PNW
PURDUE UNIVERSITY NORTHWEST

PURDUE UNIVERSITY
NORTHWEST

Token Analysis : Python vs. Java

Group 1

Vaishak Balachandra - Keywords and Identifiers
Anvesh Pavuluri - Literals and Operators
Nadeem Jaffer Mohammed - Separators/Delimiters, Comments, and Whitespace

CS51530 – Programming Languages, Interpreters and Compilers

Assignment - 1

Table of Contents

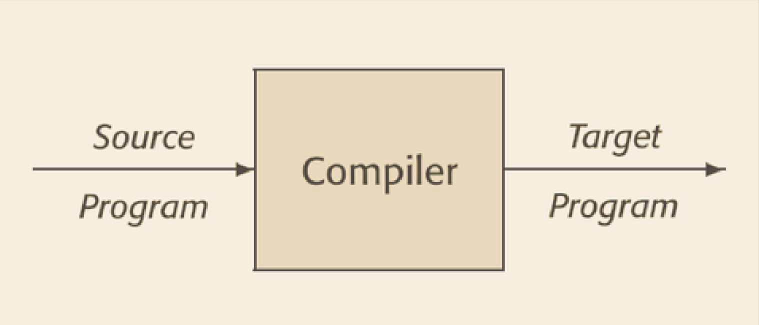
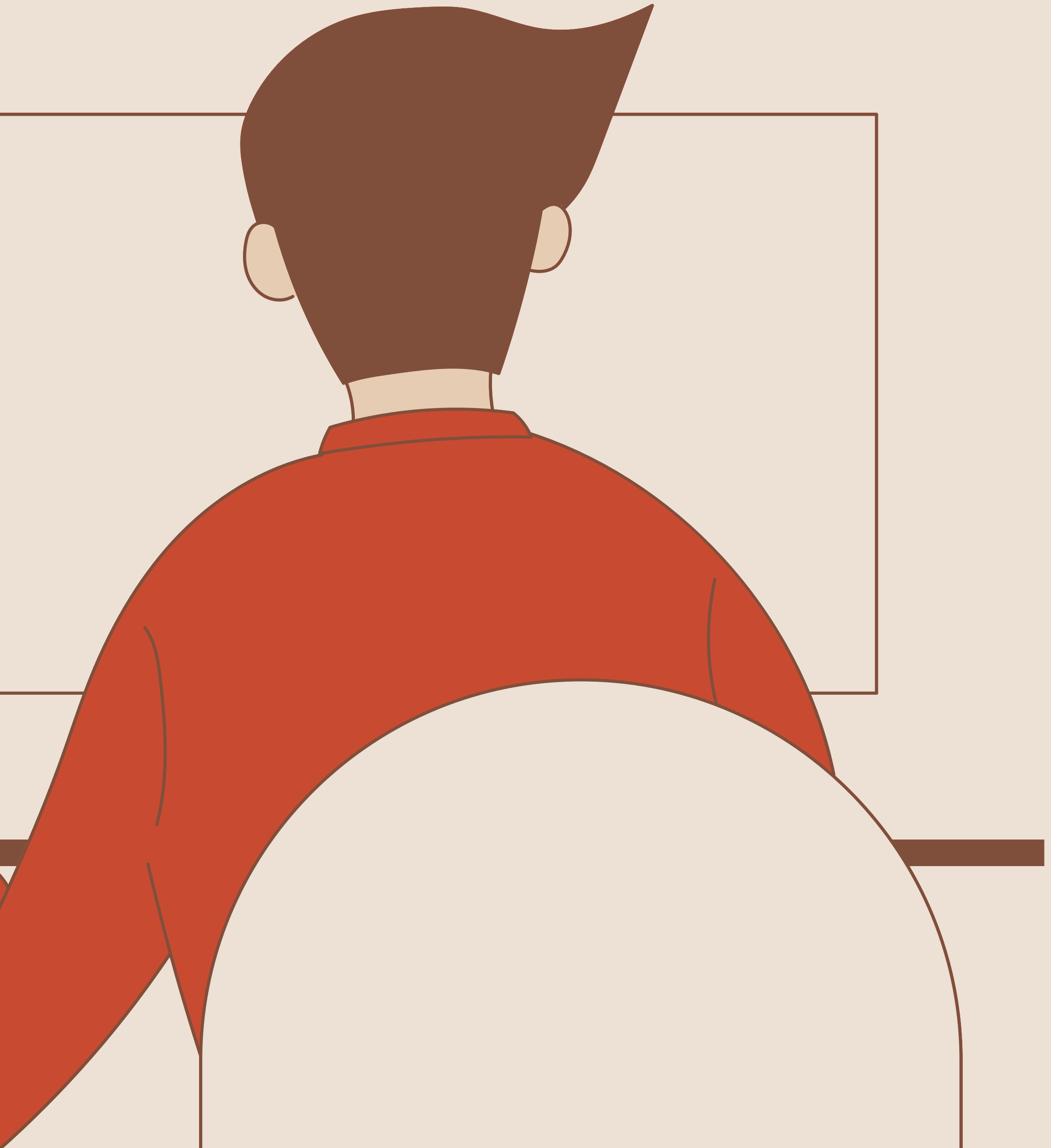


FIG 1. COMPILER ACTION



- **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

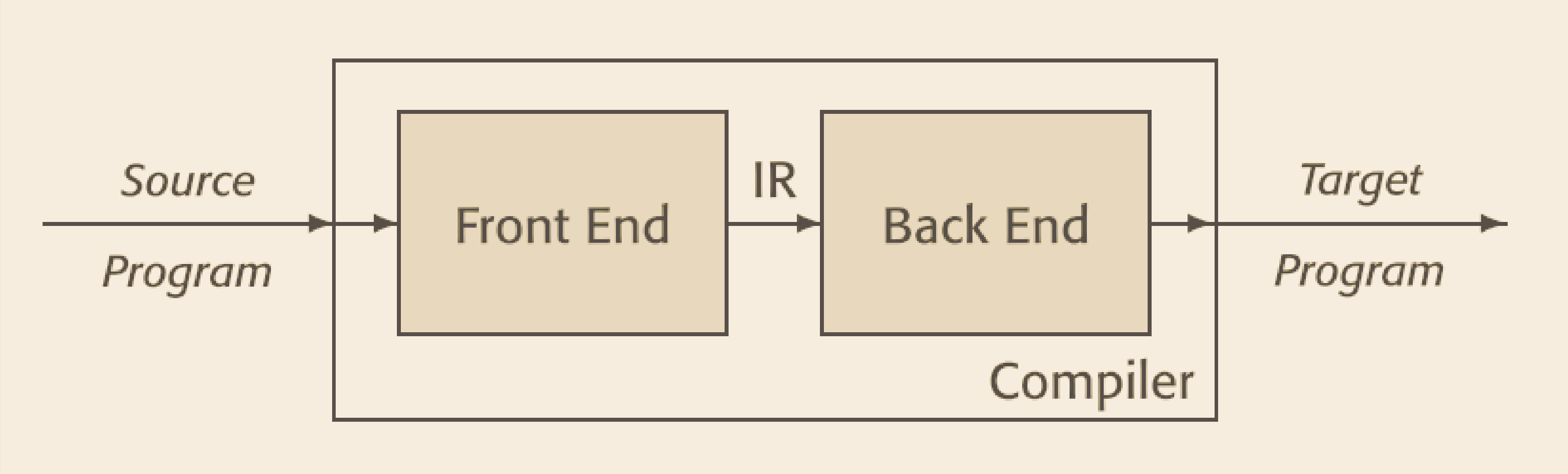
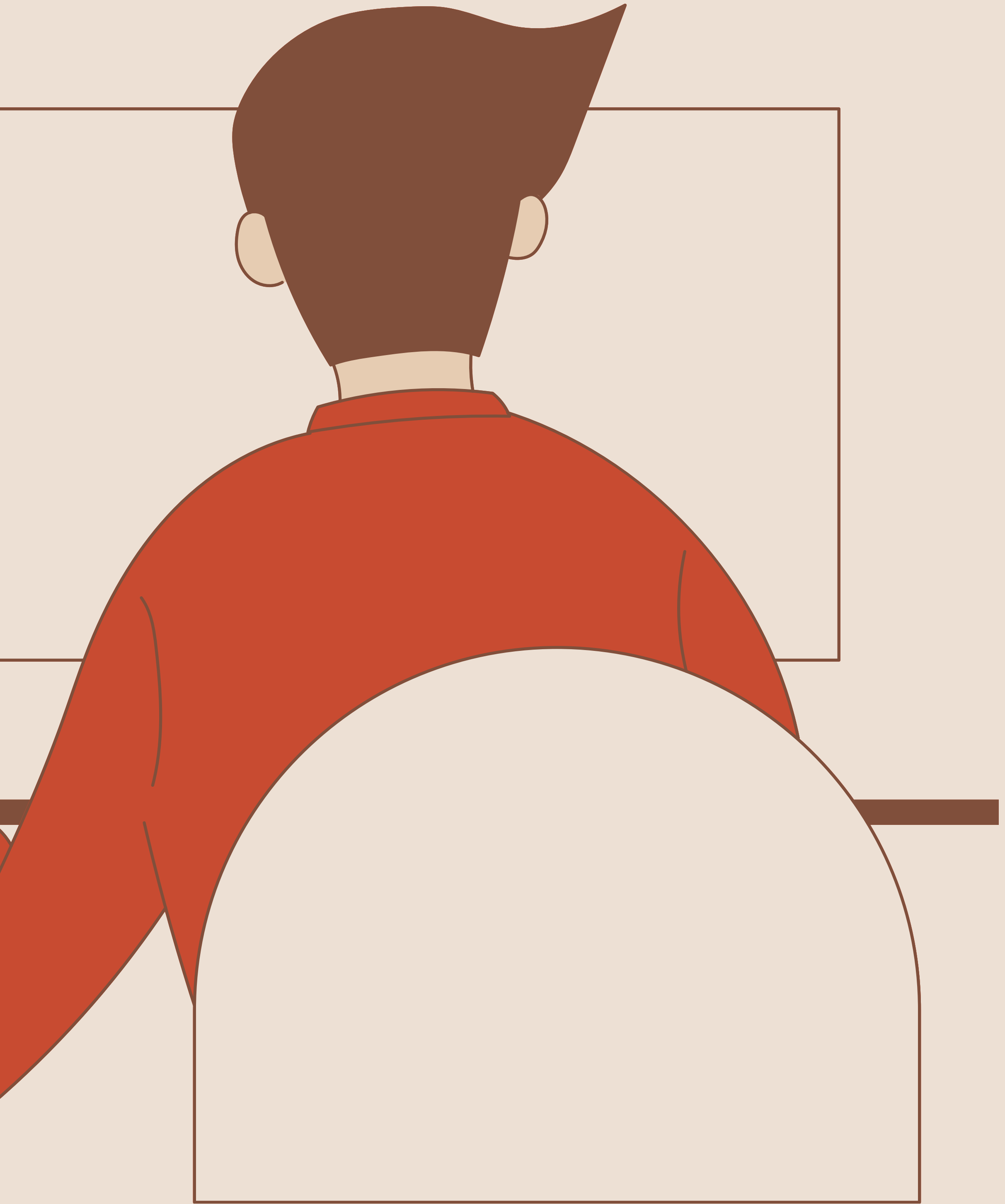


FIG 2. TWO PHASE COMPILER BLOCK DIAGRAM



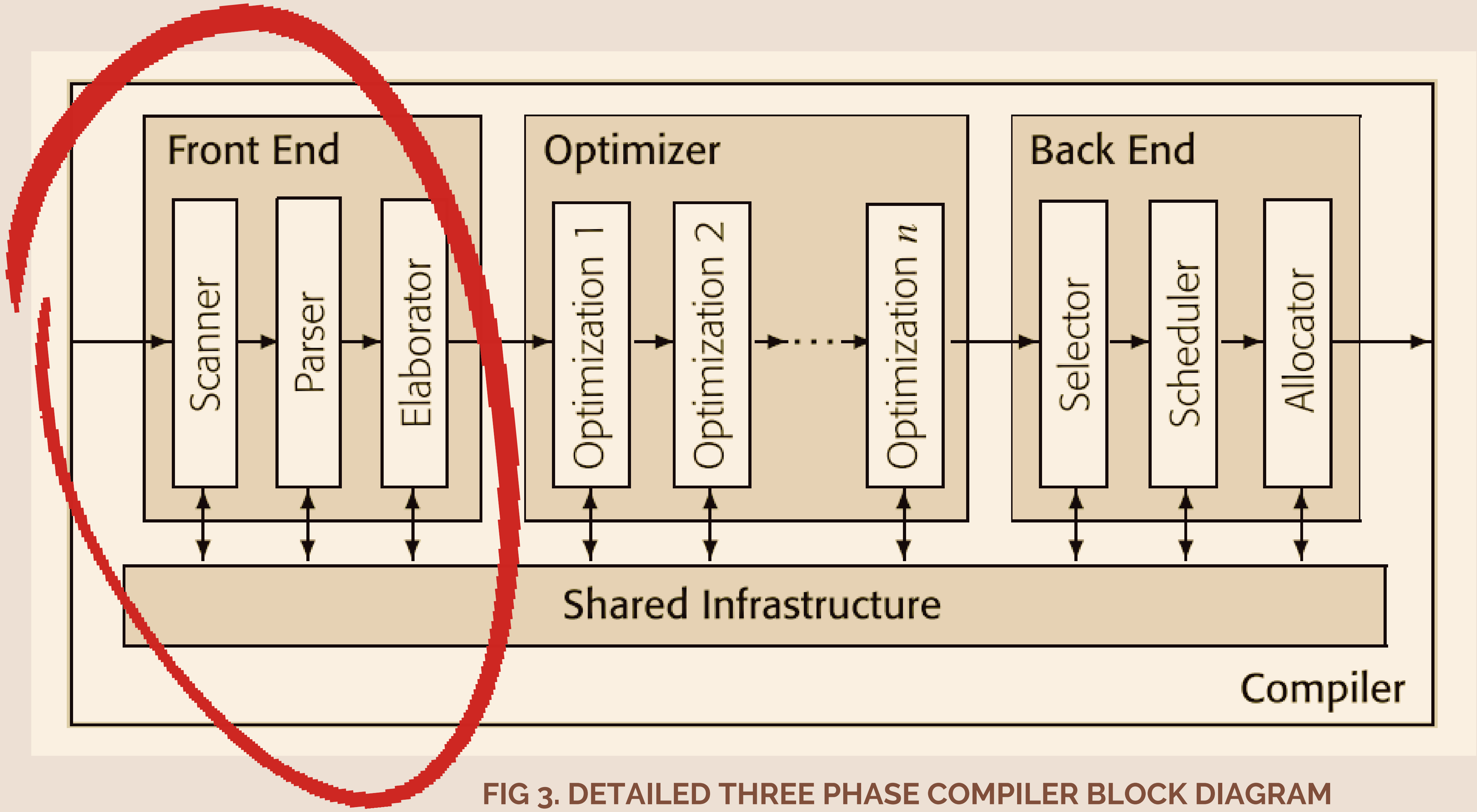
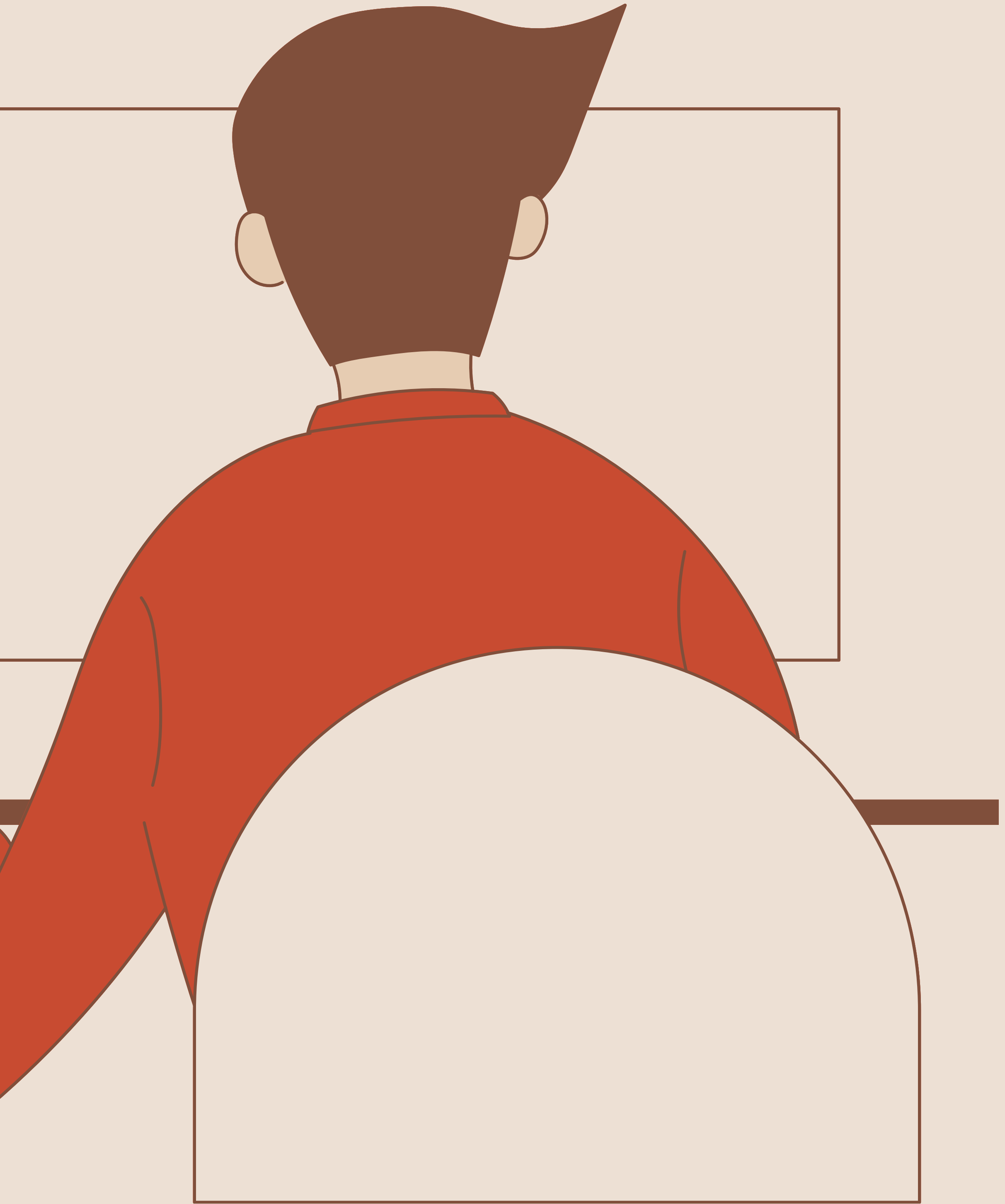


FIG 3. DETAILED THREE PHASE COMPILER BLOCK DIAGRAM

- **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).



The Scanner Phase (Lexical Analysis)

Role of the Scanner:

- Reads the source code character by character.
- Groups characters into meaningful units called tokens.
- Eliminates whitespace and comments.

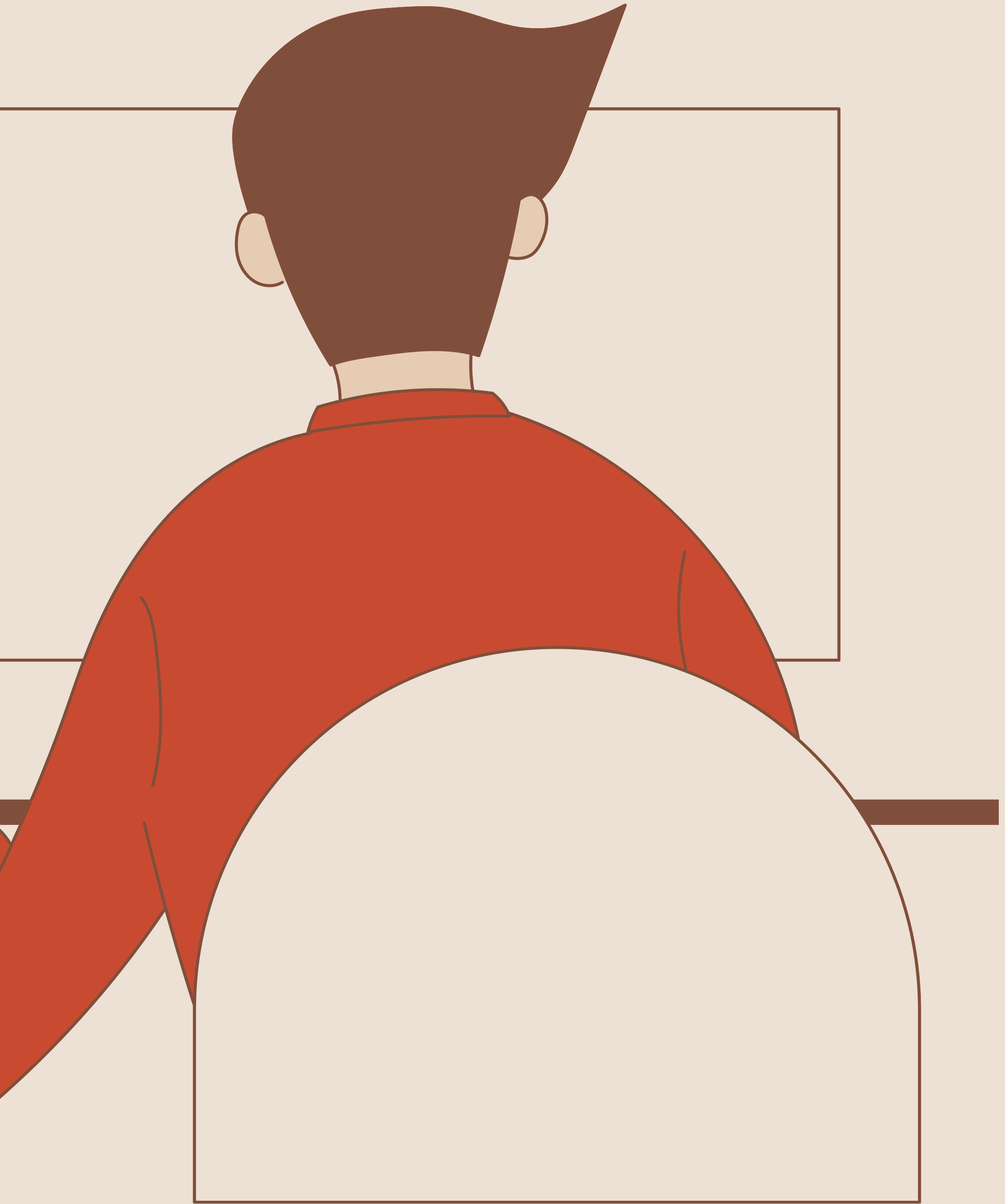
Output: A sequence of tokens passed to the parser.

Example:

Python Source Code:

```
if (x > 0):
   print(x)
```

Tokens: if, x, >, 0, :, print, (, x,).



- **Tokens – Definition and Categories**
  - Definition: Tokens are the smallest units of a program, generated during lexical analysis.
  - Categories of Tokens:
    a. Keywords (Reserved words like if, else).
    b. Identifiers (Variable or function names like x, y).
    c. Literals (Fixed values like 42, "Hello").
    d. Operators (+, -, =).
    e. Separators/Delimiters ({, }, ;).
    f. Comments (Ignored by the compiler).
    g. Whitespace (Used for formatting).

Token Representation:
(token_Category, token)

Keywords & Identifiers

• KEYWORDS

Definition: Reserved words with predefined meanings that cannot be used as identifiers.

Role: Define the syntax and control flow of a language.

Python:

Total: 35 keywords.

Examples: def, class, lambda, async...

Dynamic, focuses on readability.

TABLE 1. KEYWORDS IN PYTHON

async	continue	while	def	as
await	elif	del	lambda	and
False	else	assert	yield	in
None	for	except	from	is
True	if	finally	global	not
with	return	raise	import	or
break	try	class	nonlocal	pass

Java:

Total: 51 keywords.

Examples: public, static, synchronized, final...

Statically typed, more verbose.

TABLE 2. KEYWORDS IN JAVA

private	extends	new	case	for	while	float	void	try	native
protected	implements	package	continue	goto	boolean	int	catch	abstract	static
public	import	super	default	if	byte	long	finally	assert	strictfp
class	instanceof	this	do	return	char	null	throw	const	transient
enum	interface	break	else	switch	double	short	throws	final	Synchronized
volatile									

• IDENTIFIERS

Definition: Definition: User-defined names for variables, functions, and classes.

Rules: The rules to write an identifiers changes from one programming language to another.

Similarities Between Java and Python

- 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
- 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.).

• IDENTIFIERS

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 (_) and no Dollar sign ($) 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 (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.

- **IDENTIFIERS**

TABLE 3. IDENTIFIERS 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

• FINITE AUTOMATA - REPRESENTATION

States:

o Start State (S0): Where the automaton begins.

o Valid Start Character (S1): For characters like a-z, A-Z, _ and $ (in Java).

o Valid Continuation Character (S2): For characters like a-z, A-Z, 0-9, _, $(in Java).

o 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.

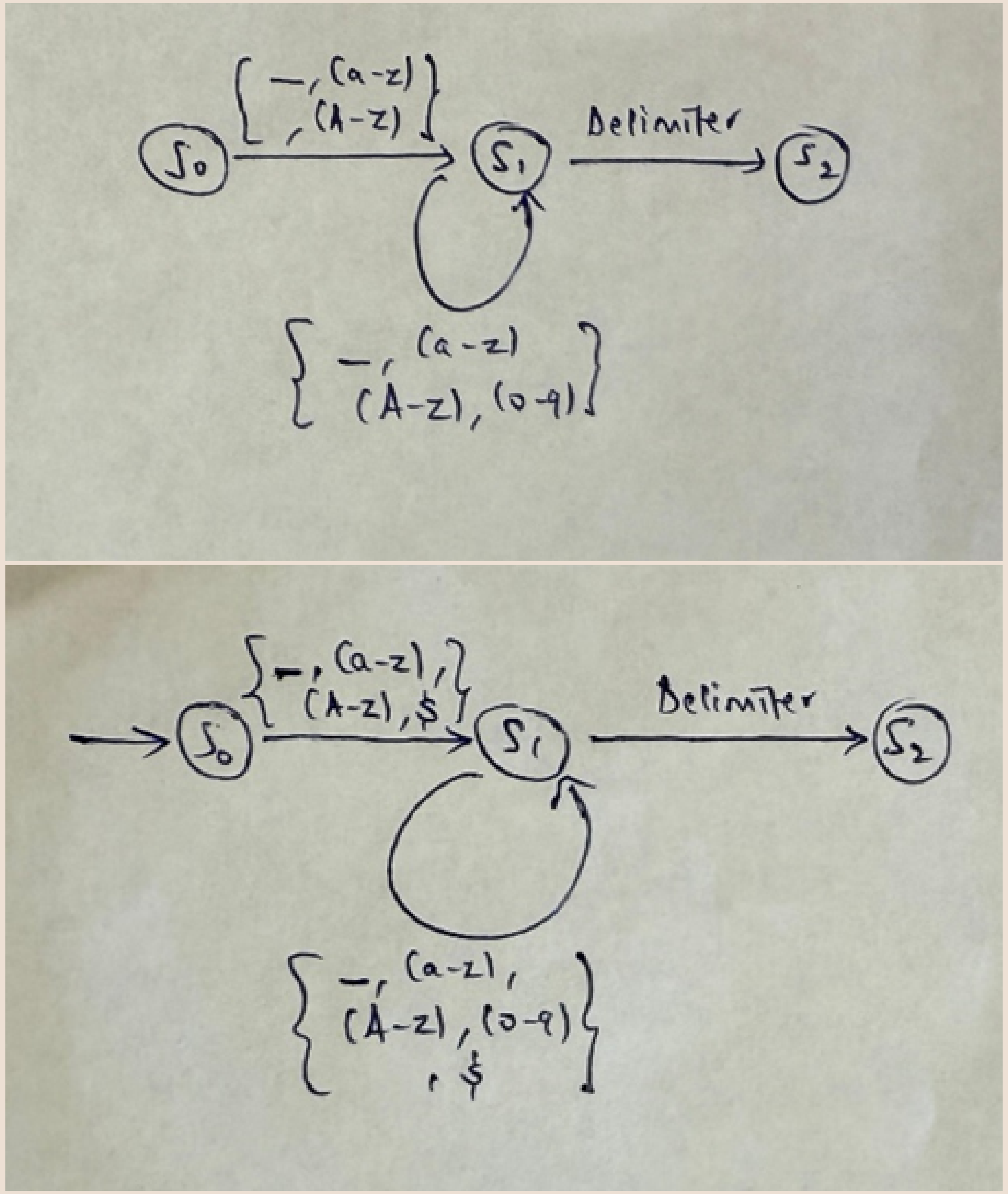


FIG 1. FINITE AUTOMATA FOR IDENTIFIERS IN PYTHON AND JAVA

- **Token Classification after Identification of Token Category**

o Keyword Matching:

After a token is scanned, it is first checked against the keyword table.

If the token matches any entry in the keyword table, it is classified as a keyword.

o 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.

o 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.

• PROGRAMMING EXAMPLE

PYTHON:

```
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

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

NOTE:

- **__init__** is a special identifier used for constructors.
- **print** is a function name defined in any of the python package.
- **f** in f"Hello, {self.name}" is a syntax feature for **f-strings** and doesn't belong to the category of keywords or identifiers. It's used to signify a formatted string literal in Python.

• PROGRAMMING EXAMPLE

JAVA:

```
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

Identifiers:

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

NOTE:

- Here, **{"System”, “out”, “println”}** are the either name of the methods or the classes, Hence considered as identifiers.

• Java vs Python: Readability, Ease of Use, Verbosity, and Token Efficiency

TABLE 4. COMPARISON OF PYTHON AND JAVA: WRT TO TOKEN COUNT

Feature	Python	Java
Keywords	class, def, if, else, return	public, class, String, private, int, if, else, return, static, void, new
Identifiers	Person, __init__, self, name, age, greet, person	Person, name, age, this, greet, main, args, person, System, out, println
Token Count	Keywords: 5 Identifiers: 7	Keywords: 10 Identifiers: 10

To Summarize:

Java: Less readable, more verbose, harder to write.

Python: More readable, easy to write, concise, but still verbose.

Token Efficiency Analysis:

- **PYTHON** has a total of **12 tokens**, which suggests that fewer elements (keywords and identifiers) are required to express the same functionality compared to Java. This results in more concise code, improving both readability and ease of use. The lower token count also typically translates to faster development and easier maintenance.

- **JAVA** has a total of **20 tokens**, which reflects its more verbose syntax. While this verbosity can provide benefits in terms of type safety and structure, it also increases cognitive load and development time. The higher token count generally results in more detailed and explicit code, but also reduces readability and increases complexity for smaller projects or quick tasks.

CS51530 – Programming Languages, Interpreters and Compilers

Literals & Operators

- ## LITERALS

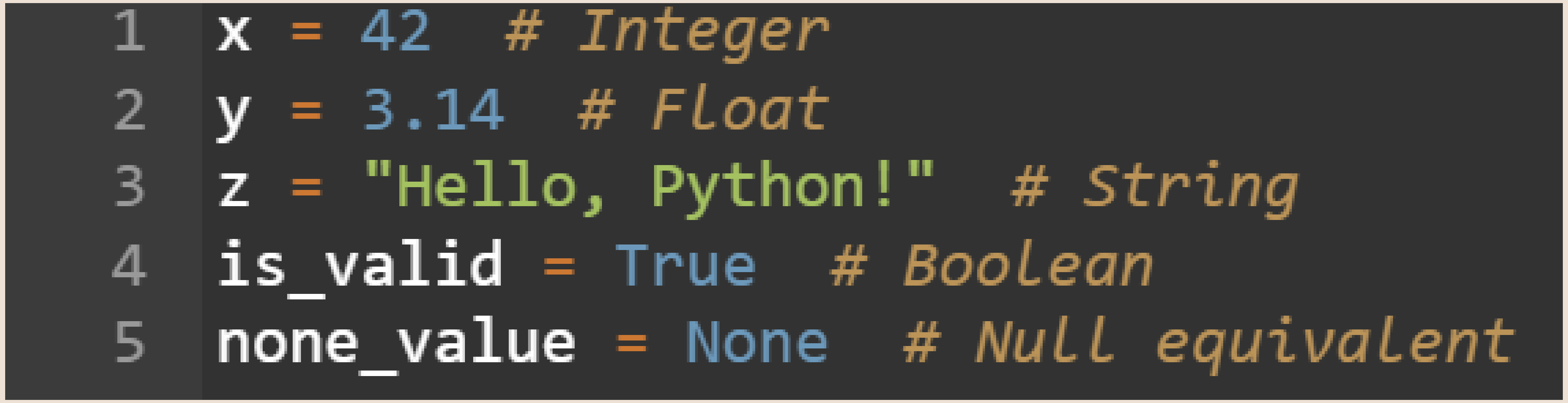
Definition: Fixed values used in the code, such as numbers, characters, or strings..

Role: Literals are treated as atomic tokens that don't require further breakdown, making them fundamental for the syntactic structure of the code..

COMPARISION TABLE:

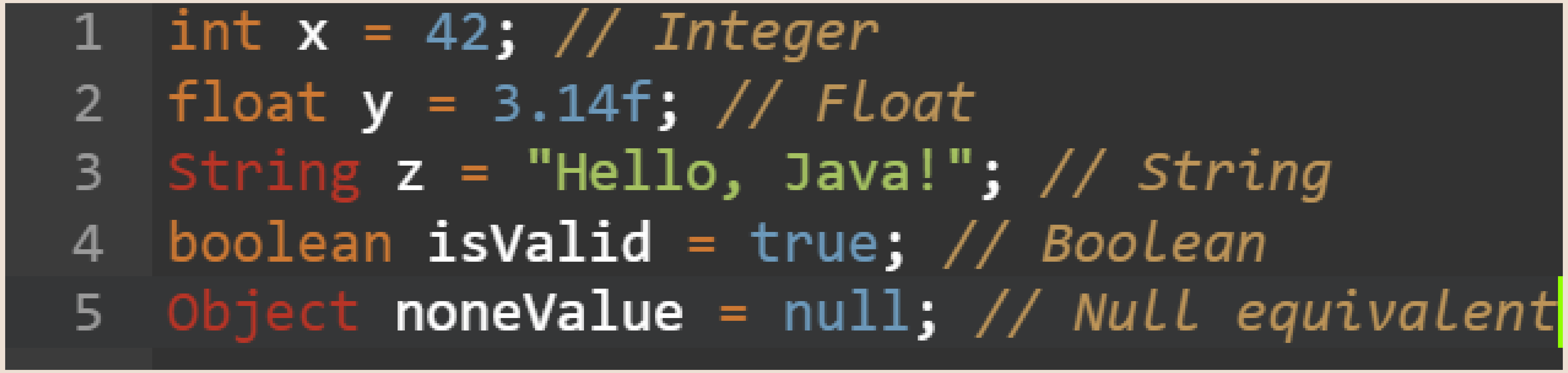
Aspect	Python	Java
Numeric Literals	Integers, floats, complex numbers	Integers, floats (double/float), long
String Literals	Enclosed in single (') or double (")	quotes Enclosed in double quotes (") only
Boolean Literals	True, False	true, false
Null	None	null

EXAMPLE (PYTHON):



```python
x = 42  # Integer
y = 3.14  # Float
z = "Hello, Python!"  # String
is_valid = True  # Boolean
none_value = None  # Null equivalent
```

EXAMPLE (JAVA):



```java
int x = 42; // Integer
float y = 3.14f; // Float
String z = "Hello, Java!"; // String
boolean isValid = true; // Boolean
Object noneValue = null; // Null equivalent
```

CS51530 – Programming Languages, Interpreters and Compilers

• OPERATORS

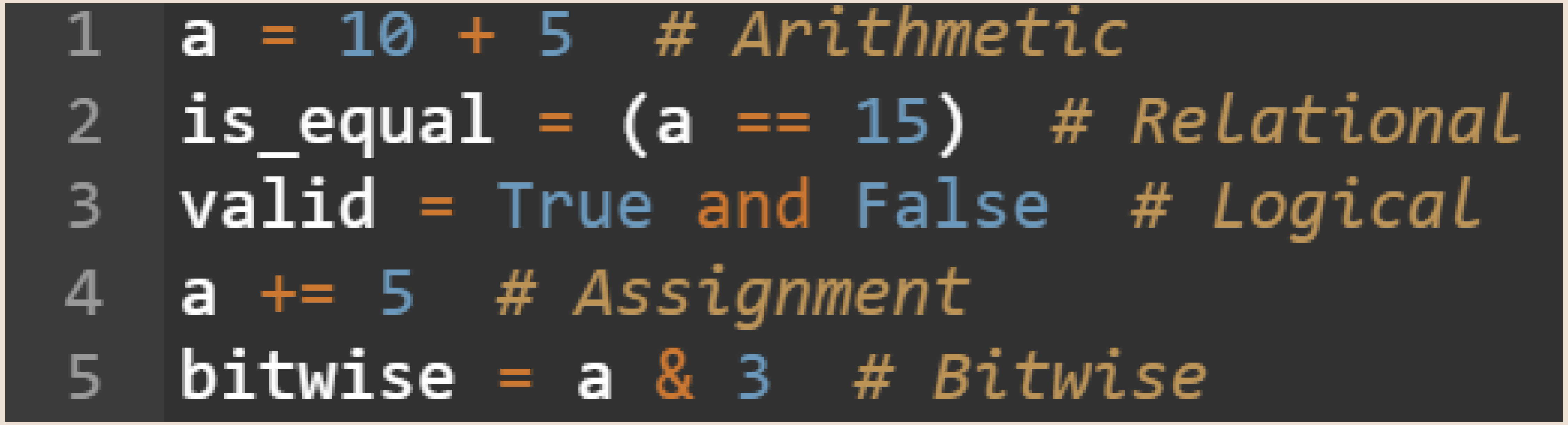
Definition: Symbols or keywords used to perform operations on variables and values.

Role: They are used to perform different operations on the operands

COMPARISION TABLE:

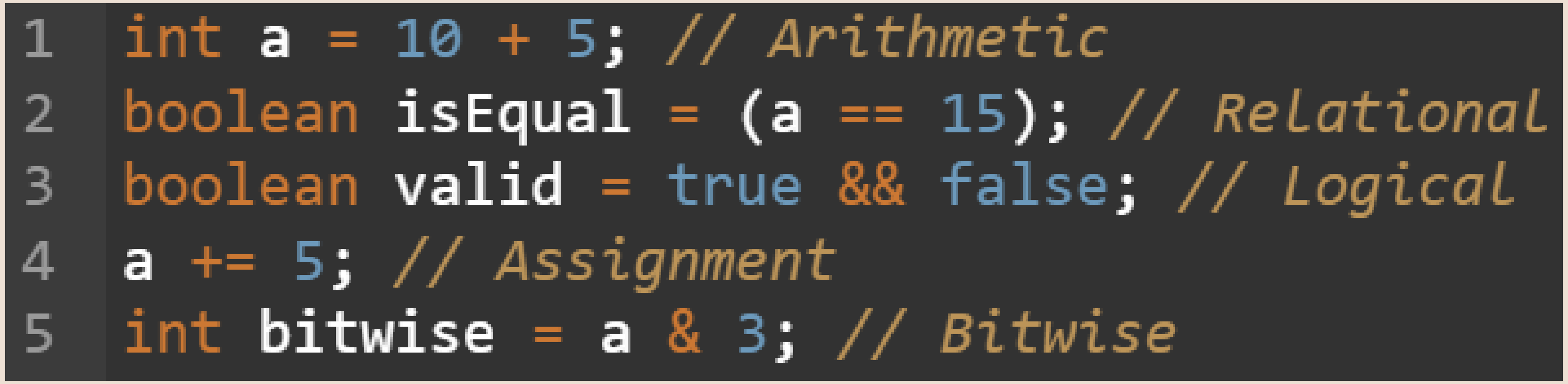
Category	Python	Java
Arithmetic Operators	+, -, *, /, %, **	+, -, *, /, %
Relational Operators	==, !=, <, >, <=, >=	==, !=, <, >, <=, >=
Logical Operators	and, or, not	&&, `
Assignment Operators	=, +=, -=, *=	=, +=, -=, *=
Bitwise Operators	&, `	, ^, ~, <<, >>`

EXAMPLE (PYTHON):



```python
a = 10 + 5   # Arithmetic
is_equal = (a == 15)  # Relational
valid = True and False  # Logical
a += 5  # Assignment
bitwise = a & 3  # Bitwise
```

EXAMPLE (JAVA):



```java
int a = 10 + 5; // Arithmetic
boolean isEqual = (a == 15); // Relational
boolean valid = true && false; // Logical
a += 5; // Assignment
int bitwise = a & 3; // Bitwise
```

Impact on Programming

- **LITERALS**
  - **PYTHON:** SIMPLER SYNTAX WITH SINGLE/DOUBLE QUOTES INTERCHANGEABLE FOR STRINGS. DYNAMIC TYPING MAKES NULL (NONE) HANDLING STRAIGHTFORWARD.
  - **JAVA:** STRONG TYPING PROVIDES BETTER COMPILE-TIME ERROR CHECKING. NULL REQUIRES EXPLICIT HANDLING.

:

- **OPERATORS**
  - **PYTHON:** MORE INTUITIVE FOR BEGINNERS (E.G., AND, OR INSTEAD OF &&, ||). SUPPORTS ADDITIONAL FEATURES LIKE EXPONENTIATION (**).
  - **JAVA:** STRICT TYPING ENFORCES ROBUST CODE BUT CAN FEEL VERBOSE (E.G., && VS. AND).

Separators, Whitespace, and Comments

SEPARATORS

Separators (also called delimiters or punctuation symbols) are special symbols used to:

- **Structure** the program (e.g., block boundaries, statement boundaries).
- **Separate** items in lists or parameters in function calls.
- **Divide** code elements (like identifiers, operators, keywords) so the compiler/interpreter can parse them correctly.

Python’s Separators

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

Java’s Separators

( ) { } [ ] ; , ... @ ::

CS51530 – Programming Languages, Interpreters and Compilers

• SEPARATORS

Block Definition:

Python: Uses a colon : plus indentation to define code blocks; braces {} are for data structures (dict, set).

Java: Uses braces {} to define code blocks (classes, methods, loops).

Semicolon Usage:

Python: Semicolons are optional; newlines typically end statements.

Java: Semicolons must terminate every statement.

Colon:

Python: The colon is crucial for control structures and function definitions.

Java: The colon is not used as a block opener; rarely appears (e.g., in "enhanced for" loops like for (Type var : array) but it's not for code blocks).

Token Labeling:

Python: Usually lumps separators under an OP (operator) token category.

Java: The compiler (javac) has distinct token names for each separator but doesn't expose them in a public token stream.

- **SEPARATORS**

TABLE : SEPARATORS SUMMARY TABLE (JAVA VS PYTHON)

Rule	Python	Java
Common Symbols	(,), [,], {, }, ,, ., :, ;	(,), [,], {, }, ,, ., ;
Block Structure	Uses : + indentation (not braces)	Uses curly braces { ... } for classes, methods, loops
Semicolon	Optional; newlines usually separate statements	Mandatory to end statements (e.g., int x = 0;)
Token Label	Typically labeled as OP tokens in the tokenize module	Distinct tokens inside javac (e.g., LPAREN, RPAREN, LCURLY, etc.), not exposed publicly
Counting Tokens	Count each punctuation as an OP token	Each punctuation is a separate token internally, but we can't easily see them from javac alone

- ## WHITESPACE

  - ### What Is Whitespace in Programming?

    Definition: Spaces, tabs, newlines

    Role: Typically separates tokens (identifiers, operators, etc.)

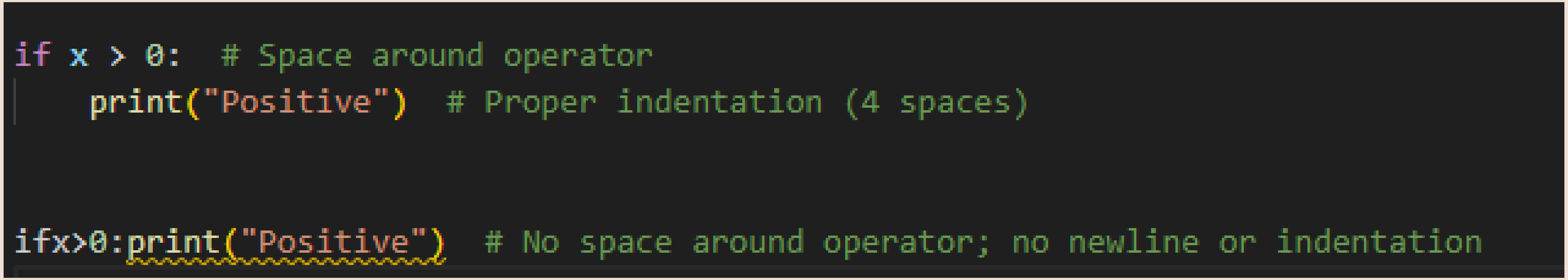
    Visibility: Invisible characters but crucial for readability

  - ### Similarities Between Python & Java

    Separates Tokens: Both languages use spaces/tabs/newlines to separate keywords, identifiers, and operators.

    Ignored at Execution: Does not affect program performance or logic once compiled/interpreted.

    Stylistic Readability: Both encourage using whitespace to make code more readable and structured.



```
if x > 0:  # Space around operator
    print("Positive")  # Proper indentation (4 spaces)


ifx>0:print("Positive")  # No space around operator; no newline or indentation
```

CS51530 – Programming Languages, Interpreters and Compilers

WHITESPACE

- **Python's INDENT, DEDENT → INDENT / DEDENT**

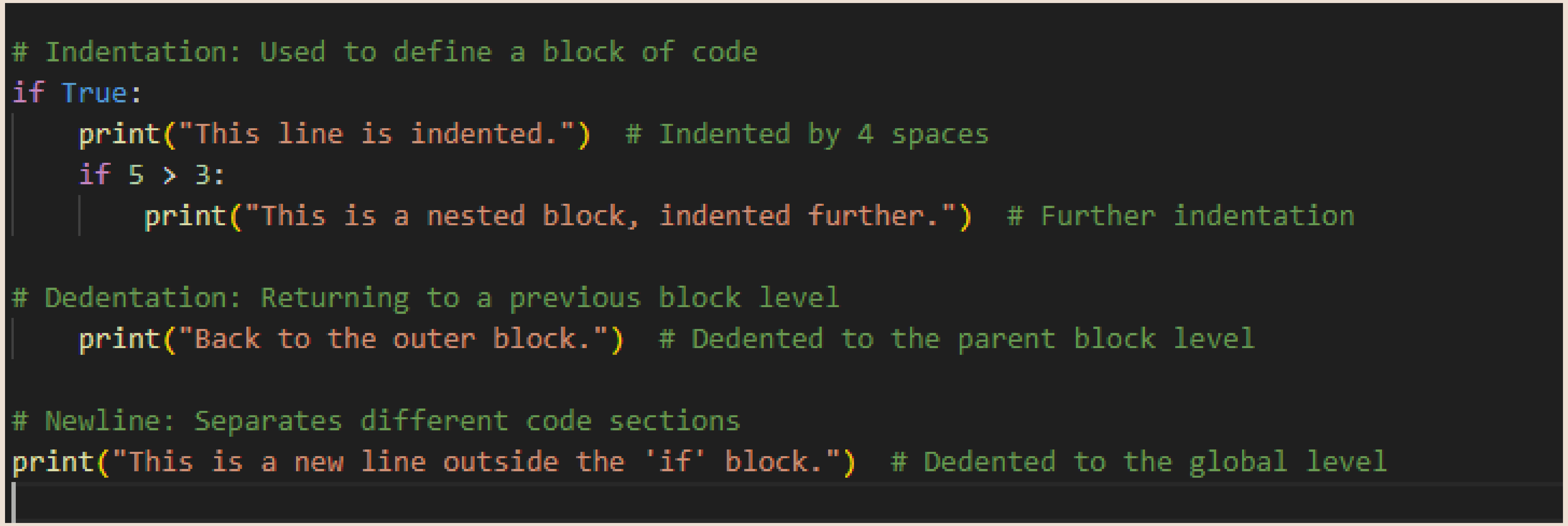
  Defines code blocks (e.g., if condition: then indent)

  Changing indentation can change the program's flow or cause errors

- **Newlines → NEWLINE**

  Typically end statements unless a backslash is used (\)

  Each line break becomes a NEWLINE token in Python's tokenized



```
# Indentation: Used to define a block of code
if True:
    print("This line is indented.")  # Indented by 4 spaces
    if 5 > 3:
        print("This is a nested block, indented further.")  # Further indentation

# Dedentation: Returning to a previous block level
    print("Back to the outer block.")  # Dedented to the parent block level

# Newline: Separates different code sections
print("This is a new line outside the 'if' block.")  # Dedented to the global level
```

CS51530 – Programming Languages, Interpreters and Compilers

• WHITESPACE

TABLE : WHITESPACE SUMMARY TABLE (JAVA VS PYTHON)

Aspect	Python	Java (javac)
Indentation	Significant – yields INDENT/DEDENT tokens	Ignored by compiler; no tokens for indentation
Newlines	Often produce NEWLINE tokens (end statements)	Not meaningful; semicolons end statements
Spaces / Tabs	Not tokenized; just separate tokens	Skipped entirely (no separate tokens)
Effect on Code Structure	Indentation defines blocks	Braces define blocks, whitespace is irrelevant to structure
Counting Whitespace	You see counts of NEWLINE, INDENT, DEDENT in tokenize	javac does not expose whitespace tokens (count = 0)

- **COMMENTS**
  - **Types of comments in Python**
    - **Single-Line Comment**

      Uses # symbol

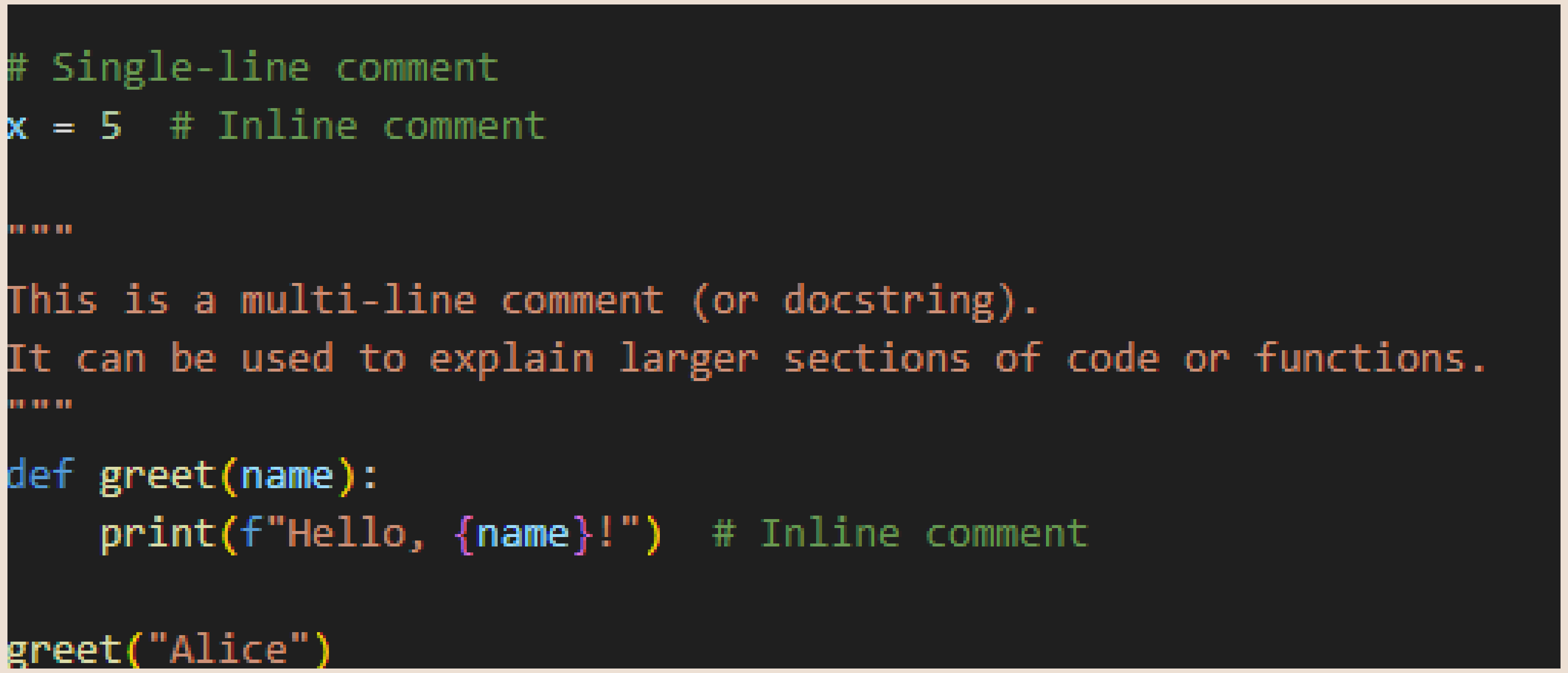
      Extends to the end of the line

      Emitted as COMMENT token by Python's tokenize module
    - **Multi-Line Comments?**

      No dedicated syntax (just multiple # lines)
    - **Docstrings (""" … """)**

      String literals, not comment tokens

      Used for documentation; appear as STRING tokens, not COMMENT



```
# Single-line comment
x = 5  # Inline comment

"""
This is a multi-line comment (or docstring).
It can be used to explain larger sections of code or functions.
"""
def greet(name):
    print(f"Hello, {name}!")  # Inline comment

greet("Alice")
```

- **COMMENTS**

1. Types of Comments in Java

- Single-Line: // ...
- Block (Multi-Line): /* ... */
- Javadoc: /** ... */

2. Java Behavior

- **Discarding Comments:**

  Officially discards comments during lexical analysis.

  Comments do not appear as tokens in the final parse.

- **Compiler Handling:**

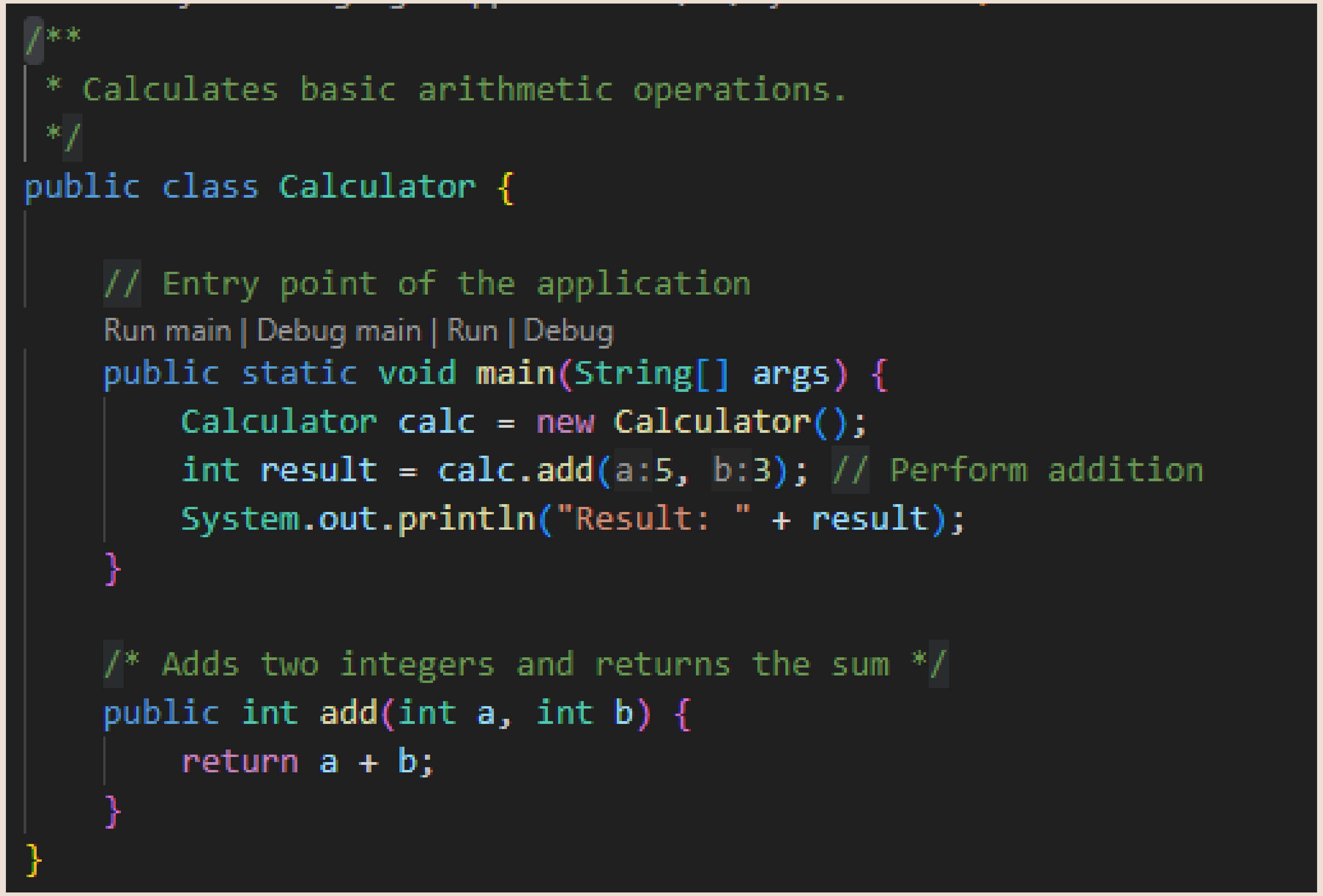
  No standard “comment token” in the AST or bytecode.

  The Java Language Specification states comments are ignored by the compiler.

- **Javadoc Specifics:**

  Javadoc comments are used by the javadoc tool.

  Still ignored by the javac compiler.



```
/**
 * Calculates basic arithmetic operations.
 */
public class Calculator {

    // Entry point of the application
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.add(a:5, b:3); // Perform addition
        System.out.println("Result: " + result);
    }

    /* Adds two integers and returns the sum */
    public int add(int a, int b) {
        return a + b;
    }
}
```

- **COMMENTS**

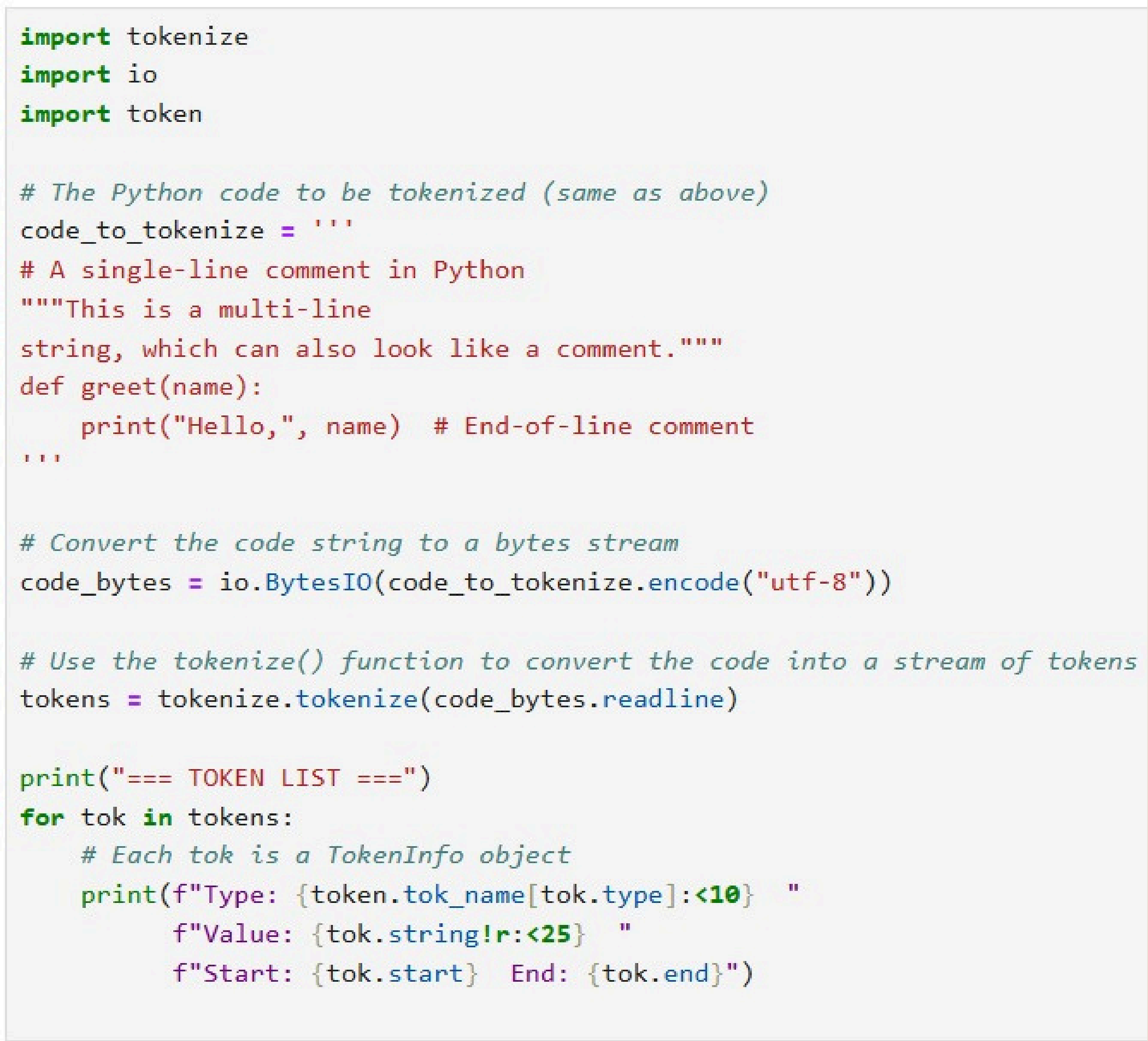
TABLE : COMMENTS SUMMARY TABLE (JAVA VS PYTHON)

Aspect	Python	Java
Single-Line	# ... → COMMENT token	// ... (ignored by compiler)
Multi-Line	None (repeat #)	/* ... */ (ignored by compiler)
Docstring / Javadoc	Docstrings are STRING tokens (""" ... """)	/** ... */ recognized by javadoc, but compiler ignores
Availability	Built-in tokenize module → easy to see COMMENT tokens	No direct compiler support; 3rd-party tools (ANTLR, etc.) needed to see them
Runtime Impact	Ignored by interpreter; no effect on execution	Ignored by compiler; no effect on bytecode

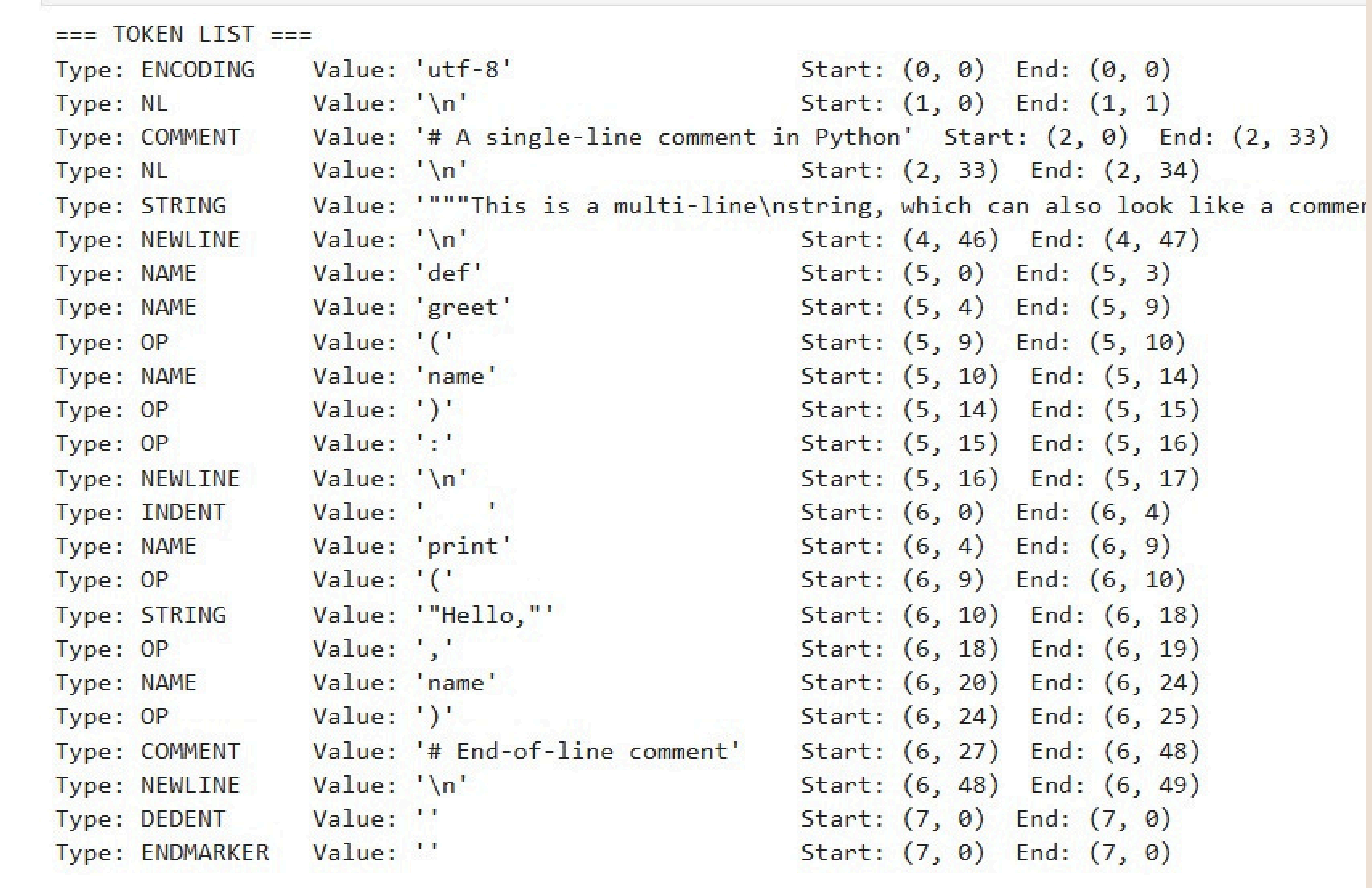
CS51530 – Programming Languages, Interpreters and Compilers

- # PROGRAMMING EXAMPLE

PYTHON:



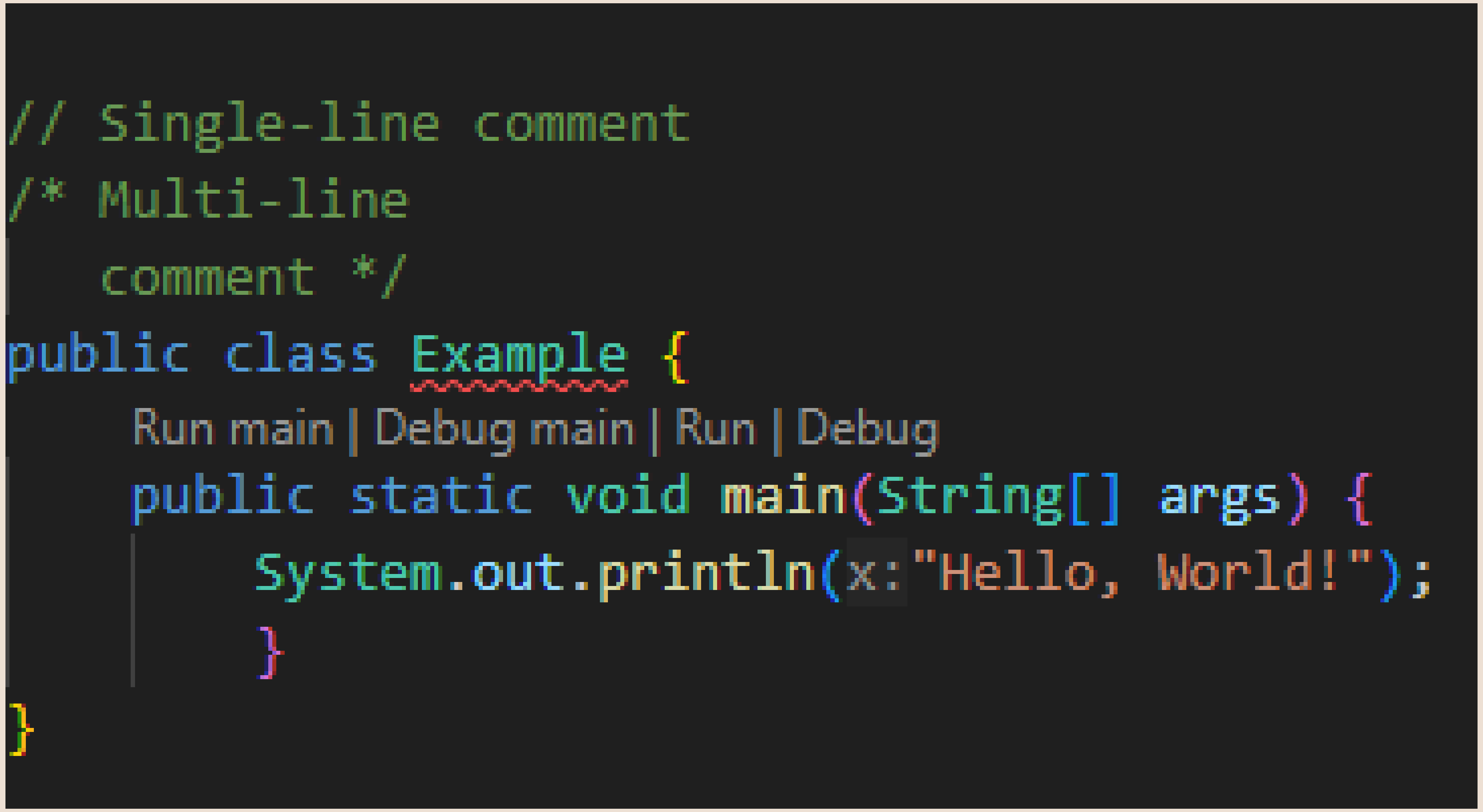
```python
import tokenize
import io
import token

# The Python code to be tokenized (same as above)
code_to_tokenize = '''
# A single-line comment in Python
"""This is a multi-line
string, which can also look like a comment."""
def greet(name):
    print("Hello,", name)  # End-of-line comment
'''

# Convert the code string to a bytes stream
code_bytes = io.BytesIO(code_to_tokenize.encode("utf-8"))

# Use the tokenize() function to convert the code into a stream of tokens
tokens = tokenize.tokenize(code_bytes.readline)

print("=== TOKEN LIST ===")
for tok in tokens:
    # Each tok is a TokenInfo object
    print(f"Type: {token.tok_name[tok.type]:<10}  "
          f"Value: {tok.string!r:<25}  "
          f"Start: {tok.start}  End: {tok.end}")
```



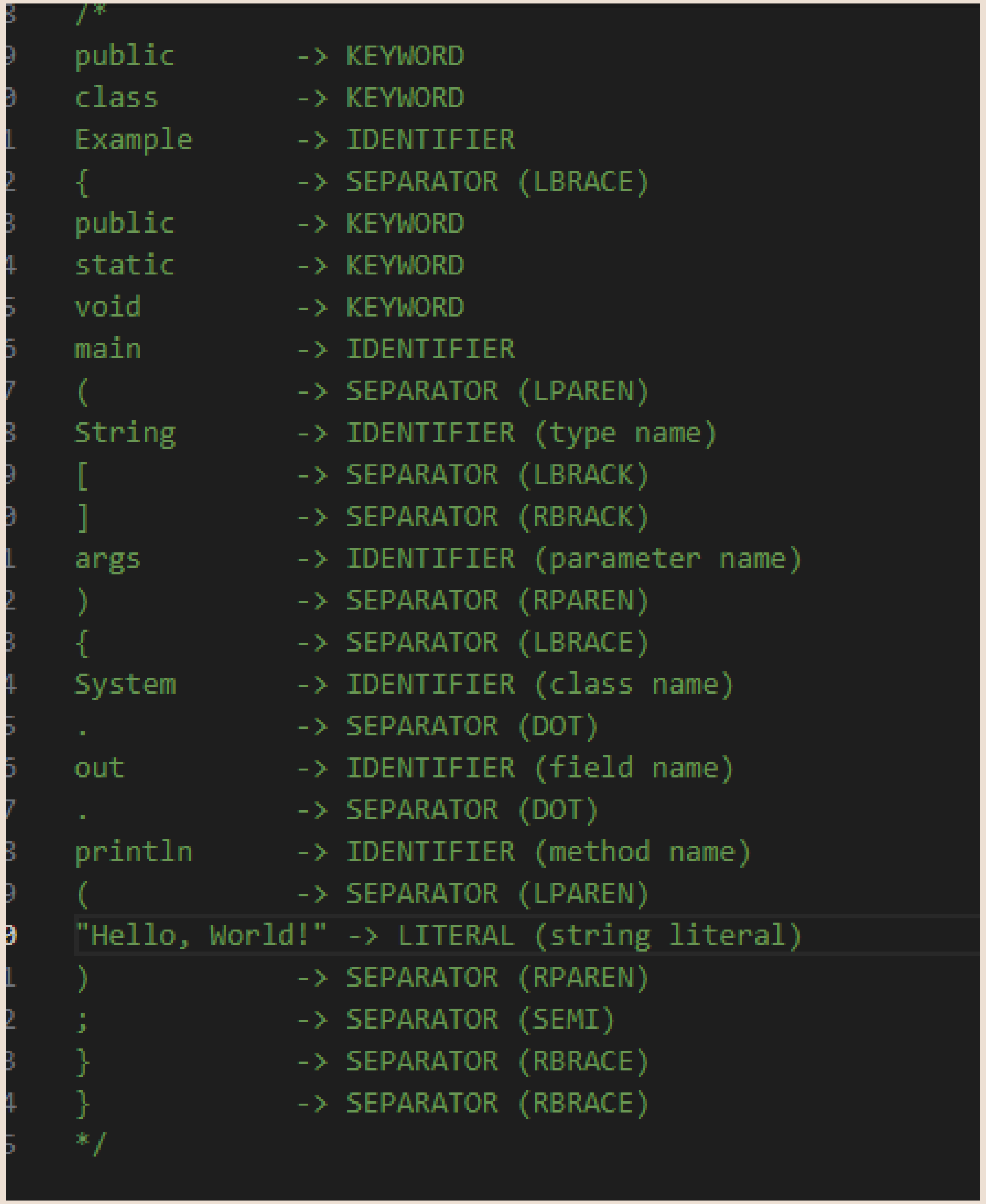
```
=== TOKEN LIST ===
Type: ENCODING    Value: 'utf-8'                    Start: (0, 0)  End: (0, 0)
Type: NL          Value: '\n'                       Start: (1, 0)  End: (1, 1)
Type: COMMENT     Value: '# A single-line comment in Python'  Start: (2, 0)  End: (2, 33)
Type: NL          Value: '\n'                       Start: (2, 33)  End: (2, 34)
Type: STRING      Value: '"""This is a multi-line\nstring, which can also look like a commer
Type: NEWLINE     Value: '\n'                       Start: (4, 46)  End: (4, 47)
Type: NAME        Value: 'def'                      Start: (5, 0)  End: (5, 3)
Type: NAME        Value: 'greet'                    Start: (5, 4)  End: (5, 9)
Type: OP          Value: '('                        Start: (5, 9)  End: (5, 10)
Type: NAME        Value: 'name'                     Start: (5, 10)  End: (5, 14)
Type: OP          Value: ')'                        Start: (5, 14)  End: (5, 15)
Type: OP          Value: ':'                        Start: (5, 15)  End: (5, 16)
Type: NEWLINE     Value: '\n'                       Start: (5, 16)  End: (5, 17)
Type: INDENT      Value: '    '                     Start: (6, 0)  End: (6, 4)
Type: NAME        Value: 'print'                    Start: (6, 4)  End: (6, 9)
Type: OP          Value: '('                        Start: (6, 9)  End: (6, 10)
Type: STRING      Value: '"Hello,"'                 Start: (6, 10)  End: (6, 18)
Type: OP          Value: ','                        Start: (6, 18)  End: (6, 19)
Type: NAME        Value: 'name'                     Start: (6, 20)  End: (6, 24)
Type: OP          Value: ')'                        Start: (6, 24)  End: (6, 25)
Type: COMMENT     Value: '# End-of-line comment'    Start: (6, 27)  End: (6, 48)
Type: NEWLINE     Value: '\n'                       Start: (6, 48)  End: (6, 49)
Type: DEDENT      Value: ''                         Start: (7, 0)  End: (7, 0)
Type: ENDMARKER   Value: ''                         Start: (7, 0)  End: (7, 0)
```

- **PROGRAMMING EXAMPLE**

JAVA



```java
// Single-line comment
/* Multi-line
   comment */
public class Example {
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {
        System.out.println(x:"Hello, World!");
        }
}
```



```
/*
public         -> KEYWORD
class          -> KEYWORD
Example        -> IDENTIFIER
{              -> SEPARATOR (LBRACE)
public         -> KEYWORD
static         -> KEYWORD
void           -> KEYWORD
main           -> IDENTIFIER
(              -> SEPARATOR (LPAREN)
String         -> IDENTIFIER (type name)
[              -> SEPARATOR (LBRACK)
]              -> SEPARATOR (RBRACK)
args           -> IDENTIFIER (parameter name)
)              -> SEPARATOR (RPAREN)
{              -> SEPARATOR (LBRACE)
System         -> IDENTIFIER (class name)
.              -> SEPARATOR (DOT)
out            -> IDENTIFIER (field name)
.              -> SEPARATOR (DOT)
println        -> IDENTIFIER (method name)
(              -> SEPARATOR (LPAREN)
"Hello, World!" -> LITERAL (string literal)
)              -> SEPARATOR (RPAREN)
;              -> SEPARATOR (SEMI)
}              -> SEPARATOR (RBRACE)
}              -> SEPARATOR (RBRACE)
*/
```

• CONCLUSION

1. Separators

- **Java**: Uses symbols like {, }, (,), ;, and , to define code blocks, method calls, and statement boundaries.
- **Python**: Minimizes separators and instead relies on indentation for code blocks, but still uses punctuation (e.g., :, (,), ,) for structure.

2. Whitespace

- **Java**: Whitespace separates tokens but is otherwise not significant for code structure. Indentation is a style choice, not enforced by the language.
- **Python**: Whitespace (indentation) defines block scope and is syntactically significant, making consistent use of spaces or tabs essential.

3. Comments

- **Java**: Single-line (//) and multi-line (/* … */) comments. The compiler ignores them, but they remain in the source to guide readers.
- **Python**:Single-line(#) and stringdoc(“””” “”””)comments comes under COMMENT and STRING token repectively.

CS51530 – Programming Languages, Interpreters and Compilers

Thank you for listening!

CS51530 – Programming Languages, Interpreters and Compilers

Assignment - 1