. Dynamic Typing:
  - Java: A statically typed language, i.e., each identifier should be accompanied with the data type.
  - Python: A dynamically typed language, so identifiers can be used with mentioning its data type.
- Type-less Variable Names:
  - Java: Identifiers are linked to specific types and need to follow strict type conventions.
  - Python: Identifiers are not type-bound, and the language allows a more flexible approach for variable names.

c) SUMMARY TABLE: JAVA VS. PYTHON IDENTIFIERS

Table 3. Summary Table (Java Vs Python)

Rule	Java	Python
Starting Character	Letters (a-z, A-Z), _, $	Letters (a-z, A-Z), _
Allowed Characters	Letters (a-z, A-Z), digits (0-9), _, $	Letters (a-z, A-Z), digits (0-9), _
Dollar Sign ($)	Allowed	Not allowed
Special Meaning of '_'	No special meaning	_var (weak internal use), __var (name mangling), __init__ (special methods)
Case Sensitivity	Case-sensitive	Case-sensitive
Length of Identifiers	No strict limit	No strict limit
Unicode Characters	Allowed	Allowed
Static vs Dynamic Typing	Statically typed (types are defined at compile-time)	Dynamically typed (types are inferred at runtime)
Reserved Keywords	Cannot use reserved words	Cannot use reserved words

d) FINITE AUTOMATA REPRESENTATION

Definition: A mathematical model of computation used to depict and examine the behavior of systems that can exist in a finite number of states is called a finite automaton (plural: automata). Based on the rules established by the automaton, it evaluates an input string of symbols (characters) and decides whether the string is approved or refused.

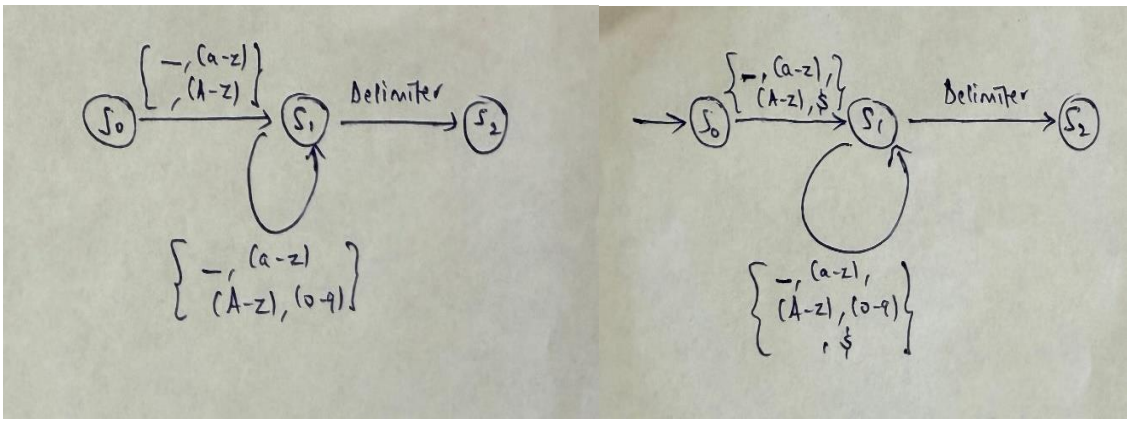

Fig 1. Finite Automata for Identifiers in Python and Java (Scanner Action)

States:

- Start State (S0): Where the automaton begins.
- Valid Start Character (S1): For characters like a-z, A-Z, _ and $ (in Java).
- Valid Continuation Character (S2): For characters like a-z, A-Z, 0-9, _, $(in Java).
- End State (S3): A valid identifier that has reached its end.

Alphabet: Characters a-z, A-Z, 0-9, _, $(in Java).

Dead State: For both Java and Python, if one encounter a character that is not allowed in an identifier, you transition to a dead state where no further valid transitions exist.

Token Classification after Identification of Token Category:

- Keyword Matching:
  - After a token is scanned, it is first checked against the keyword table (a predefined list of reserved words for the language).
  - If the token matches any entry in the keyword table, it is classified as a keyword.
- Identifier Classification:
  - If the token does not match any entry in the keyword table, the scanner then verifies whether it satisfies the identifier rules.
  - If the token satisfies the identifier rules, it is classified as an identifier.
- Invalid Tokens:
  - If the token neither matches a keyword nor satisfies the identifier rules, it is classified as an invalid token (error) and handled accordingly.

e) PROGRAM EXAMPLE

Python Code/ Program:

```
class Person:
    def __init__(self, name, age):
        self.name = name  # 'name' is an identifier, 'self' is a special identifier
        self.age = age    # 'age' is an identifier
    def greet(self):
        if self.name:
            return f"Hello, {self.name}"
        else:
            return "Hello, World!"
# Creating an instance of the class
person = Person("Alice", 30)
print(person.greet())
```

Identifiers: Person, __init__, self, name, age, greet, person

Java Code/ Program:

```
public class Person {
    private String name; // 'name' is an identifier
    private int age;     // 'age' is an identifier

    public Person(String name, int age) { // 'name' and 'age' are identifiers
        this.name = name; // 'this' is a special keyword
        this.age = age;
```

```
    }


    public String greet() {
        if (this.name != null) {
            // 'if' and 'return' are keywords
            return "Hello, " + this.name;
        } else {
            return "Hello, World!";
        }
    }

    public static void main(String[] args) {
        Person person = new Person("Alice", 30); // 'Person' and 'person' are identifiers
        System.out.println(person.greet());
    }
}
```

Identifiers: Person, name, age, this, greet, main, args, person, System, out, println

C) LITERALS

Definition:
Literals are constant values assigned to variables or used directly in a program. They represent fixed data that doesn't change during execution and are categorized based on the type of data they represent.

a) PYTHON LITERALS

Python supports several types of literals for representing data.

Key Categories of Python Literals:

- **String Literals**: Enclosed within single (') or double (") quotes. Examples: "Hello", 'Python'.
- **Numeric Literals**: Represent numbers, including integers, floating-point numbers, and complex numbers. Examples: 10, 3.14, 1+2j.
- **Boolean Literals**: Represent truth values: True or False.
- **Special Literals**: Used to represent the absence of a value. Example: None.
- **Collection Literals**: Represent collections like lists, tuples, sets, and dictionaries. Examples: [1, 2, 3], (4, 5), {6, 7}, {'key': 'value'}.

Table 4. Representation of Python Literals

Category	Example	Token Representation
String Literals	"Python"	(string, "Python")
Numeric Literals	42	(integer, 42)
	3.14	(float, 3.14)
	1+2j	(complex, 1+2j)
Boolean Literals	True	(boolean, True)
Special Literals	None	(special, None)
List Literals	[1,2,3]	(list, [1, 2, 3])
Tuple Literals	(4,5)	(tuple, (4, 5))
Set Literals	{6,7}	(set, {6, 7})
Dictionary Literals	{'key': 'value'}	(dict, {'key': 'value'})

Python Code/Program:

```
# Examples of literals in Python

name = "Alice"        # String Literal
age = 25              # Integer Literal
pi = 3.14             # Float Literal
is_student = False    # Boolean Literal
nothing = None        # Special Literal
fruits = ["apple", "banana", "cherry"]  # List Literal
```

Literals Used: "Alice", 25, 3.14, False, None, ["apple", "banana", "cherry"].

b) JAVA LITERALS

Java also supports a variety of literals to represent fixed values in the program.

Key Categories of Java Literals:

- **String Literals**: Enclosed within double quotes (") to represent a sequence of characters. Example: "Hello, Java!".
- **Numeric Literals**: Represent numbers, including integers, floating-point numbers, and hexadecimal or binary numbers. Examples: 42, 3.14, 0xFF (hex), 0b101 (binary).
- **Character Literals**: Enclosed within single quotes (') to represent a single character. Example: 'A'.
- **Boolean Literals**: Represent truth values: true or false.
- **Null Literal**: Represents the absence of a value or a reference. Example: null.

Table 5. Representation of Java Literals

Category	Example	Token Representation
String Literals	"Java"	(string, "Java")
Numeric Literals	42	(integer, 42)
	3.14	(float, 3.14)
	0xFF	(hexadecimal, 0xFF)
	0b101	(binary, 0b101)
Character Literals	'A'	(char, 'A')
Boolean Literals	true	(boolean, true)
Null Literal	null	(null, null)

Java Code/Program:

```
public class LiteralsExample {
    public static void main(String[] args) {
        String name = "Alice";    // String Literal
        int age = 25;             // Integer Literal
        double pi = 3.14;         // Float Literal
        boolean isStudent = false; // Boolean Literal
        char grade = 'A';         // Character Literal
        String nothing = null;    // Null Literal

        System.out.println(name + " is " + age + " years old.");
    }
}
```

Literals Used: "Alice", 25, 3.14, false, 'A', null.

D) OPERATORS

Definition:
Operators are special symbols or keywords used in programming to perform specific operations on variables and values. These operations include arithmetic, comparison, logical operations, and more.

a) PYTHON OPERATORS

Python supports a variety of operators classified into different categories based on their functionality.

Key Categories of Python Operators:

- **Arithmetic Operators:** +, -, *, /, %, //, **
- **Comparison Operators**: ==, !=, >, <, >=, <=
- **Logical Operators**: and, or, not
- **Bitwise Operators**: &, |, ^, ~, <<, >>
- **Assignment Operators**: =, +=, -=, *=, /=, %=, //=, **=, &=, |=, ^=, <<=, >>=
- **Membership Operators**: in, not in
- **Identity Operators**: is, is not

Table 6. Token Representation of Python Operators

Category	Operator Identified	Token Representation
Arithmetic Operators	+	(operator, “+”)
	-	(operator, “-“)
	*	(operator, "*")
	/	(operator, "/")
	%	(operator, "%")
Comparison Operators	==	(operator, "==")
	!=	(operator, “!=”)
	>	(operator, “>”)
Logical Operators	and	(operator, “and”)
	or	(operator, "or")
Membership Operators	in	(operator, "in")
	not in	(operator, “not in”)
Identity Operators	is	(operator, “is”)
	is not	(operator, "is not")

Python Code/Program Example:

```
class Calculator:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def calculate(self):
        result_sum = self.a + self.b  # Arithmetic Operator
        is_equal = self.a == self.b  # Comparison Operator
```

```
        logical_check = (self.a > 0) and (self.b > 0)  # Logical Operator

        return f"Sum: {result_sum}, Equal: {is_equal}, Logical Check: {logical_check}"

calc = Calculator(5, 10)
print(calc.calculate())
```

Operators Used: +, ==, >, and, =

b) JAVA OPERATORS

Java provides a rich set of operators, classified into various types for different purposes.

Key Categories of Java Operators:

- **Arithmetic Operators**: +, -, *, /, %, ++, --
- **Comparison/Relational Operators**: ==, !=, >, <, >=, <=
- **Logical Operators**: &&, ||, !
- **Bitwise Operators**: &, |, ^, ~, <<, >>, >>>
- **Assignment Operators**: =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=
- **Ternary Operator**: ? :
- **Instanceof Operator**: instanceof

Table 7. Token Representation of Java Operators

Category	Operator Identified	Token Representation
Arithmetic Operators	+	(operator, "+")
	-	(operator, "-")
	*	(operator, "*")
	/	(operator, "/")
	%	(operator, "%")
Logical Operators	&&	(operator, "&&")
	||	(operator, "||")
Comparison Operators	==	(operator, "==")
	!=	(operator, "!=")
Ternary Operator	?:	(operator, "?:")
Assignment Operators	=	(operator, "=")
	+=	(operator, "+=")

Java Code/Program Example:

```
public class Calculator {
    private int a;
    private int b;

    public Calculator(int a, int b) {
        this.a = a;
        this.b = b;
```

```
    }

    public String calculate() {
        int resultSum = a + b; // Arithmetic Operator
        boolean isEqual = (a == b); // Comparison Operator
        boolean logicalCheck = (a > 0) && (b > 0); // Logical Operator

        return "Sum: " + resultSum + ", Equal: " + isEqual + ", Logical Check: " + logicalCheck;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator(5, 10);
        System.out.println(calc.calculate());
    }
}
```

Operators Used: +, ==, >, &&, =

E) SEPERATORS

Definition: Separators (or delimiters) are symbols which structure and organize the code, such as parentheses (), brackets [], braces {}, commas, semicolons; and periods.

Token Labeling:

- **Python**: Usually lumps separators under an OP (operator) token category.
- **Java:** The compiler (javac) has distinct token names provided for each separator but doesn't expose them in a public token stream.

Table 8. Comparison Table: Separators

Aspect	Python	Java
Common Symbols	(), [], {}, ,, ., :, ;	(), [], {}, ,, ., ;
Token Label	Typically, OP tokens	Distinct token types (e.g., LPAREN, SEMICOLON)
Block Structure	Uses : + indentation	Uses {} for blocks
Semicolon Usage	Optional; rarely used	Mandatory to terminate statements
Counting Tokens	Each separator counts as one	Each separator is a separate token internally (requires tools to count)

a) PYTHON SEPARATORS:

Some of the standard python separators are listed as below:

() [] {}
, : ! . ; @ =
-> += -= *= /= //= %=
@= &= |= ^= >>= <<= **=

Python Example: Separators

```
# Function to calculate the square of numbers in a list
def calculate_squares(numbers):  # () used for function definition
    squares = []  # [] used for list initialization
    for number in numbers:  # : used for block structure
        squares.append(number ** 2)  # . used for method call
    return squares  # Indentation separates blocks

# Call the function
result = calculate_squares([1, 2, 3, 4])  # [] for list, () for function call
print(result)  # () for print function
```

Separators Used: (), [], ., :, ,

b) JAVA SEPARATORS:

Some of the standard java separators are listed as below:

{} () [] ,, . , ;

Java Example: Separators

```
import java.util.ArrayList;  // . for package reference
public class SeparatorExample {  // {} for class block
    public static void main(String[] args) {  // () for method signature, [] for array
        ArrayList<Integer> numbers = new ArrayList<>();  // <> for generics
        numbers.add(1);  // . for method call
        numbers.add(2);
        numbers.add(3);
        for (int number : numbers) {  // () for loop structure, : in enhanced for-loop
            System.out.println(number * number);  // ; to terminate statement
        }
    }
}
```

Separators Used: {}, (), [], ., :, ;

F) WHITESPACES

Definition:

Whitespace includes spaces, tabs, and newline characters. This are basically used to separates tokens from each other. And helps in indentation.

a) PYTHON TOKENIZATION:

- **Significance:** Indentation is syntactically significant.
- **Token Types:**
  - INDENT and DEDENT: Represent changes in indentation levels.
  - NEWLINE: Marks the end of a statement.
- **Usage:**
  - Indentation defines the scope of loops, conditionals, functions, etc. or we can say it used to define the block.
  - Spaces and tabs within lines separate tokens but do not generate separate tokens themselves.

Python Example: Whitespace

```
def greet_user(name):  # Function definition (no extra whitespace allowed before 'def')
    if name:  # Indentation to define block
        print(f"Hello, {name}")  # Further indentation
    else:  # Indentation aligns with the 'if'
        print("Hello, World!")  # Inside the 'else' block
greet_user("Alice")  # Proper whitespace for function call
```

Key Points:

- **Indentation: Mandatory for defining blocks.**
- **Extra Whitespace: Not allowed before the start of a statement.**
- **Improper indentation causes an "IndentationError".**

b) JAVA TOKENIZATION :

- **Significance**: Whitespace is insignificant for defining code structure.
- **Token Handling:**
  - Spaces, tabs, and newlines are ignored during lexical analysis.
  - Code structure is defined by braces {}, parentheses (), and semicolons(;)
- **Usage:**
  - Indentation is purely for readability; it does not affect compilation.

Java Example: Whitespace

```
public class WhitespaceExample {  // Class block begins
    public static void main(String[] args) {  // Method block begins
        String name = "Alice";  // Variable declaration (spaces optional)
```

```
        if (name != null) {  // No mandatory indentation, but good practice
            System.out.println("Hello, " + name);  // Statement within block
        } else {  // Curly braces define blocks, whitespace is optional
            System.out.println("Hello, World!");
        }
    }
}
```

Key Points:

- **Whitespace is optional for code structure but enhances readability.**
- **Curly Braces ({}): Define blocks instead of relying on indentation.**
- **Best Practice: Use proper indentation (4 spaces or a tab) for readability.**

Table 9. Comparison Table: Whitespace

Aspect	Python	Java (javac)
Indentation	Significant; generates INDENT/DEDENT tokens	Ignored; no tokens generated
Newlines	Generate NEWLINE tokens	Ignored; semicolons terminate statements
Spaces/Tabs	Separate tokens but do not generate tokens	Ignored; serve only to separate tokens
Impact on Structure	Defines code blocks and scopes	Braces define blocks; no impact from whitespace
Counting Tokens	Count INDENT, DEDENT, and NEWLINE tokens	javac does not expose whitespace tokens

G) COMMENTS

Definition:
Comments are non-executable text within code used for documentation and explanations.

a) PYTHON COMMENTS

- **Types of Comments:**
  - Single-Line: Start with # and continue to the end of the line.
  - Docstrings: Triple-quoted strings ("""...""") used for documentation; treated as STRING tokens.
- **Token Types:**
  COMMENT: Represents single-line comments starting with #.
- **Usage:**
  - Single-line comments provide inline explanations.
  - Docstrings serve as documentation for modules, classes, and functions, accessible at runtime via . doc .

Python Comments Example:

```
# Single-Line Comment
# This is a single-line comment in Python
name = "Alice"  # Inline comment: Assigning a value to the variable
print(f"Hello, {name}")  # Display the greeting
"""
Multi-Line Comment
This is a multi-line comment in Python.
It is often used for documentation or
providing detailed explanations.
"""
def greet():
    print("Welcome to Python!")
'''
Another Multi-Line Comment
This can also be used, though it's less common.
'''
greet()
```

b) JAVA COMMENTS

- **Types of Comments:**
  - Single-Line: Start with // and continue to the end of the line.
  - Block: Enclosed between /* and */, can span multiple lines.
  - Javadoc: Enclosed between /** and */, used for documentation generation.
- **Token Handling:**

- The javac compiler discards all comments during lexical analysis.
- Comments do not appear as tokens in the final Abstract Syntax Tree (AST).

- **Usage:**
  - Single-line comments provide inline explanations.
  - Block comments are used for multi-line explanations.
  - Javadoc comments are processed by the javadoc tool to generate documentation but ignored by javac.

Java Comments Example:

```
// Single-Line Comment
// This is a single-line comment in Java
String name = "Alice";  // Inline comment: Assigning a value to the variable
System.out.println("Hello, " + name);  // Display the greeting
/*
Multi-Line Comment
This is a multi-line comment in Java.
It can span multiple lines and is commonly used
for detailed explanations or block comments.
*/
public class CommentsExample {
    public static void main(String[] args) {
        System.out.println("Welcome to Java!");
    }
}
public class JavadocExample {
    /**
     * Main method to run the program.
     * @param args Command-line arguments
     */
    public static void main(String[] args) {
        System.out.println("Using Javadoc for documentation!");
    }
}
```

Table 10. Comparison Table: Comments

Aspect	Python	Java
Single-Line Syntax	# ... → COMMENT token	// ... → Discarded by javac
Block Comments	No dedicated syntax; use multiple # lines	/* ... */ → Discarded by javac
Docstrings/Javadoc	""" ... """ treated as STRING tokens	/** ... */ used by javadoc tool; discarded by javac
Token Emission	Each # ... line is a COMMENT token	No comment tokens appear in the final parse
Counting Comments	Each # line counts as one COMMENT token	javac does not count comments; external tools needed

CONCLUSION

Python:

- Lower token count allows for concise, readable, and maintainable code.
- Minimalistic syntax with indentation-based blocks and optional semicolons makes it user-friendly.
- Well-suited for rapid development, smaller projects, scripting, and data science tasks.

Java:

- Higher token count emphasizes explicitness and type safety.
- Verbose syntax with mandatory braces and semicolons enhances structure and robustness.
- Ideal for large-scale, enterprise-level applications requiring clarity and scalability.

Comparison Summary:

- Python excels in simplicity and faster development.
- Java provides better structure and safety for complex, performance-critical projects.
- The choice depends on project needs, balancing efficiency, readability, and scalability.