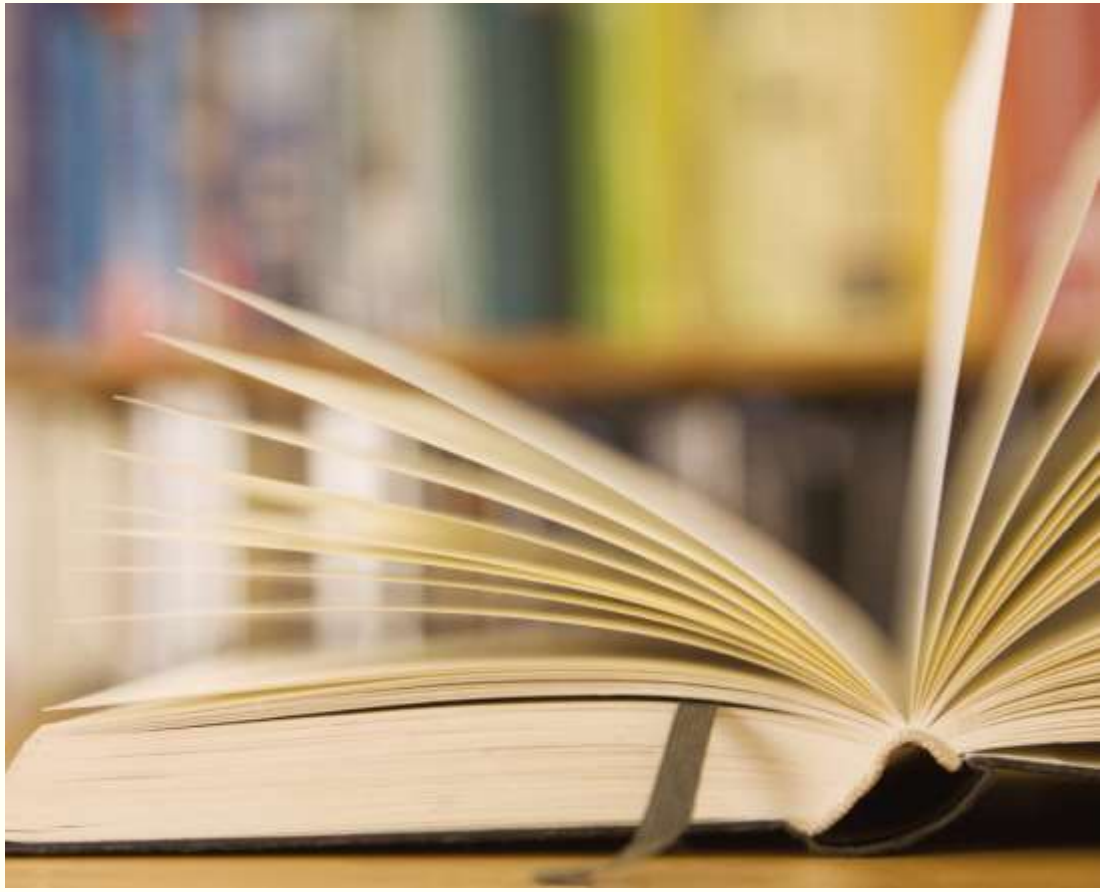
7. PROGRAMMING



FPT Education

FPT UNIVERSITY

Content

- 7.1 Evolution
- 7.2 Translation
- 7.3 Programming paradigms
- 7.4 Common concepts

Objectives

After studying this chapter, the student should be able to:

- Describe the evolution of programming languages from machine language to high-Level languages.
- Understand how a program in a high-level language is translated into machine language using an interpreter or a compiler.
- Distinguish between four computer language paradigms.
- Understand the procedural paradigm and the interaction between a program unit and data items in the paradigm.
- Understand the object-oriented paradigm and the interaction between a program unit and objects in this paradigm.
- Define functional paradigm and understand its applications.
- Define a declaration paradigm and understand its applications.
- Define common concepts in procedural and object-oriented languages.

1 - EVOLUTION

1. EVOLUTION

- To write a program for a computer, we must use a computer language. A **computer language** is a set of predefined words that are combined into a program according to predefined rules **(syntax)**. Over the years, computer languages have evolved from *machine language* to *high- level languages*.

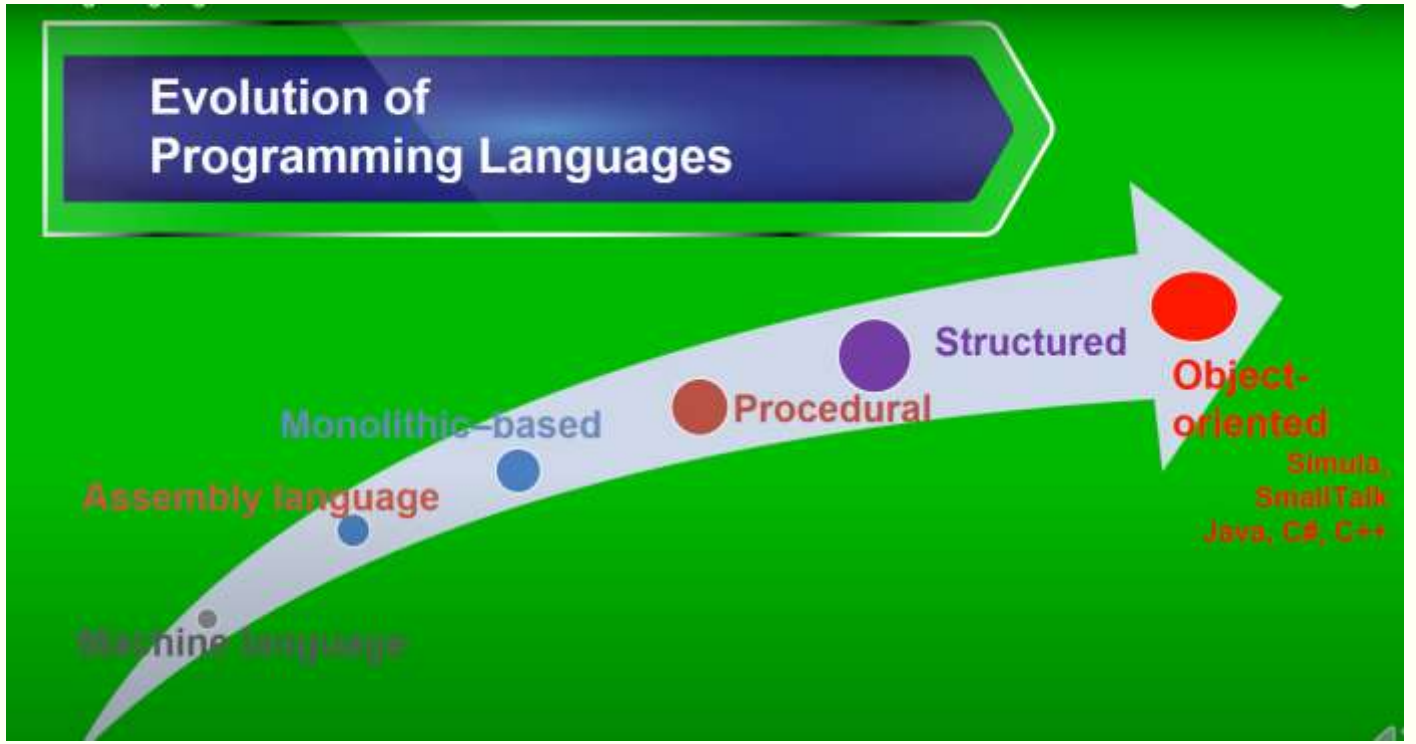


Figure 7.1 Evolution of programing

2. Machine languages

- In the earliest days of computers, the only programming languages available were machine languages.
- Each computer had its own machine language, which was made of streams of 0s and 1s.
- Use eleven lines of code to read two integers, add them, and print the result. These lines of code, when written in machine language, make eleven lines of binary code, each of 16 bits, as shown in Table 8.1.

Hexadecimal	*Code in machine language*			
$(1FEF)_{16}$	0001	1111	1110	1111
$(240F)_{16}$	0010	0100	0000	1111
$(1FEF)_{16}$	0001	1111	1110	1111
$(241F)_{16}$	0010	0100	0001	1111
$(1040)_{16}$	0001	0000	0100	0000
$(1141)_{16}$	0001	0001	0100	0001
$(3201)_{16}$	0011	0010	0000	0001
$(2422)_{16}$	0010	0100	0010	0010
$(1F42)_{16}$	0001	1111	0100	0010
$(2FFF)_{16}$	0010	1111	1111	1111
$(0000)_{16}$	0000	0000	0000	0000

Table 7.1 Code in machine language to add two integers

2. Assembly languages

- The next evolution in programming came with the idea of replacing binary code for instruction and addresses with symbols or mnemonics. Because they used symbols, these languages were first known as **symbolic languages**.

- The set of these mnemonic languages were later referred to as **assembly languages**. The assembly language for our hypothetical computer to replace the machine language in Table 8.2 is shown in Table 9.2.

Code in assembly language				*Description*
LOAD	RF	Keyboard		Load from keyboard controller to register F
STORE	Number1	RF		Store register F into Number1
LOAD	RF	Keyboard		Load from keyboard controller to register F
STORE	Number2	RF		Store register F into Number2
LOAD	R0	Number1		Load Number1 into register 0
LOAD	R1	Number2		Load Number2 into register 1
ADDI	R2	R0	R1	Add registers 0 and 1 with result in register 2
STORE	Result	R2		Store register 2 into Result
LOAD	RF	Result		Load Result into register F
STORE	Monitor	RF		Store register F into monitor controller
HALT				Stop

Table 7.2 Code in machine language to add two integers

3. High-level languages

- Although assembly languages greatly improved programming efficiency, they still required programmers to concentrate on the hardware they were using. Working with symbolic languages was also very tedious, because each machine instruction had to be individually coded.
- The desire to improve programmer efficiency and to change the focus from the computer to the problem being solved led to the development of **high-level language**
- Over the years, various languages, most notably BASIC, COBOL, Pascal, Ada, C, C+ +, and Java, were developed. Figure 8-3 shows the code for adding two integers as it would appear in the C++

```
/*      This program reads two integers from keyboard and prints their sum.
        Written by:
        Date:
*/
#include <iostream>
using namespace std;
int main ()
{
        // Local Declarations
        int number1;
        int number2;
        int result;
        // Statements
        cin >> number1;
        cin >> number2;
        result = number1 + number2;
        cout << result;
        return 0;
} // main
```

Figure 7.2 Code in machine language to add two integers

2- TRANSLATION

1. Introduction

- Programs today are normally written in one of the high-level languages. To run the program on a computer, the program needs to be translated into the machine language of the computer on which it will run.

- The program in a high-level language is called the **source program**. The translated program in machine language is called the **object program**.

- Two methods are used for translation: **compilation** and **interpretation**.

❑ **Compilation :** A **compiler** normally translates the whole source program into the object program.

❑ **Interpretation :** Some computer languages use an **interpreter** to translate the source program into the object program. Interpretation refers to the process of translating each line of the source program into the corresponding line of the object program and executing the line.

2. Translation process

- Compilation and interpretation differ in that the first translates the whole source code before executing it, while the second translates and executes the source code a line at a time.

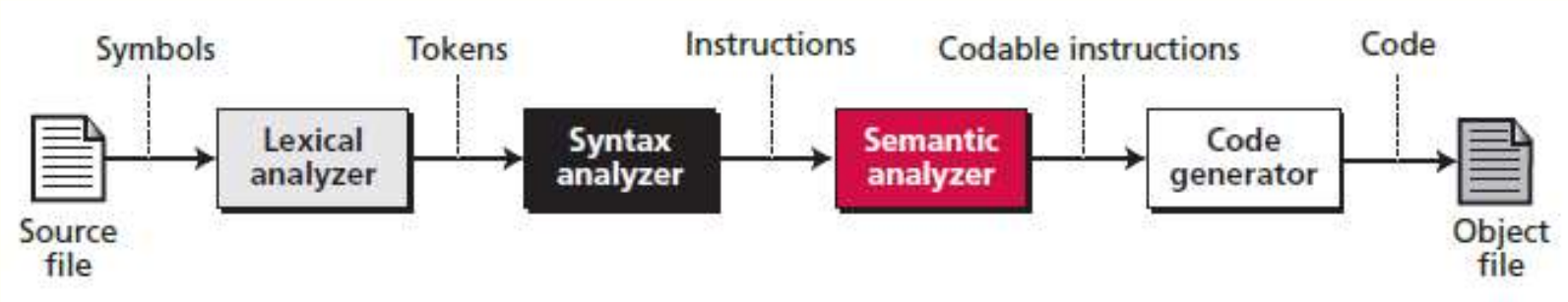


Figure 7.3 Evolution of programing

- ***Lexical analyzer*** reads the source code, symbol by symbol, and creates a list of **tokens** in the source language.
- ***Syntax analyzer*** parses a set of tokens to find instructions.
- ***Semantic analyzer*** checks the sentences created by the syntax analyzer to be sure that they contain no ambiguity.
- ***Code generator*** After unambiguous instructions are created by the semantic analyzer, each instruction is converted to a set of machine language instructions for the computer on which the program will run.

3 - PROGRAMMING PARADIGMS

1. Introduction

- Today computer languages are categorized according to the approach they use to solve a problem. A paradigm, therefore, is a way in which a computer language looks at the problem to be solved. We divide computer languages into four paradigms: procedural, object-oriented, functional, and declarative

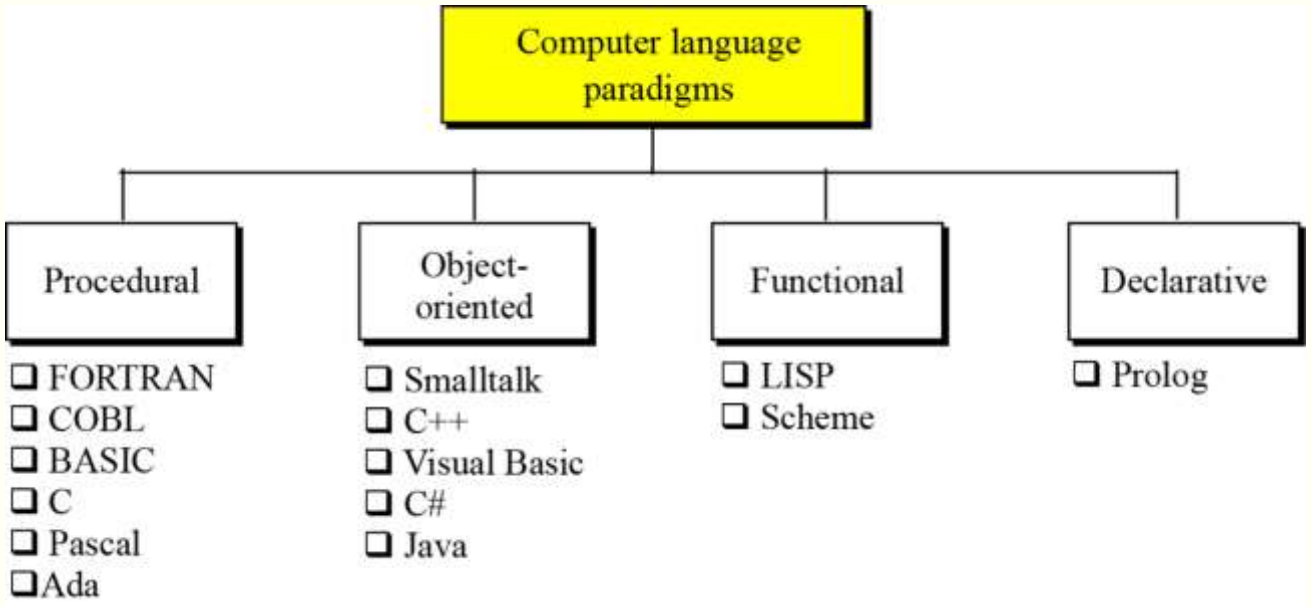


Figure 7.4 *Categories of programming languages*

2. The procedural paradigm

- In the procedural paradigm (or imperative paradigm) we can think of a program as an active agent that manipulates passive objects. We encounter many passive objects in our daily life: a stone, a book, a lamp, and so on. A passive object cannot initiate an action by itself, but it can receive actions from active agents.

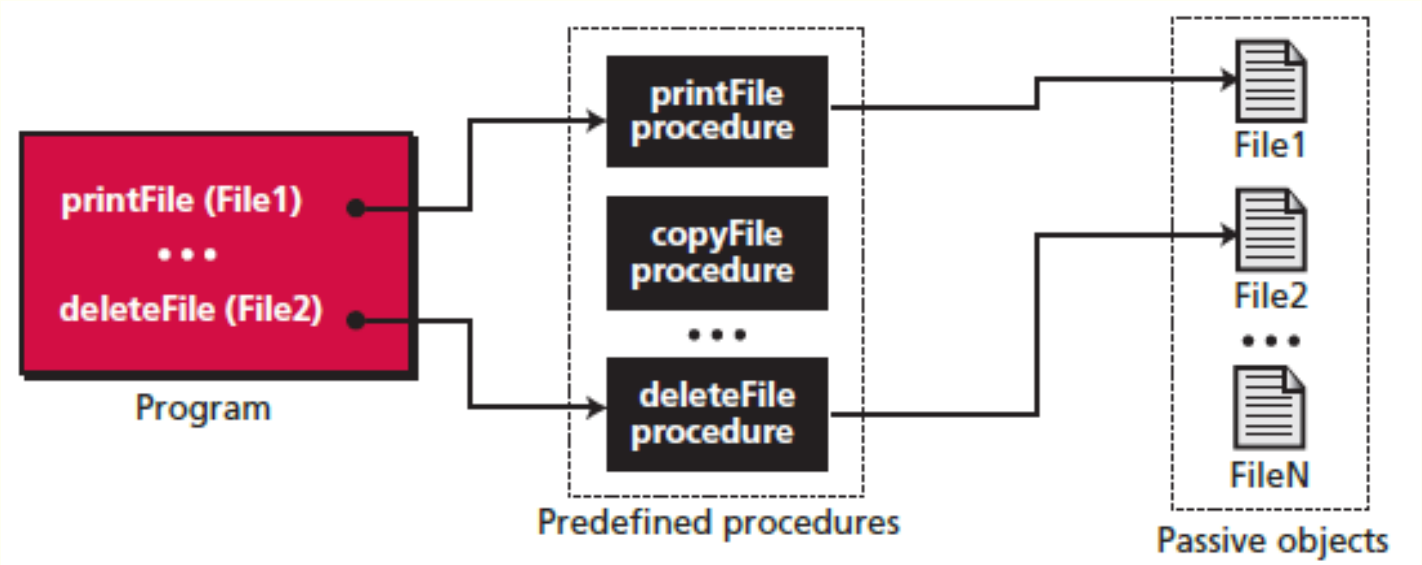


Figure 7.5 *The concept of the procedural paradigm*

3. The object-oriented paradigm

- The object-oriented paradigm deals with active objects instead of passive objects. We encounter many active objects in our daily life: a vehicle, an automatic door, a dishwasher, and so on.
- The actions to be performed on these objects are included in the object: the objects need only to receive the appropriate stimulus from outside to perform one of the actions

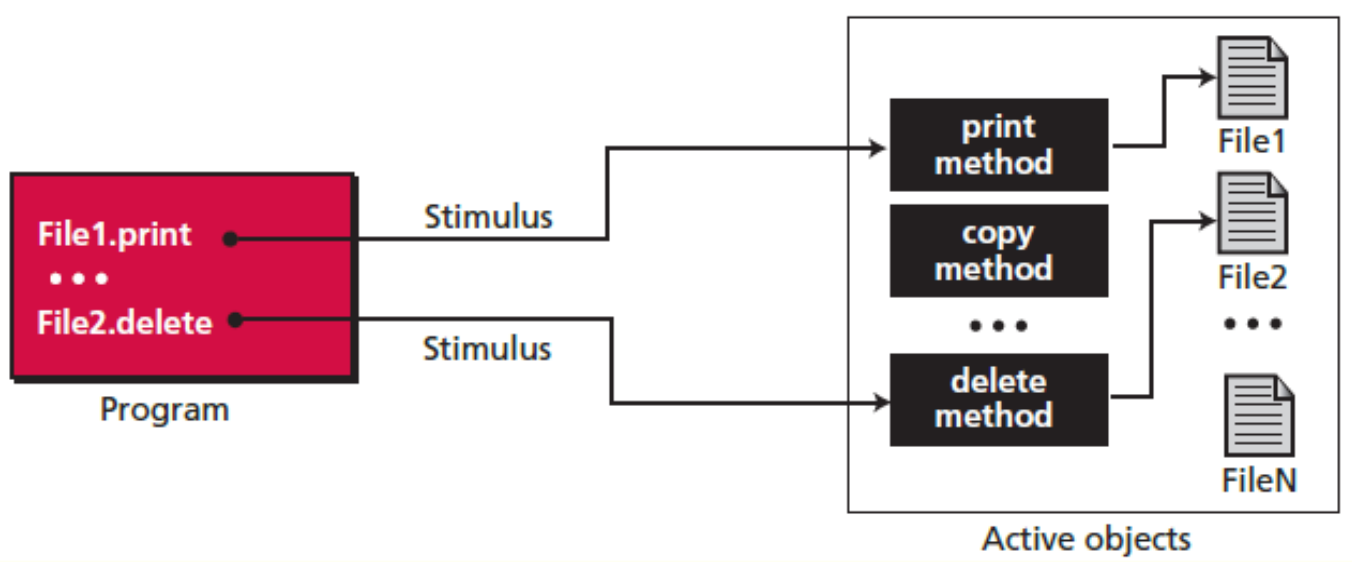


Figure 7.6 *The concept of an object-oriented paradigm*

4. The functional paradigm

- In the functional paradigm a program is considered a mathematical function. In this context, a function is a **black box** that maps a list of **inputs** to a list of **outputs**.

Figure 7.7 *The concept of an object-oriented paradigm*

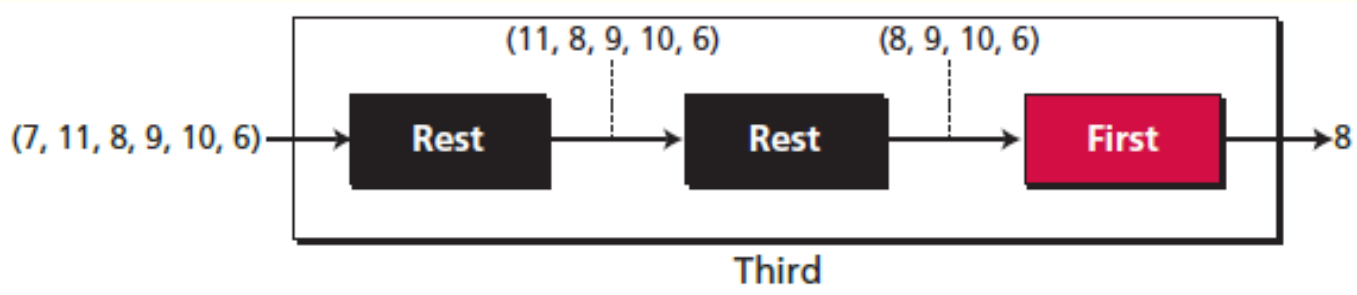


Figure 7.8 *Extracting the third element of a list*

5. The declarative paradigm

- A declarative paradigm uses the principle of logical reasoning to answer queries. It is based on formal logic defined by Greek mathematicians and later developed into first-order predicate calculus.
- For example, the famous rule of deduction in logic is:

If (A is B) and (B is C), then (A is C)

- Using this rule and the two following facts:

Fact 1: Socrates is a human → A is B
Fact 2: A human is mortal → B is C

- we can deduce a new fact:

Fact 3: Socrates is mortal → A is C

4- COMMON CONCEPTS

1. Introduction

- In this section we conduct a quick navigation through some procedural languages to find common concepts.
- Some of these concepts are also available in most object-oriented languages because, as we explained, an object-oriented paradigm uses the procedural paradigm when creating methods..
  - ❑ Identifiers
  - ❑ Data types
  - ❑ Variables
  - ❑ Literals
  - ❑ Constants
  - ❑ Inputs and Outputs

2. Identifiers

- One feature present in all procedural languages, as well as in other languages, is the identifier—that is, the name of objects. Identifiers allow us to name objects in the program.

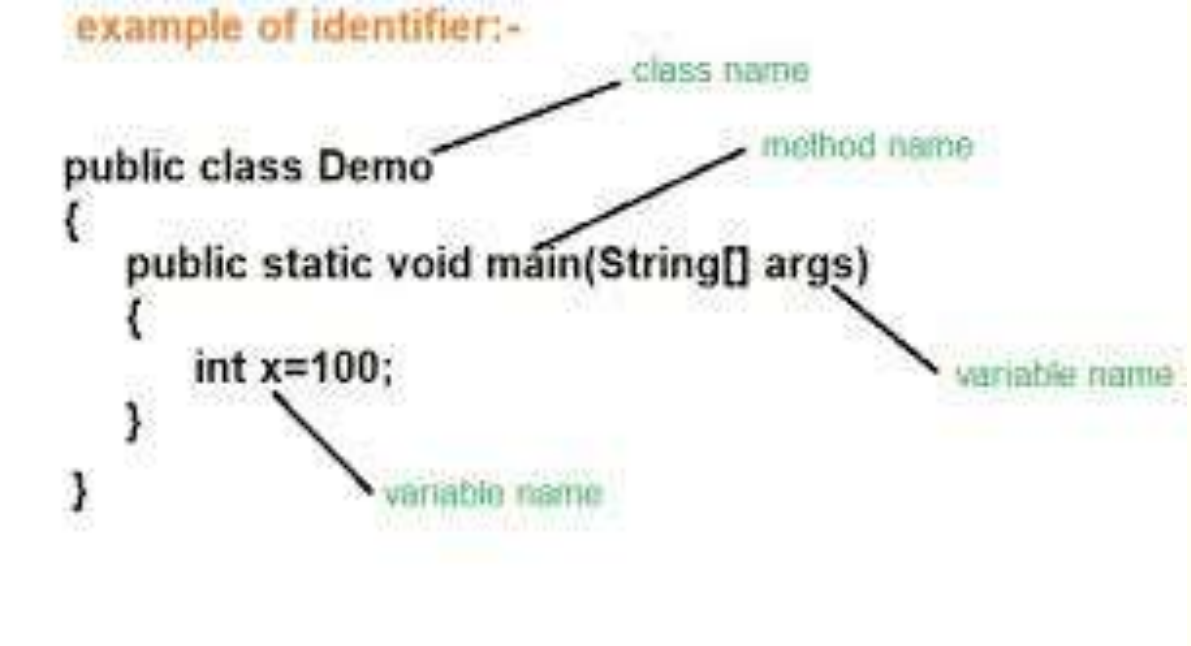


Figure 7.10 Example of Identifiers

3. Data types

- A data type defines a set of values and a set of operations that can be applied to those values. The set of values for each type is known as the domain for the type.
- Most languages define two categories of data types: simple types (Primitive in Java) and composite types (Non-Primitive in Java)

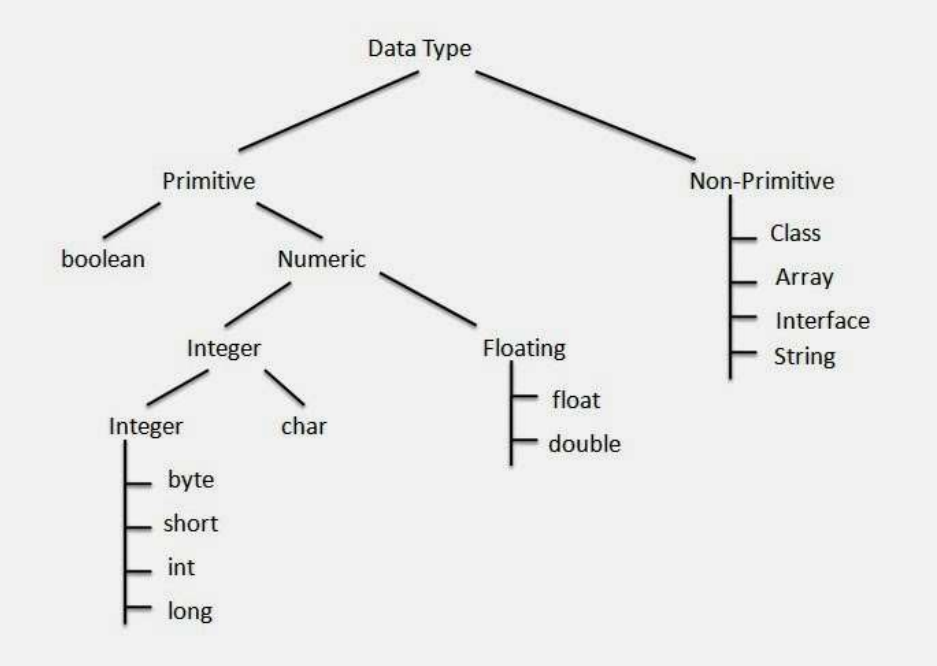


Figure 7.11 Example of data type

4. Variables

- Variables are names for memory locations. Although the addresses are used by the computer internally, it is very inconvenient for the programmer to use addresses.
- A programmer can use a variable, such as count, to store the integer value of a count number received in a test.

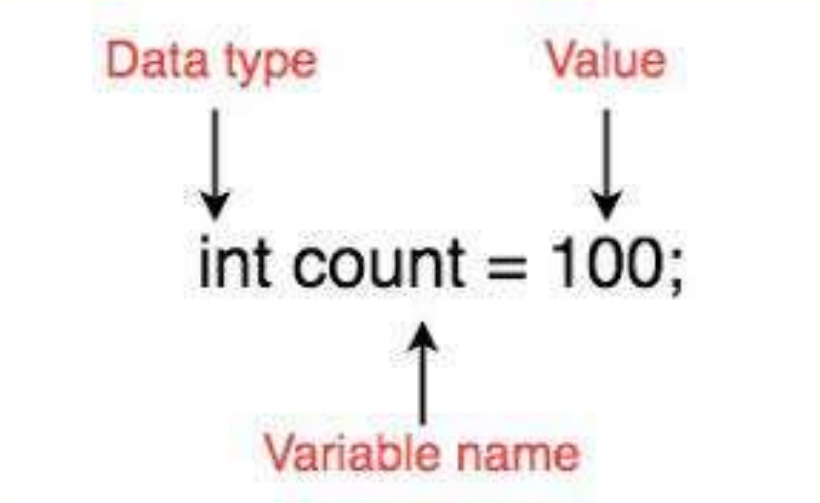


Figure 7.12 Example of variable

5. Literals

- A literal is a predetermined value used in a program. For example, if we need to calculate the area of circle when the value of the radius is stored in the variable r, we can use the expression 3.14 x r2, in which the approximate value of π (pi) is used as a literal.

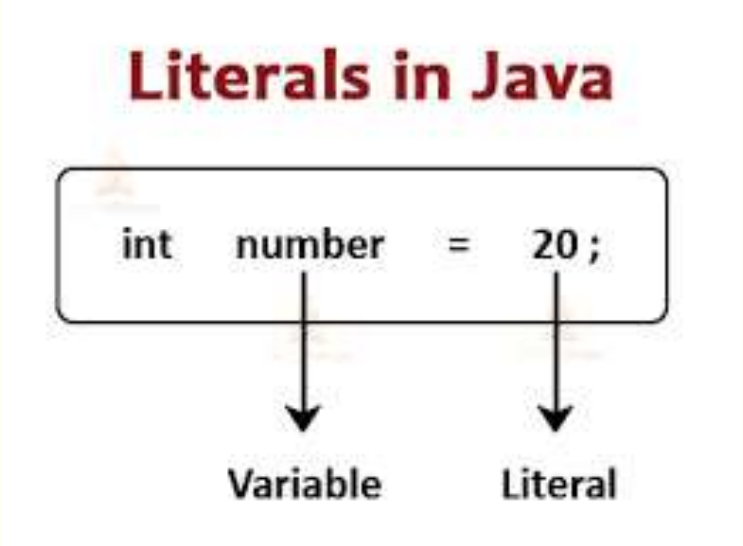


Figure 7.13 Example of literal

5. Constants

- The use of literals is not considered good programming practice unless we are sure that the value of the literal will not change with time (such as the value of π in geometry).
- However, most literals may change value with time. For example, if a sales tax is 8 per cent this year, it may not be the same next year.

```
const float taxMultiplier = 1.08;
...
cost  = price * taxMultiplier;
```

Figure 7.14 Example of constants

7. Inputs and Outputs

- Almost every program needs to read and/or write data. These operations can be quite complex, especially when we read and write large files. Most programming languages use a predefined function for input and output.
- Input : Data is input by either a statement or a predefined function. Example in C language

```
scanf ("%d", &num);
```

- Output : Output Data is output by either a statement or a predefined function

```
printf ("The value of the number is: %d", num);
```