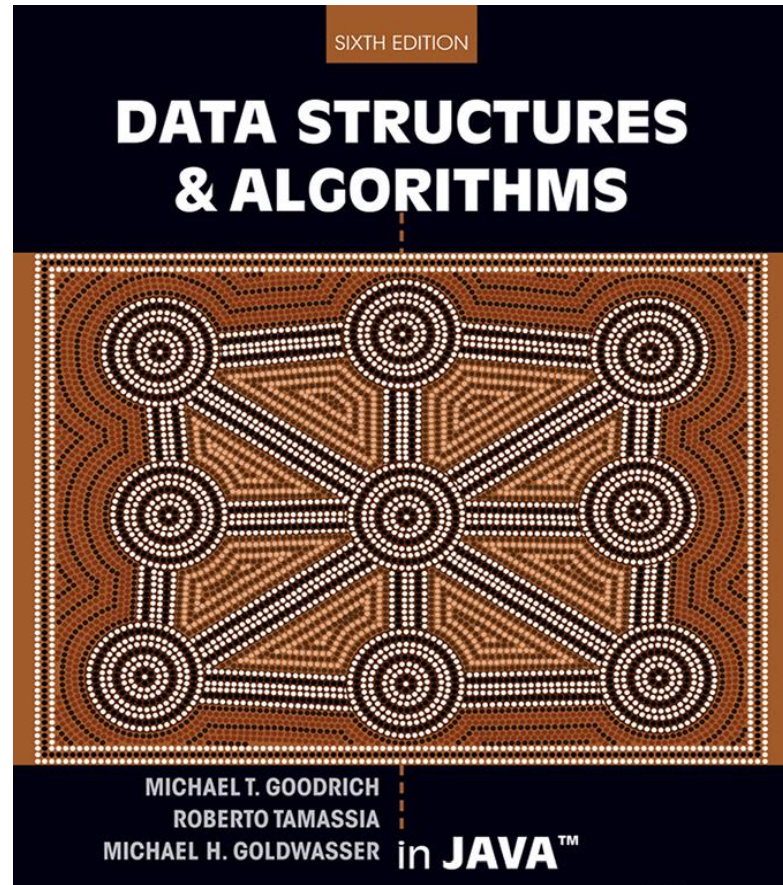


DATA STRUCTURES & ALGORITHMS



Data Structures & Algorithms

Lecture 1

Introduction to DSA, Java and OOP

Introduction to Course

- Build an application or website that utilizes basic or complex algorithms to solve a real-life problem
- Propose an alternative algorithm to replace the aforementioned one
- Evaluate the efficiency of the algorithm mentioned above

Learning Outcomes

- LO1: Examine abstract data types, concrete data structures and algorithms
- LO2: Specify abstract data types and algorithms in a formal notation
- LO3: Implement complex data structures and algorithms
- LO4: Assess the effectiveness of data structures and algorithms

Materials of Course

- Login to CMS (<http://cms.btec.edu.vn/>) with student account (name@fpt.edu.vn)
- Choose correct class
- Enroll by class name (SE0xx0x)
- CMS Folder:
 - Slides
 - Lab
 - Assignment
 - EBook

Preparation for Course

- Drawing tools (choose one):
 - Visio
 - Draw.io or Lucichart (online)
 - Astah (recommendation, using student email to register full version)
- IDE
 - Apache NetBeans IDE 17
 - JDK 17

Topics

- Overview of data structures and algorithms
- Introduction to Java Basic
- Overview to Object-Oriented

What is Data Structures?

- **A data structure is defined by**

- (1) The logical arrangement of data elements, combined with
- (2) The set of operations we need to access the elements

Why do we need data structures?



Intuition/Expectation

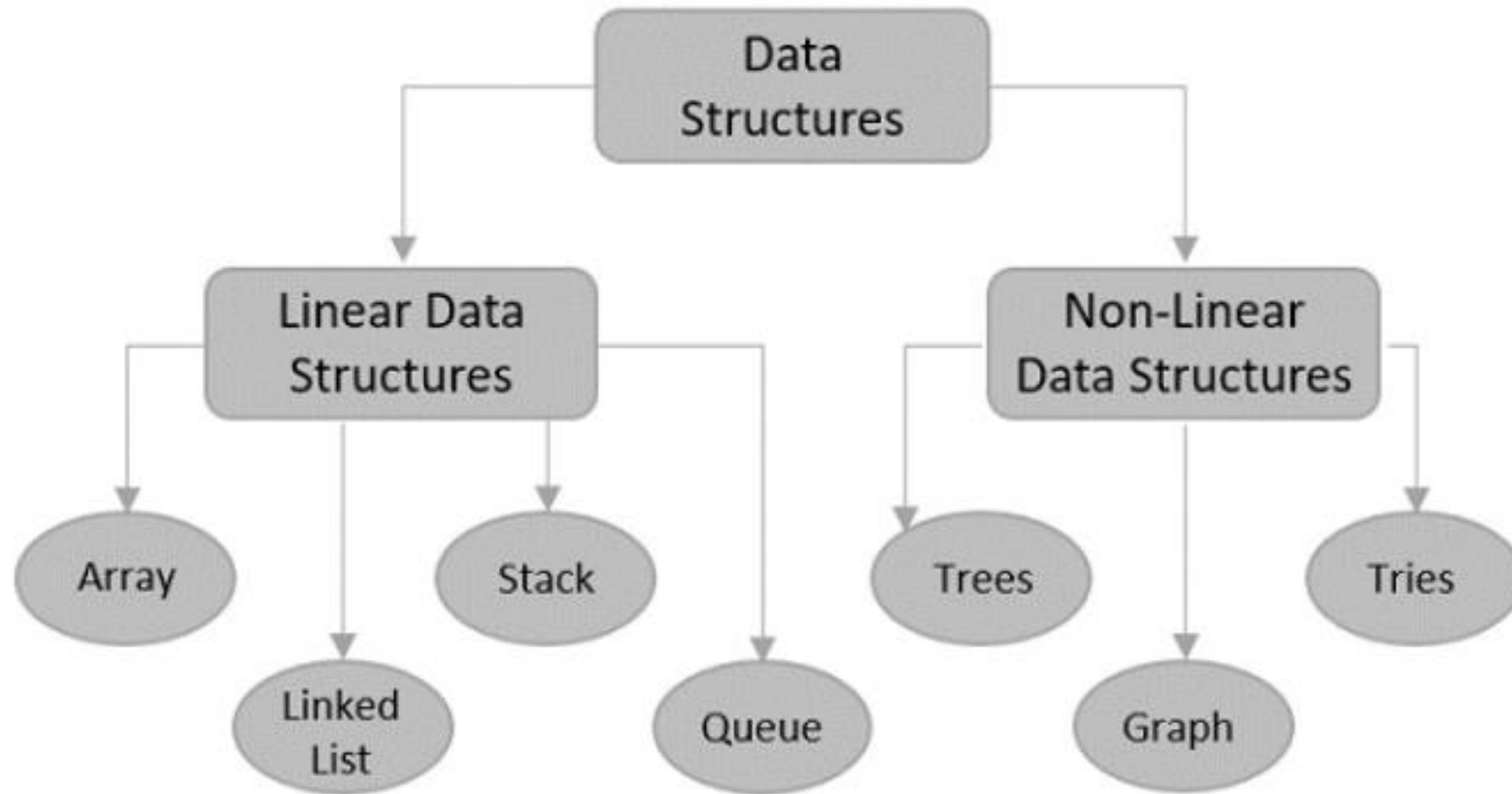
if we want to find a given book as fast as possible then we need complicated algorithms

Reality

if we store the books in an efficient manner then there is no need for complicated algorithms

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships” Linus Torvalds

Data Structures



What is Abstract Data Types (ADT)? (1/2)

Abstract Data Type (ADT) is a theoretical concept in computer science that defines a data structure in terms of its behavior from the perspective of a user, rather than its implementation. It focuses on what the data structure does rather than how it does it.

What is Abstract Data Types (ADT)? (2/2)

Key Features of ADT:

Abstraction:

- Hides the implementation details
- Users interact with the data structure through a well-defined interface

Encapsulation:

- Combines data and associated operations
- Prevents direct access to internal representation

Behavior-Oriented:

- Defined by a set of operations (e.g., insert, delete, search) and their expected outcomes

What is Algorithms? (1/2)

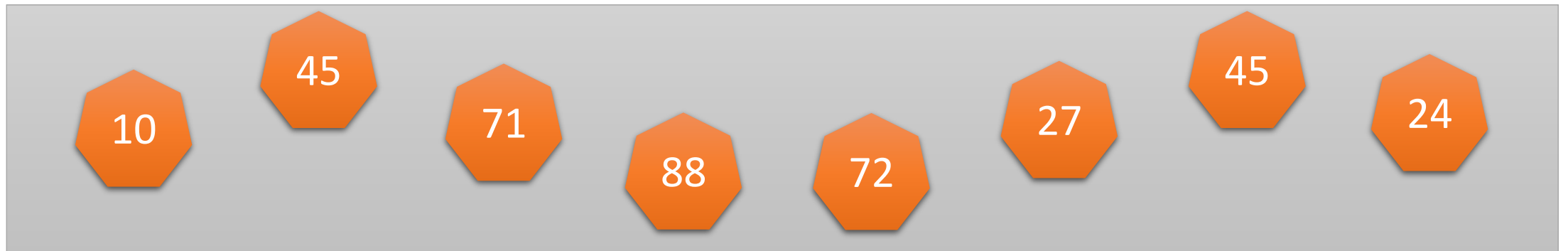
■ Scenario

						
North America Grounded NEMA 5-15	Japan Non-grounded JIS C 8303	Europe German style CEE7/4 Schuko	Europe French style Schuko	Europe/Russia Non-grounded CEE7/16 Europlug	Great Britain Grounded BS-1363	Great Britain "Shaver socket" BS-4573
						
Australia/China Grounded AS-3112	Italy Grounded CEI 23-16	Switzerland Grounded SEV-1011	Denmark Grounded SRAF 1962/DB	Israel Grounded SI 32 (IS 16A-R)	India Grounded BS-546 "Small"	South Africa Grounded BS-546 "Large"

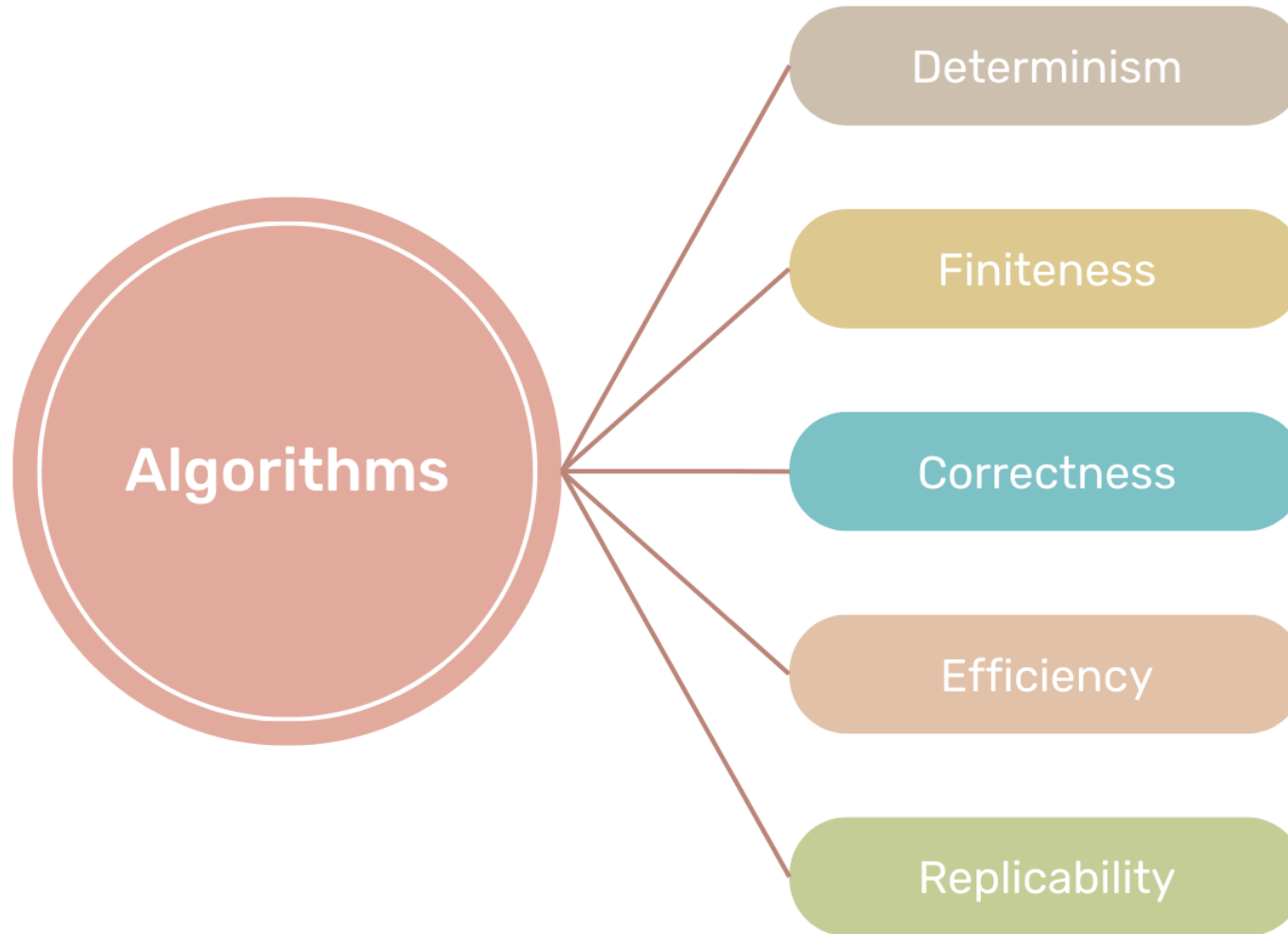


What is Algorithm? (2/2)

- A computable set of steps to achieve a desired result
- Relationship to Data Structures
 - Ex: Find an element, Sort list



Key Characteristics of Algorithms Include (1/3)



Key Characteristics of Algorithms Include (2/3)

- **Input:** An algorithm takes input values or data to work with
- **Output:** It produces an output or result based on the input and the steps performed
- **Determinism:** Algorithms are deterministic, meaning that given the same input, they will always produce the same output
- **Finiteness:** Algorithms must terminate after a finite number of steps

Key Characteristics of Algorithms Include (3/3)

- **Correctness:** Algorithms must produce the correct output for all valid inputs
- **Efficiency:** Algorithms aim to be efficient, optimizing the use of time and resources required to solve a problem
- **Replicability:** Algorithms can be implemented in any programming language and replicated on different systems

Algorithms Representation

- Natural language
- Flowchart
- Pseudo code

Natural Language (1/2)

- List sequentially the steps in natural language to represent the algorithms
- Advantages: Simple, no knowledge of representation (pseudocode, flowchart,...)
- Disadvantages: Long, unstructured. Sometimes it's hard to understand, can't express the algorithm

Natural Language (2/2)

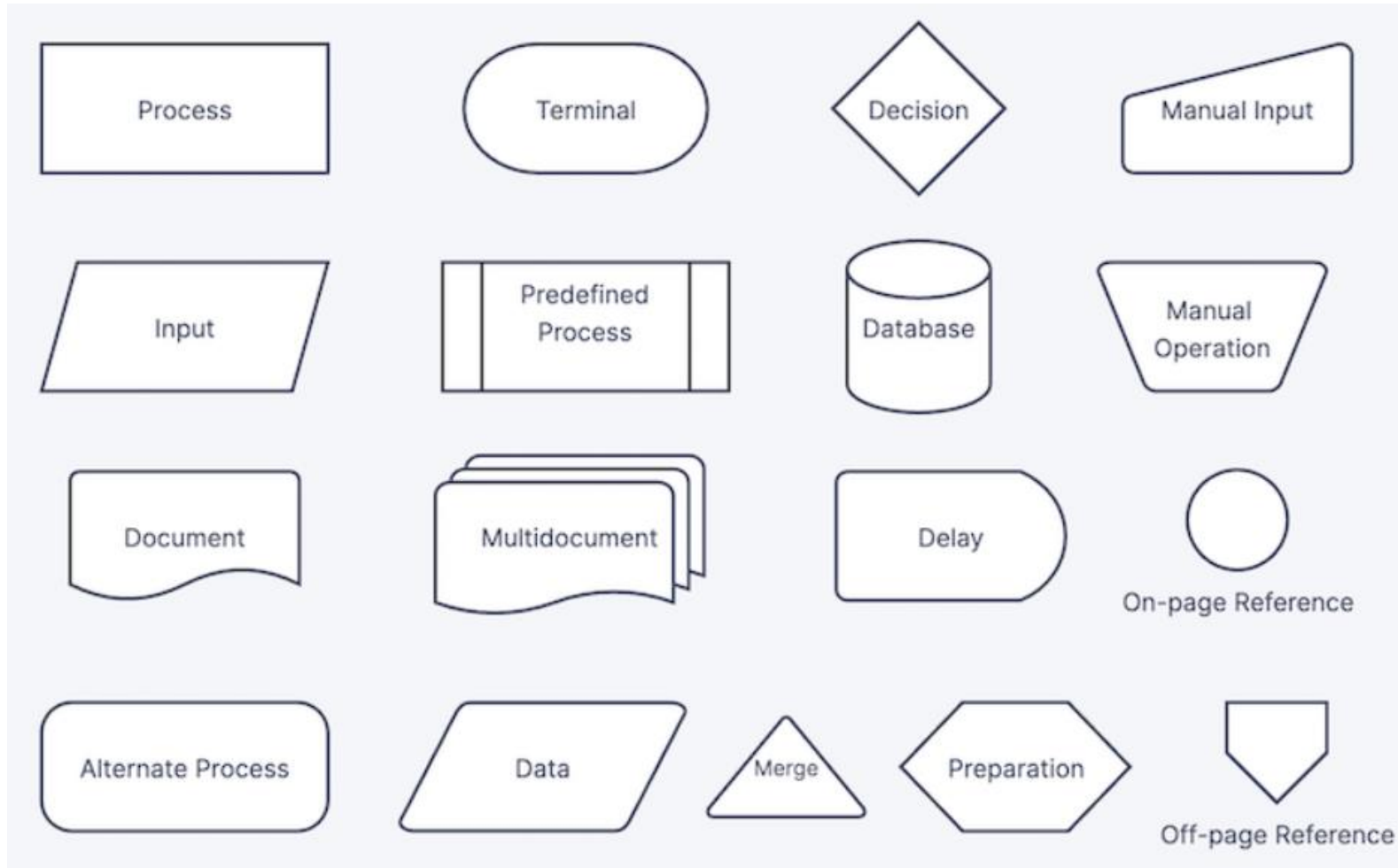
Ex: Using natural language to represent an algorithms for calculating the sum of two integers a and b:

- Input: Two integers a, b.
- Output: The sum of the two integers a, b.
- Algorithms:
 - Step 1: Enter the values of a and b;
 - Step 2: Calculate $\text{Sum} = a + b$;
 - Step 3: Display the result Sum;
 - Step 4: End.

Flowchart

- Flowchart is a tool used to represent algorithms, describe input, output data, and flow of processing through graphical symbols.
- Advantages: Easy to understand, Intuitive, Easy to maintain and modify, ...
- Disadvantages: Limitations of flowcharts, Difficult to track, Conversion capability,...

Flowchart Symbols (1/4)



Flowchart Symbols (2/4)

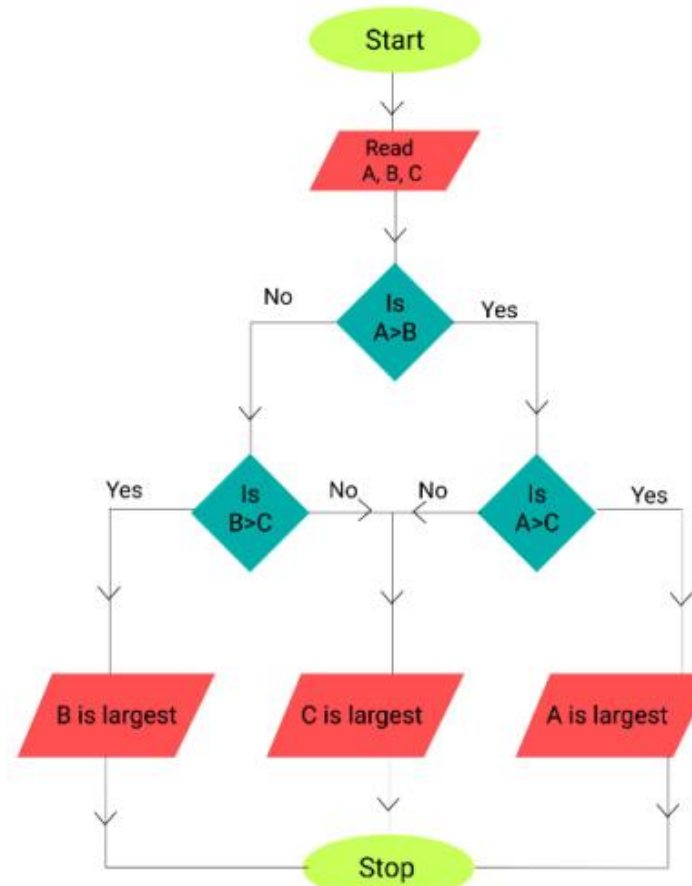
- **Flowline:** Represents the flow direction of a process. Each flowline connects two blocks.
- **Terminal:** Indicates the start or end of a diagram.
- **Process:** Represents a step in a process. This is the most common component of a flowchart.
- **Decision:** Depicts a decision step in a process. Typically, it is a yes/no or true/false question.
- **Input/Output:** Represents the input or output of data in a process.

Flowchart Symbols (3/4)

- Annotation: Provides additional information related to a step in the process.
- Predefined Process: Displays a named process that is defined elsewhere.
- On-page Connectors: Pair of on-page connectors are used to replace long flowlines on the diagram page.
- Off-page Connectors: Off-page connectors are used when the target is on a different page.

Flowchart Symbols (4/4)

- **Ex:** Draw a flowchart to find the largest of three numbers x , y and z .



Pseudo Code (1/2)

- Pseudocode is a formal language that helps programmers develop algorithms. Pseudocode often borrows the syntax of a specific language to represent the algorithms.
- Advantages: Easy to understand, Simulation capability, Flexibility, ...
- Disadvantages: Non-executable, Difficult complexity analysis, Misinterpretation potential,...

Pseudo Code (2/2)

- **Ex:** Using pseudocode to represent the algorithms for solving a linear equation $ax + b = 0$ (*a, b are real numbers*).

- Input: 2 real numbers a, b
- Output: Solution of the linear equation $ax + b = 0$

If $a = 0$ Then

Begin

 If $b = 0$ Then

 print "The equation has infinitely many solutions"

 Else

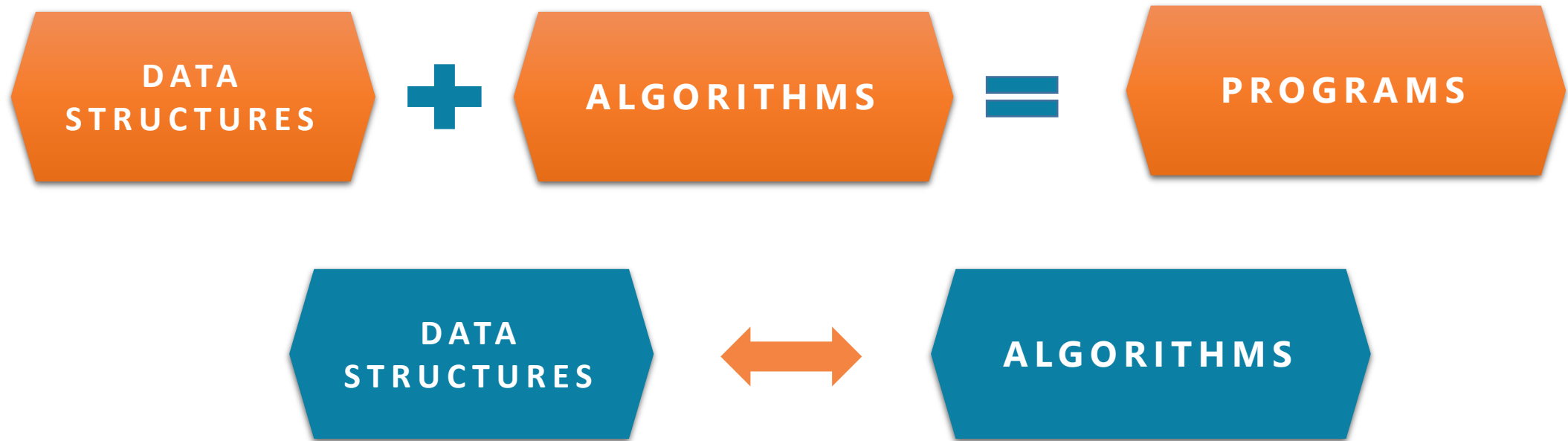
 print "The equation has no solution"

End

Else

 print "The solution of the equation is $x = -b/a$ "

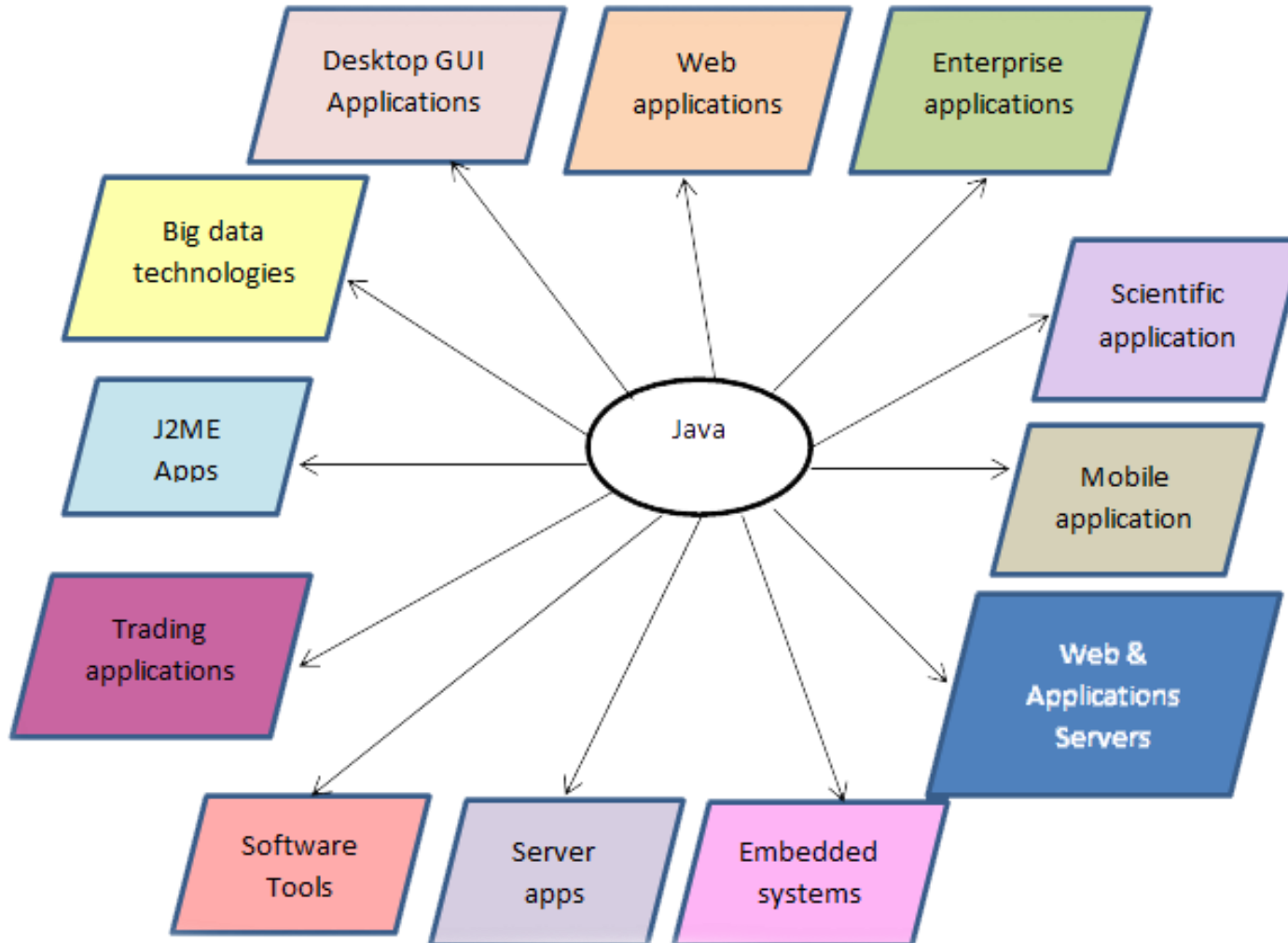
Computer Program



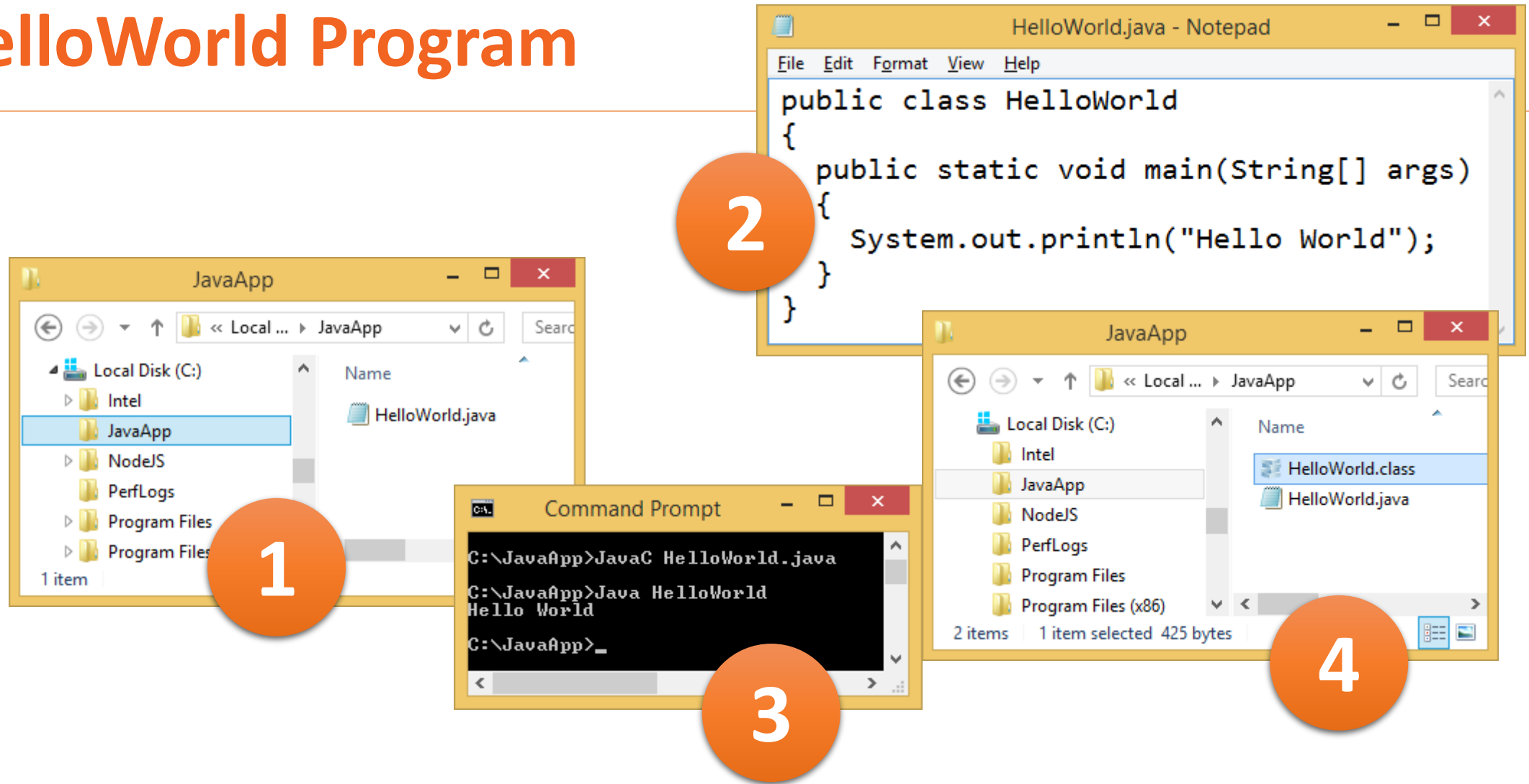
Introduction to Java

- Java: A versatile and widely-used object-oriented programming language
- Developed by James Gosling and team at Sun Microsystems
(*Now owned by Oracle Corporation*)
- Released in 1995

Applications of Java



HelloWorld Program



Structure of Java Program

Documentation Section
Package Statement
Import Statements
Interface Statements
Class Definitions
main method class { main method definition }

Java Hello Program Structure

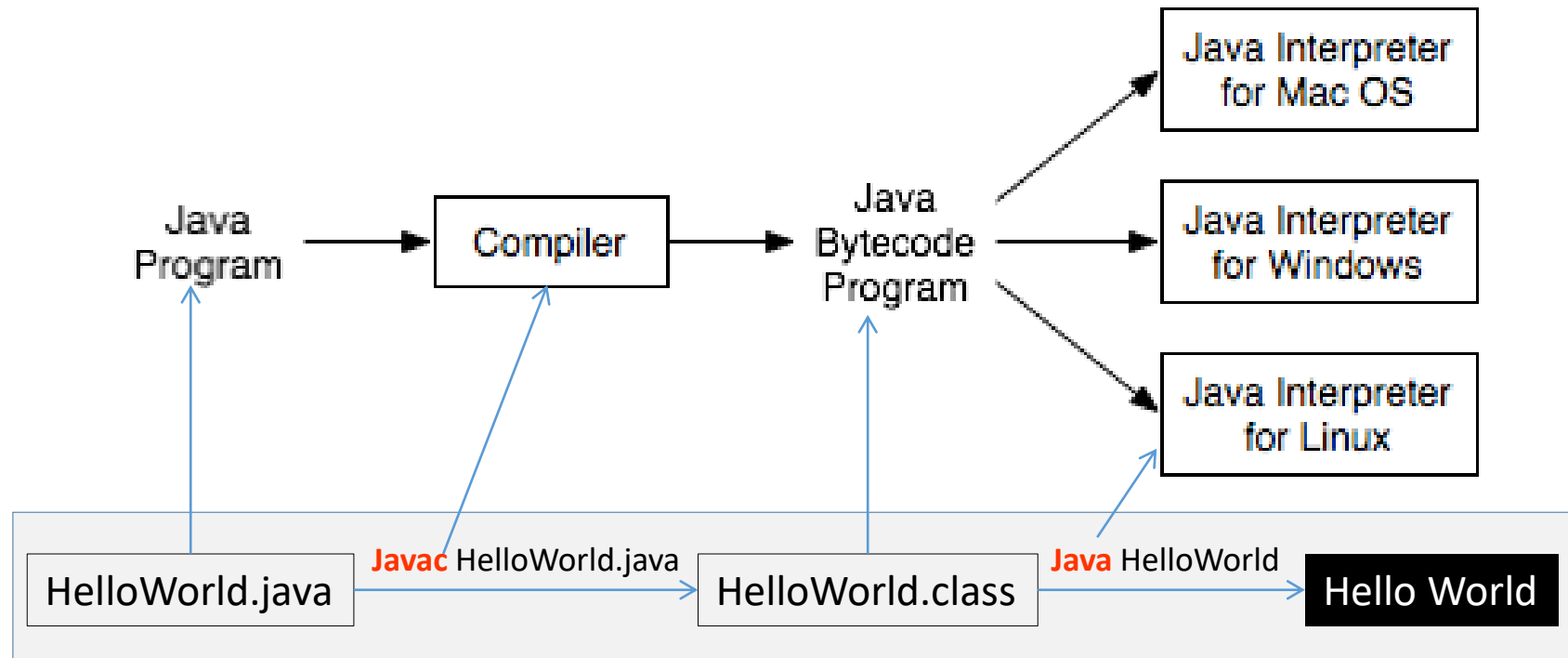
```
class HelloJava
{
    public static void main(String args[])
    {
        /*multiline
        comment*/
        System.out.println("HELLO JAVA");
        //single line comment
    }
}
```

class with name Hello

Accepts commandline arguments

Semicolon

Write Once, Run Anywhere



Java Variables (1/3)

- Variables in Java are used to store data in memory and enable programs to manipulate that data.
- Syntax for Variable Declaration in Java

`<data_type> <variable_name>;` *// Variable declaration syntax*

`int age;` *// Ex: declare an integer variable*

`<variable_name> = <value>;` *// Assign value to a variable*

`age = 25;` *// Ex: assign a value to the age variable*

`<data_type> <variable_name> = <value>;` *// Declare and assign a value in the same line*

`String name = "John";` *// Ex: declare a String variable and assign a value in the same line*

Java Variables (2/3)

// Example: Calculate the sum of two numbers

```
int a = 5; int b = 3; int sum = a + b;
```

```
System.out.println("The sum is: " + sum);
```

// Example: Use variables in if-else statements

```
int age = 18;
```

```
if (age >= 18) {
```

```
    System.out.println("You are eligible to vote");
```

```
} else {
```

```
    System.out.println("You are not eligible to vote");
```

```
}
```

// Example: Use variables in loops

```
int n = 5;
```

```
for (int i = 1; i <= n; i++) {
```

```
    System.out.println("Iteration " + i);
```

```
}
```

Java Variables (3/3)

The general rules for naming variables are:

- Use alphabetic characters, numbers, \$, or underscore (_). Do not start with a number and avoid using keywords
- Names are case-sensitive
- Ex:

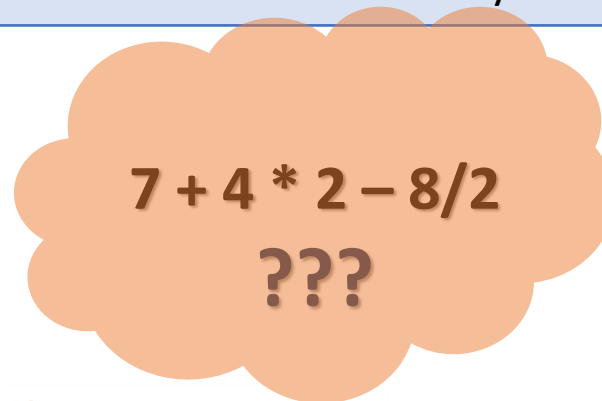
```
int age;  
String fullName;  
double accountBalance;  
boolean isStudent;
```

Arithmetic Operators

Operator	Name	Description	Example
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	$x \% y$
++	Increment	Increases the value of a variable by 1	$++x$
--	Decrement	Decreases the value of a variable by 1	$--x$

■ Order of precedence:

- *Multiplication and division;*
- *Addition and subtraction;*
- *Left to right.*



Output to the Screen (1/3)

- `System.out.print()`: *Prints without a newline character at the end.*
- `System.out.println()`: *Prints with a newline character at the end.*
- `System.out.printf()`: *Prints with formatted output using format specifiers.*
 - **%d**: integer
 - **%f**: floating-point number
 - ✓ By default, it uses 6 decimal places for floating-point numbers.
 - ✓ **%.3f**: formats the number with 3 decimal places.
 - **%s**: string

Output to the Screen (2/3)

1. Using System.out.print():

```
int age = 25;
```

```
System.out.print("My age is ");
```

```
System.out.print(age);
```

Output: My age is 25

2. Using System.out.println():

```
String name = "John";
```

```
System.out.println("Hello, " + name + "!");
```

```
System.out.println("Welcome to Java!");
```

Output: Hello, John! Welcome to Java!

Output to the Screen (3/3)

3. Using `System.out.printf()` with format specifiers:

```
double pi = 3.14159;
```

```
System.out.printf("The value of pi is %.2f%n", pi);
```

Output: The value of pi is 3.14

4. Combining output with multiple variables:

```
int apples = 5;
```

```
int oranges = 3;
```

```
System.out.println("I have " + apples + " apples and " + oranges + " oranges.");
```

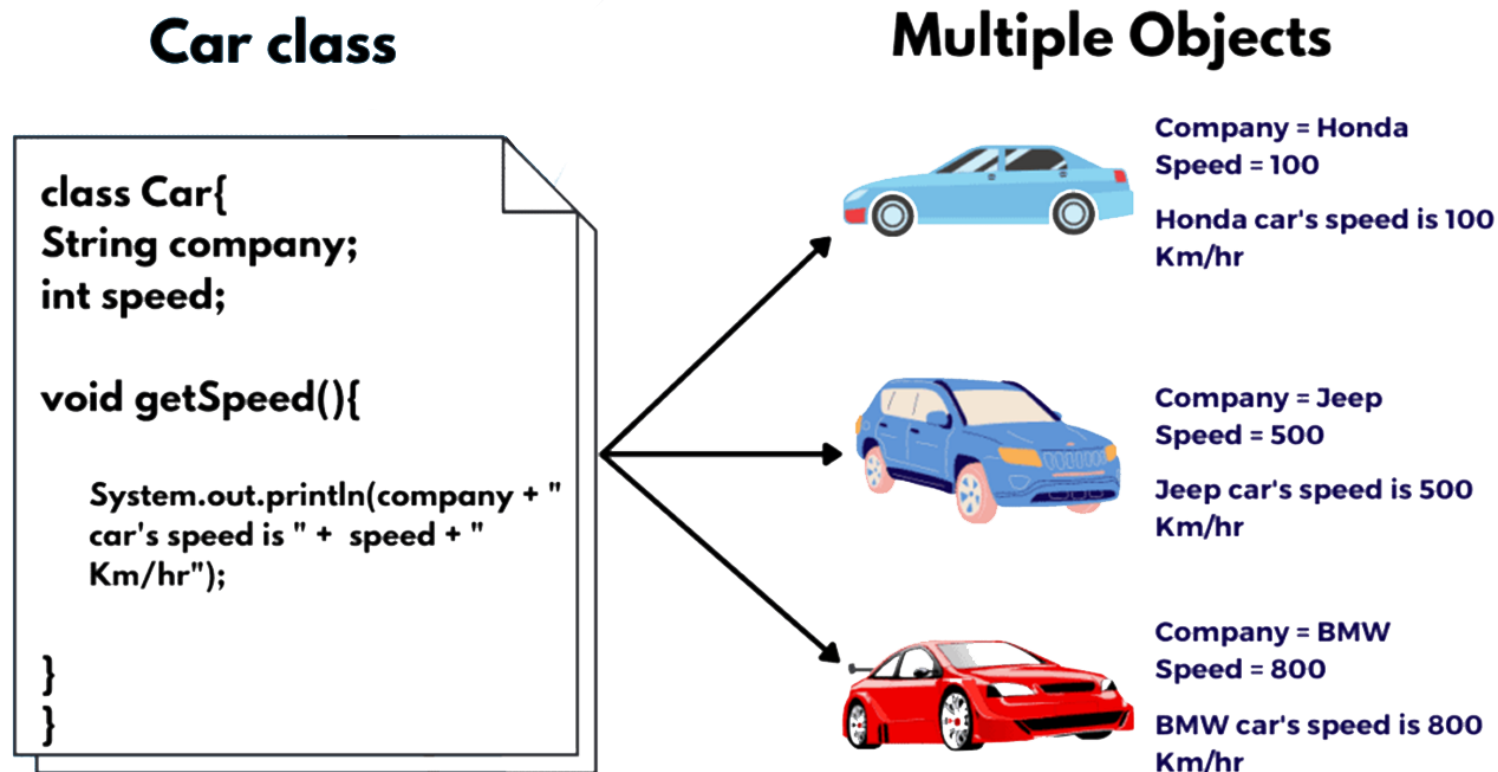
Output: I have 5 apples and 3 oranges.

Input From the Keyboard

- java.util.Scanner allows simple input of data from the keyboard
- Create a Scanner object:
 - Scanner **scanner** = new Scanner(System.in)
- Commonly used methods:
 - scanner.**nextLine()**
 - ✓ Receive a line of input from the keyboard
 - scanner.**nextInt()**
 - ✓ Receive an integer input from the keyboard
 - scanner.**nextDouble()**
 - ✓ Receive a decimal input from the keyboard

Classes and Objects

- A class is a template used to describe objects of the same type
- It consists of attributes and methods



Fields and Methods

■ Fields

- Company
- Model
- Year
- Color

Noun

■ Methods

- Start()
- Stop()
- Brake()
- Turn on Wiper()

Verb



Class Definition

```
class <<ClassName>>
```

```
{
```

```
    <<type>> <<field1>>;
```

```
    ...
```

```
    <<type>> <<fieldN>>;
```

```
    <<type>> <<method1>>([parameters]) {  
        // body of method
```

```
    }
```

```
    ...
```

```
    <<type>> <<methodN>>([parameters]) {  
        // body of method
```

```
    }
```

```
}
```

Example

```
public class Car {  
    // Fields  
    String brand;  
    String color;  
    int year;  
    // Methods  
    public void start() {  
        System.out.println("The car is starting");  
    }  
    public void stop() {  
        System.out.println("The car is stoping");  
    }  
    public void brake() {  
        System.out.println("The car is braking");  
    }  
}
```

Create a Object

- The following code snippet uses the **Car** class to create an **myCar** and then calls the class's methods

```
public static void main(String[] args) {  
    Car myCar = new Car();  
    // Creating an object of the Car class  
    myCar.brand = "Toyota";  
    myCar.color = "Red";  
    myCar.year = 2022;  
    myCar.start();  
    myCar.stop();  
    myCar.brake();  
}
```

Note:

- The "new" operator is used to create an object.
- The variable "myCar" holds a reference to the object.
- Use the dot (.) operator to access members of the class (fields and methods).

Create a Method

- A method is a code module that performs a specific task
- A method can have one or more parameters
- A method can have a return type or void (no return value)
- Syntax:

```
<<return type>> <<Method name>> ([parameters])  
{  
    // Method body  
}
```

Constructor (1/2)

- A constructor is a special method used to create objects
- Characteristics of a constructor:
 - It has the same name as the class
 - It does not return a value

- Example:

Class

```
public class Car {  
    String brand, color; int year;  
    // Parameterized constructor  
    public Car(String brand, String color, int year) {  
        this.brand = brand;  
        this.color = color;  
        this.year = year;  
    }  
}
```

Object

```
Car myCar1 = new Car("Toyota", "Red", 2022);  
Car myCar2 = new Car("Honda", "Blue", 2023);
```

Constructor (2/2)

- Within a class, multiple parameterized constructors can be defined, each providing a different way to create objects.
- If a constructor is not declared, Java automatically provides a default constructor (*without parameters*).

```
public class Car {  
    String brand, color; int year;  
    // Constructor  
    public Car() {  
        // Default constructor  
    }  
    // Parameterized constructor  
    public Car(String brand, String color, int year) {  
        this.brand = brand;  
        this.color = color;  
        this.year = year;  
    }  
}
```

`Car myCar1 = new Car();` // Using the default constructor

`Car myCar2 = new Car("Honda", "Blue", 2023);` // Using the parameterized constructor

This Keyword

- This is used to represent the current object. It is used within a class to refer to the members of the class (*fields and methods*).
- Use 'this.field' to differentiate the field from local variables or method parameters.

Field

```
public class MyClass {  
    int field;  
    public void method( int field) {  
        this.field = field;  
    }  
}
```

Parameter

Overloading

- In a class, there can be multiple methods with the same name but different parameters (*type, quantity, and order*)

```
public class MyClass{  
    void method(){...}  
    void method(int x){...}  
    void method(float x){...}  
    void method(int x, double y){...}  
}
```

Package (1/3)

- A package in Java is used to group related classes
- Think of it as a folder in a file directory
- We use packages to avoid name conflicts, and to write a better maintainable code
- Packages are divided into two categories:
 - Built-in Packages (*packages from the Java API*)
 - User-defined Packages (*create your own packages*)

Package (2/3)

- In Java, there are numerous packages categorized based on their functionalities
 - java.util: contains utility classes
 - java.io: contains input/output classes
 - java.lang: contains commonly used classes...

- Syntax:

```
import package.name.Class; // Import a single class
import package.name.*; // Import the whole package
```

- Import a class Scanner:

```
import java.util.Scanner;
```

Package (3/3)

■ Import a Package:

```
import java.util.*; // We used the Scanner class from the java.util package
```

■ User-defined Packages:

```
package mypack;  
class MyPackageClass {  
    public static void main(String[] args) {  
        System.out.println("This is my package!");  
    }  
}
```

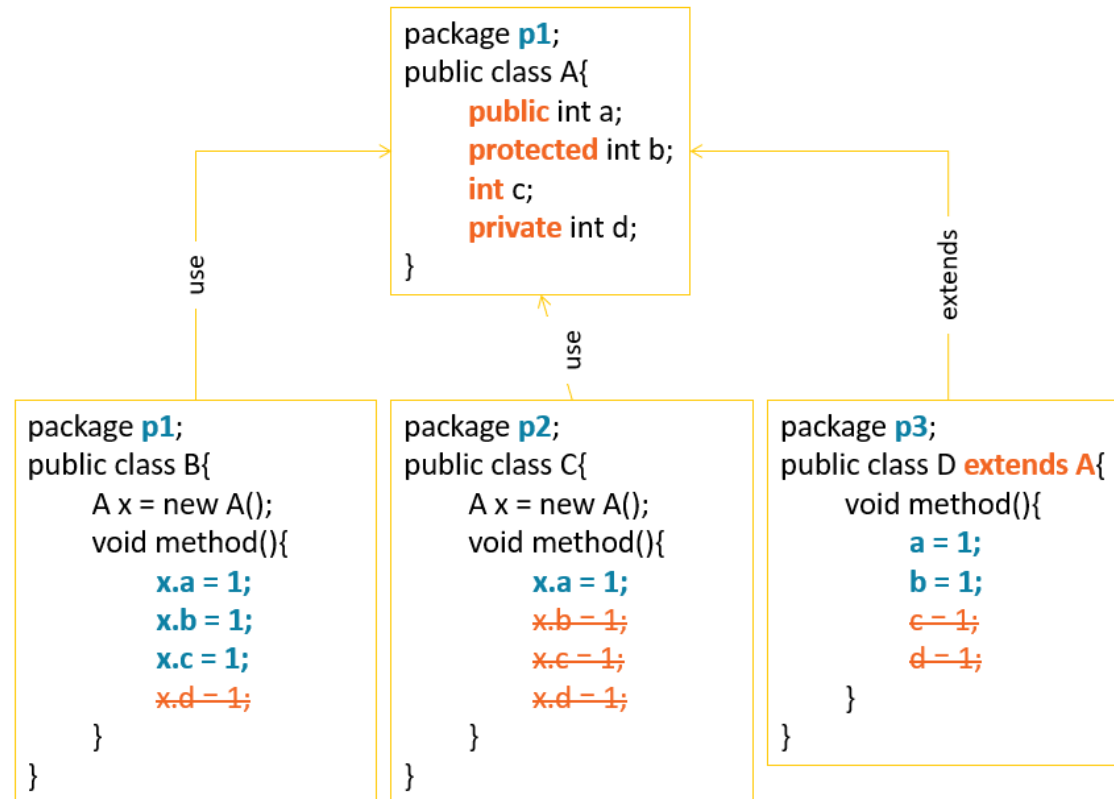
Access Modifiers (1/2)

- Access specification is used to define the level of access granted to members of a class
- In Java, there are four different access specifications:
 - **private**: Only accessible within the class itself
 - **public**: Completely accessible from anywhere
 - **{default}**:
 - + Public within the same package
 - + Private when accessed from a different package
 - **protected**: Same as {default}, but allows inheritance even if the subclass is in a different package

Access Modifiers (2/2)

- The level of encapsulation increases in the direction of the arrow

public → **protected** → **{default}** → **private**



Encapsulation (1/2)

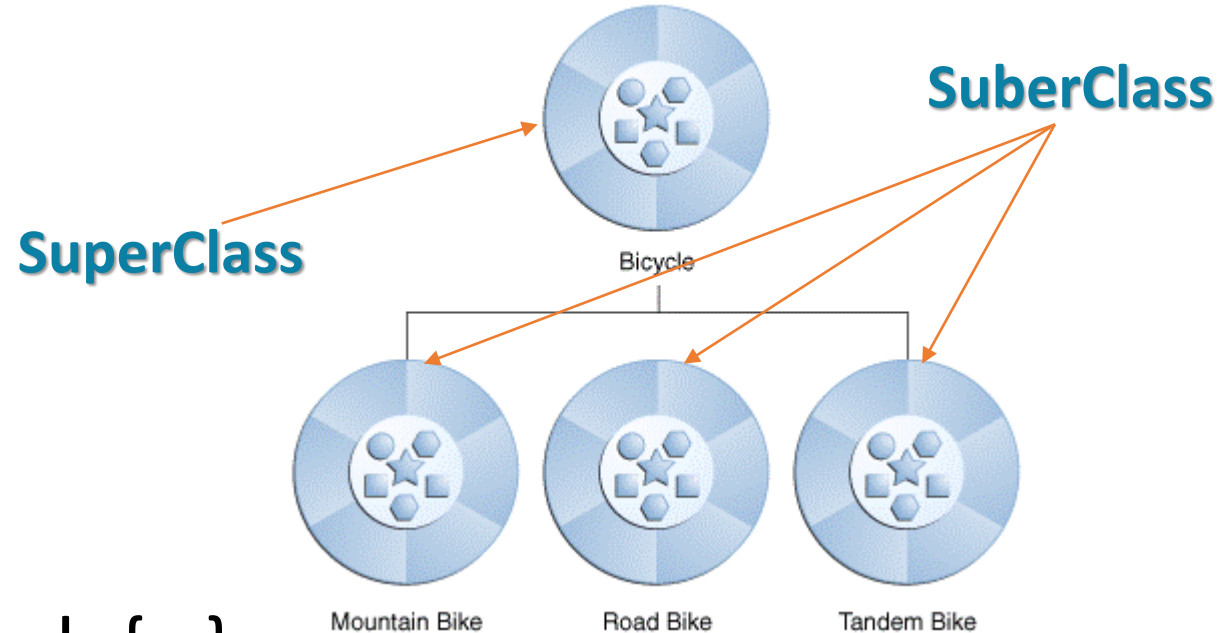
The meaning of Encapsulation, is to make sure that "*sensitive*" data is hidden from users. To achieve this, you must:

- Declare class variables/attributes as private
- Provide public get and set methods to access and update the value of a private variable

Encapsulation (2/2)

```
public class Person {  
    private String name; // private = restricted access  
    // Getter  
    public String getName() {  
        return name;  
    }  
    // Setter  
    public void setName(String newName) {  
        this.name = newName;  
    }  
}
```

Inheritance (1/3)



```
class Bicycle{...}  
class MountainBike extends Bicycle{...}  
class RoadBike extends Bicycle{...}  
class TandemBike extends Bicycle{...}
```

Inheritance (2/3)

- The purpose of inheritance is reuse
- A child class is allowed to inherit the fields and methods of the parent class
- A child class is allowed to inherit the public or protected properties of the parent class
- A child class is also allowed to inherit the default properties of the parent class if the child class and the parent class are defined in the same package
- A child class cannot access the private members of the parent class

Inheritance (3/3)

```
package fpi.hn;
public class Employee {
    public String fullName;
    protected double salary;
    public Employee(String fullName, double salary) {
        this.fullName = fullName;
        this.salary = salary;
    }
    public void output() {
        System.out.println("Fullname: " + fullName);
        System.out.println("Salary: " + salary);
    }
    private double incomeTax() {
        return 0.0;
    }
}
```


```
package fpi.ct;
public class Manager extends Employee {
    public double responsibility;
    public Manager(String fullName, double salary,
        double responsibility) {
        super(fullName, salary);
        this.responsibility = responsibility;
    }
    public void output() {
        super.out();
        System.out.println("Responsibility: " + responsibility);
    }
}
```

Use Super

- Accessing the members of the superclass using the super
- Keyword Super can be used to call the constructor of the parent class

```
public class Parent{  
    public String name;  
    public void method(){}  
}
```

```
public class Child extends Parent{  
    public String name;  
    public void method(){  
        this.name = super.name;  
        super.method()  
    }  
}
```

A blue arrow points from the 'Child' class box to the 'Parent' class box, indicating that 'Child' inherits from 'Parent'.

Overriding

- Overriding occurs when both the subclass and the superclass have a method with the same signature



- Both the Parent and Child classes have a method() with the same syntax, so the method() in the Child class will override the method() in the Parent class

```
Parent o = new Child();  
o.method()
```

Even though the variable 'o' is of type Parent, when 'o.method()' is invoked, the method() of the Child class will be executed because it has been overridden.

Abstract Class (1/3)

- An abstract class is a class that has undefined behaviors
 - **Ex1:** *When dealing with shapes, we know that every shape has an area and perimeter, but the specific calculation methods cannot be determined until we have a specific shape such as a rectangle, circle, or triangle*
 - **Ex2:** *When it comes to students, they certainly have an average score, but the specific calculation method cannot be determined until we know the major they are studying and the specific subjects and formulas used to calculate the score*
- Therefore, the Shape and Student classes are abstract classes because the methods for calculating perimeter, area, and average score have not been implemented yet

Abstract Class (2/3)

```
abstract public class MyClass{  
    abstract public type MyMethod();  
}
```

Use the keyword
"abstract" to
define an abstract
class and abstract
methods

```
abstract public class Student{  
    abstract public double get getAverage();  
}
```

```
abstract public class Shape{  
    abstract public double getgetPerimeter();  
    abstract public double getArea();  
}
```


Abstract Class (3/3)

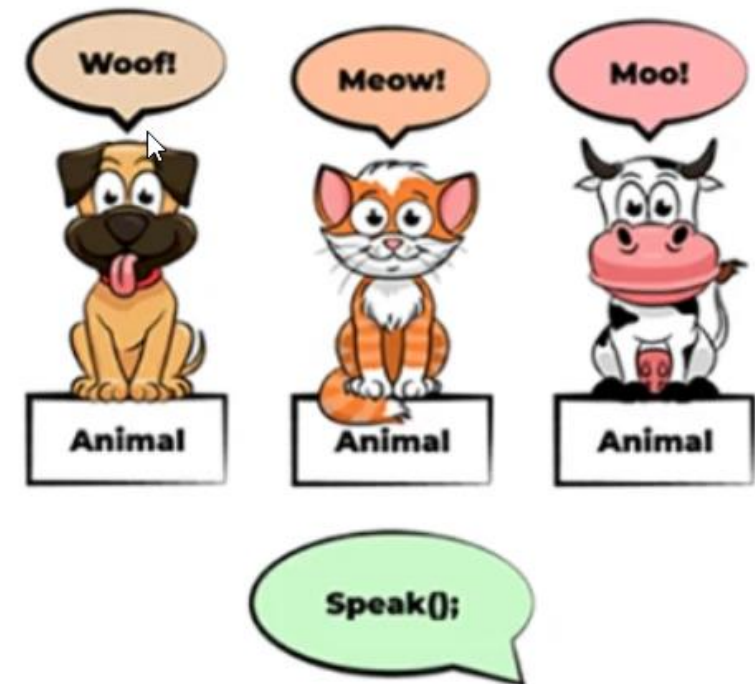
```
abstract public class Student{  
    public String fullName;  
    abstract public double getAverage();  
}
```

```
public class StudentIT extends Student{  
    public double java;  
    public double css;  
    @Override  
    public double getAverage(){  
        return (2 * java + css)/3;  
    }  
}
```

```
public class StudentBiz extends Student {  
    public double accountan;  
    public double marketting;  
    public double sale;  
    @Override  
    public double getAverage(){  
        return  
            (accountan + marketting + sale)/3;  
    }  
}
```

Polymorphism (1/2)

- Overriding achieves polymorphism in object-oriented programming by allowing a behavior to be manifested in different forms
- Invoking an overridden method is determined at runtime, not at compile time



Polymorphism (2/2)

```
abstract public class Animal{  
    abstract public void speak();  
}
```

```
Animal dog = new Dog();  
Animal cat = new Cat();  
Animal cow = new Cow();
```

```
dog.speak();  
cat.speak();  
cow.speak();
```

```
public class Dog extends Animal{  
    public void speak(){  
        System.out.println("Woof");  
    }  
}
```

```
public class Cat extends Animal{  
    public void speak(){  
        System.out.println("Meow");  
    }  
}
```

```
public class Cow extends Animal{  
    public void speak(){  
        System.out.println("Moo");  
    }  
}
```

Interface

- In an interface, only abstract methods and final variables are present
- When a class implements an interface, it must write (*override*) the methods in the interface
- An interface can be either public or default
- An interface can be inherited
- An interface can be implemented by multiple classes, and a class can implement multiple interfaces

Declare Interface

```
package mypackage;  
  
interface MyInterface{  
    void myMethod1();  
    void myMethod2();  
}
```

Implement Interface

```
package mypackage;
public class MyClass implements MyInterface{
    public void myMethod1(){
        System.out.println("Override my method 1");
    }
    public void myMethod2(){
        System.out.println("Override my method 2");
    }
    void mymethod3(){
        System.out.println("My method 3");
    }
}
```

Use Interface

```
public static void main (String a[]){  
    MyClass myObject1 = new MyClass();  
    obj1.mymethod1();  
    obj1.mymethod2();  
    obj1.mymethod3();  
  
    MyInterface myObject2 = new MyClass();  
    obj2.mymethod1();  
    obj2.mymethod2();  
    obj2.mymethod3();// error;  
  
}
```

Implement Multiple Interfaces

```
public interface MyInter1{
    void meth1();
    void meth2();
}
public interface MyInter2{
    void meth3();
}
public class MyClass implements MyInter1, MyInter2 {
    public void meth1(){
        System.out.println("Implements method 1");
    }
    public void meth2(){
        System.out.println("Implements method 2");
    }
    public void meth3(){
        System.out.println("Implements method 3");
    }
}
```


Inner Class

- An inner class is a class declared inside another class
- There are two types: static inner class and regular inner class
- The inner class can only be defined within the outermost class and can access the members of the enclosing class

```
public class MyClass{  
    static public class MyInnerStaticClass{}  
    public class MyInnerClass{}  
}
```

Use Inner Class

```
MyClass.MyInnerStaticClass x = new MyClass.MyInnerStaticClass();  
MyClass.MyInnerClass y = new MyClass().new MyInnerClass();
```

Static (1/2)

- The keyword “static” is used to define blocks and static members (*inner classes, methods, fields*)

```
public class MyClass{  
    static public int X;  
    static{  
        X+=100;  
    }  
    static public void method(){  
        X+=200;  
    }  
    static class MyInnerClass{  
    }  
}
```

MyClass.X = 700;
MyClass.method()

Static (2/2)

```
public class MyClass{  
    static public int X = 100;  
    static{  
        X+=100;  
    }  
    static public void method(){  
        X+=200;  
    }  
}
```

```
MyClass o = new MyClass();  
o.X += 300;  
MyClass.X += 500;  
MyClass.method()
```

MyClass.X, o.X
How much is it?

Final

- In Java, there are 3 types of constants:
 - A constant class is a class that does not allow inheritance
 - A constant method is a method that does not allow overriding
 - A constant variable is a variable that does not allow changing its value
- The keyword “final” is used to define constants

```
final public class MyFinalClass{...}
```

```
public class MyClass{  
    final public double PI = 3.14  
    final public void method(){...}  
}
```

Summary

- Understand the concept of data structures and algorithms
- Gain an understanding of the structure of a simple Java program
- Write a HelloWorld program
- Class, Object, Fields and Methods
- Inheritance
- Abstract & Interface
- Static, final and Inner Class

THANK
you