

# Java

## Лекція 1

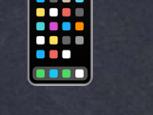
# Зміст

- ⦿ Вступ
- ⦿ Класи та об'єкти
- ⦿ Створення об'єктів
- ⦿ Пакети
- ⦿ Наслідування
- ⦿ Модифікатори доступу

# Вступ

- Java — об'єктно-орієнтована мова програмування.
- Створена у 1995 р. компанією Sun Microsystems, тепер належить Oracle.
- Використання: веб, мобільні, серверні, десктоп.
- Принцип: “**Write once – run anywhere**”.

# Чому Java?

-  Портативність – програми працюють на будь-якій платформі з JVM.
-  Об'єктно-орієнтований підхід – зручна модель для великих проектів.
-  Велика стандартна бібліотека – готові класи для роботи з файлами, мережею, колекціями тощо.
-  Масштабованість та надійність – використовується у великих корпоративних системах (банки, e-commerce).
-  Популярність і спільнота – мільйони розробників та тисячі бібліотек.
-  Універсальність застосування – Android, веб-сервіси, корпоративні додатки, фінансові системи.

# Java платформа

## 1. JDK (Java Development Kit)

Повний набір для розробки

Містить JRE + інструменти розробки

- **javac** - компілятор Java
- **javadoc** - генератор документації
- **debugger** - відладчик
- **jar** - архіватор
- **javap** - дизасемблер

## 2. JRE (Java Runtime Environment)

Середовище для виконання

Містить JVM + стандартні бібліотеки Java

- **Java API** - стандартні бібліотеки (java.lang, java.util, java.io)
- **Deployment технології** - Java Web Start, Java Plug-in
- **UI Toolkits** - AWT, Swing
- **Integration Libraries** - JDBC, JNDI, RMI

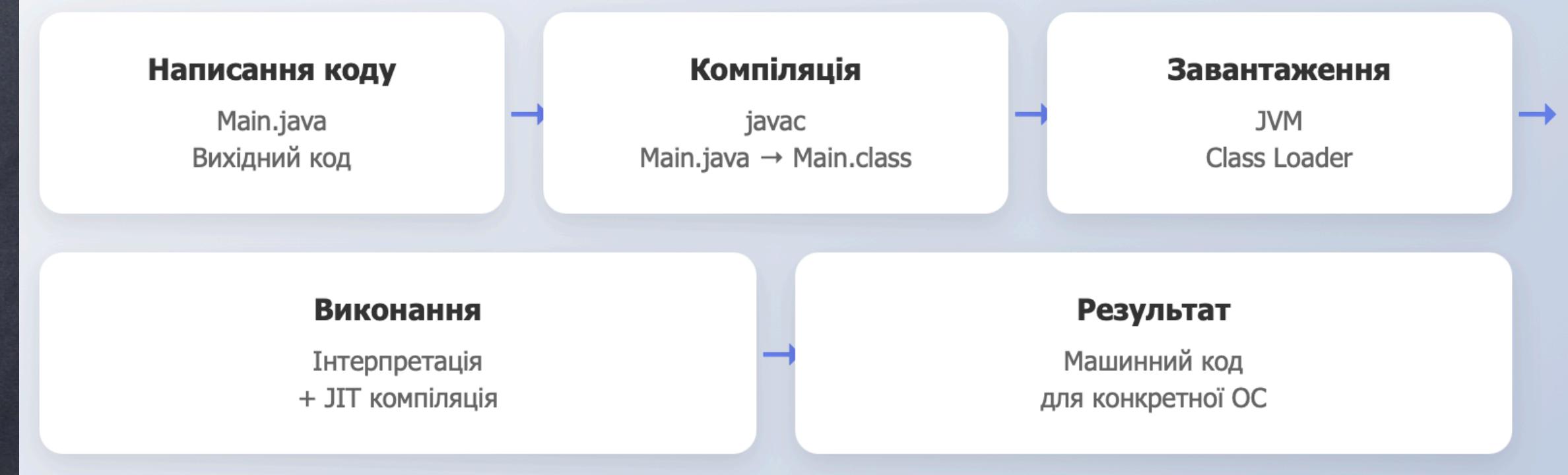
## 3. JVM (Java Virtual Machine)

Власне віртуальна машина

Компілює та інтерпретує байт-код у машинний код

- **Class Loader** - завантаження та верифікація класів
- **Memory Management** - керування пам'яттю (Heap, Stack, Method Area)
- **Execution Engine** - інтерпретатор + JIT компілятор
- **Garbage Collector** - автоматичне прибирання пам'яті

### Процес виконання Java-програми



# Де використовується Java?

## Мобільні застосунки

- **Android**: значна частина застосунків і бібліотек написані на Java.
- **Kotlin** активно розвивається, але Java залишається основою багатьох **Android**-проектів, адже **Kotlin** працює на **JVM** і сумісний із **Java**-кодом.

## Веб-додатки та сервіси

- **Spring Framework** - провідний фреймворк для бекенду.
- Використовується у **Netflix**, **Amazon**, **LinkedIn**, **eBay**.

## Корпоративні системи

- Банківські та фінансові рішення (**JPMorgan**, **Goldman Sachs**).
- **ERP/CRM** для великих компаній.

## Хмарні сервіси

- Java підтримується на **AWS**, **Google Cloud**, **Microsoft Azure**.

## Інші сфери

- Ігри (**Minecraft**)
- Будовані системи (**IoT**, смарт-карти).

# Синтаксис java

- Java створювалася як мова без багатьох складних і небезпечних моментів C++, але з легким для розуміння синтаксисом.
- Базові конструкції, знайомі з C++:
- Умовні оператори: if, else, switch
- Цикли: for, while, do-while
- Оператори: арифметичні + - \* / %, логічні & & || !, порівняння == != < > <= >=
- Масиви: створення і доступ через індекси int[] arr = new int[10];
- Методи/функції: визначення і виклик методів з параметрами та поверненням значення
- Класи та об'єкти: class, new, this

# Завдання № (повторення синтаксису)

- <https://www.tutorialspoint.com/java/index.htm>
- <https://www.geeksforgeeks.org/java/java/>
- <https://classroom.github.com/a/P7QL0s71>
- <https://github.com/romanienko-natali/java-2025-syntax>

# Класи та об'єкти

- Клас – шаблон для створення об'єктів.
- Об'єкт – екземпляр класу.

```
class Car {  
    String model;  
    void drive() {  
        System.out.println(model + " is driving...");  
    }  
}
```

# Реалізація принципів ООП

## Інкапсуляція (Encapsulation)

- Приховування внутрішнього стану об'єкта
- Поля робляться **private**, доступ до них – через **public** методи
- Забезпечує контролюваний доступ до даних

## Наслідування (Inheritance)

- Створення нових класів на основі існуючих (**extends**)
- Дозволяє повторно використовувати код та розширювати функціональність

## Поліморфізм (Polymorphism)

- Однаковий інтерфейс, різна реалізація
- Використання спільного типу (наприклад, базового класу) для об'єктів різних підкласів
- Ключовий у поєднанні з наслідуванням (перевизначення методів)

## Абстракція (Abstraction)

- Виділення суттєвих характеристик без деталей реалізації
- Досягається через абстрактні класи та інтерфейси
- Дозволяє проектувати систему на рівні концепцій

# Класи

- Поля (*Fields / Attributes*)

- Змінні, що описують стан об'єкта.
- Можуть бути приватними (*private*), захищеними (*protected*), публічними (*public*), доступними в пакеті (без модифікатора доступу).
- Можуть бути статичними (*static*) – спільними для всіх об'єктів класу.

- Методи (*Methods*)

- Визначають поведінку класу.
- Можуть бути статичними (*static*) – викликаються без створення об'єкта.

- Можуть мати модифікатори доступу (*private*, *protected*, *public*, *default*).

- Конструктори (*Constructors*)

- Спеціальні методи для ініціалізації об'єкта при створенні.
- Можуть приймати параметри.
- Можуть бути перевантаженими (один клас – декілька конструкторів з різними параметрами).
- Можуть мати модифікатори доступу (*private*, *protected*, *public*, *default*).

<https://replit.com/Languages/java10>

## Code 1

<https://www.jdoodle.com/online-java-compiler>

```
public class Car {  
    }  
  
    // Поля  
  
    private String model; // Методы  
  
    public int year;  
  
    static int totalCars = 0; // статичне поле  
  
    public void drive() { // нестатичний метод  
        System.out.println(model + " is driving.");  
    }  
  
    // Конструктор  
  
    public Car(String model, int year) {  
        this.model = model;  
  
        this.year = year;  
  
        totalCars++;  
    }  
  
    public static int getTotalCars() { // статичний метод  
        return totalCars;  
    }  
}
```

# Static

- Поле або метод належить класу, а не конкретному об'єкту.
- Не потрібно створювати об'єкт, щоб звернутися до `static` члена.
- Статичні методи не можуть безпосередньо звертатися до нестатичних полів або методів класу. Причина: статичний метод належить класу, а нестатичні поля/методи належать конкретному об'єкту.
- Зручно для констант, утилітних методів і лічильників об'єктів.

# Приклад використання static

## Code2

```
class Counter {  
    int instanceVar;      // нестатичне поле  
    static int staticVar = 0; // статичне поле спільне для всіх об'єктів  
  
    Counter(int value) {  
        this.instanceVar = value;  
        staticVar++;      // збільшуємо статичне поле  
    }  
  
    static void showStatic() {  
        System.out.println("Static count: " + staticVar);  
  
        // System.out.println(instanceVar); // ✗ помилка! Не можна  
        // звертатися до нестатичного поля без об'єкта  
  
        // Правильний доступ через об'єкт  
        Counter temp = new Counter(0);  
        System.out.println("Access instanceVar via object: " +  
temp.instanceVar);  
    }  
  
    void showInstance() {  
        System.out.println("Instance value: " + instanceVar);  
        System.out.println("Static count from instance: " + staticVar);  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Counter c1 = new Counter(10);  
        Counter c2 = new Counter(20);  
  
        c1.showInstance();  
        c2.showInstance();  
  
        // Виклик статичного методу без об'єкта  
        Counter.showStatic();  
    }  
}
```

# Конструктор

Спеціальний метод для ініціалізації об'єкта.

Має таку ж назву, як і клас.

```
class Car {  
    String model;  
    Car(String m) {  
        model = m;  
    }  
}
```

# Створення об'єктів (1)

- ◆ 1. Створення через конструктор

```
public class Car {  
    String model;  
  
    public Car(String model) {  
        this.model = model;  
    }  
}
```

```
public class Main {  
    public static void main(String[]  
args) {  
        Car car1 = new Car("Toyota");  
        System.out.println(car1.model);  
    }  
}
```

# Створення об'єктів (2)

## ◆ 2. Створення через статичний метод (Factory Method)

Використовується, коли конструктор приватний.

Переваги: контроль створення об'єктів, можливість кешування, обмеження кількості екземплярів, застосовується у патернах проектування (Singleton, Factory, Builder).

```
public class Car {  
    private String model;  
  
    // Приватний конструктор  
    private Car(String model) {  
        this.model = model;  
    }  
  
    // Статичний метод для створення об'єкта  
    public static Car createCar(String model) {
```

```
        return new Car(model);  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Car car2 = Car.createCar("BMW"); // Виклик  
        // статичного методу  
        System.out.println(car2.model);  
    }  
}
```

# Package, imports

- `package` – групування класів (аналог папки).
- `import` – дозволяє підключати класи.

```
package vehicles;
```

```
public class Car { }
```

```
import vehicles.Car;
```

# Source root

## ☛ Папка, з якої починається структура пакетів

```
/Users/nataliia/IdeaProjects/java-2025-syntax    ← project root
  └ src/main/java
    └ ua/
      └ university/
        └ Main.java
        └ Main.class
```

### ◆ 1. Компіляція

- Можна перебувати будь-де, головне вказати шлях до `.java` файлу:

```
cd
/Users/nataliia/IdeaProjects/java-2025-syntax
javac
src/main/java/ua/university/Main.java
```

- `.class` файл з'явиться у тій же папці (ієрархія пакетів збережеться):

```
src/main/java/ua/university/Main.class
```

### ◆ 2. Запуск

#### Варіант А: перебуваємо у source root

```
cd
src/main/java
java
ua.university.Main
```

JVM бачить `ua/university/Main.class` відносно source root.

#### Варіант В: перебуваємо у project root

```
cd
/Users/nataliia/IdeaProjects/java-2025-syntax
java
-cp src/main/java ua.university.Main
```

JVM шукає клас у папці, зазначеній через `-cp` (classpath).

# Наслідування

```
class Product {  
    String name;  
    double price;  
  
    Product(String name, double price) {  
        this.name = name;  
        this.price = price;  
    }  
  
    void showInfo() {  
        System.out.println(name + " коштує " + price + " грн");  
    }  
  
class Electronics extends Product {  
    int warrantyMonths;  
  
    Electronics(String name, double price, int warrantyMonths) {  
        super(name, price);  
        this.warrantyMonths = warrantyMonths;  
    }  
  
    @Override  
    void showInfo() {
```

```
        System.out.println(name + " (електроніка) коштує " + price +  
        " грн, гарантія: " + warrantyMonths + " місяців");  
    }  
  
    }  
  
class Food extends Product {  
    String expirationDate;  
  
    Food(String name, double price, String expirationDate) {  
        super(name, price);  
        this.expirationDate = expirationDate;  
    }  
  
    @Override  
    void showInfo() {  
        System.out.println(name + " (їжа) коштує " + price +  
        " грн, придатна до: " + expirationDate);  
    }  
}
```

# Модифікатори доступу

Модифікатор	Клас	Пакет	Підклас	Весь код
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

```
package animals;

public class Animal {
    private String privateName = "Private Animal";
    String defaultName = "Default Animal";
    protected String protectedName = "Protected Animal";
    public String publicName = "Public Animal";

    private void privateMethod() {
        System.out.println("Private method");
    }

    void defaultMethod() {
        System.out.println("Default method");
    }

    protected void protectedMethod() {
        System.out.println("Protected method");
    }

    public void publicMethod() {
        System.out.println("Public method");
    }

    public void showNames() {
        System.out.println("Private: " + privateName);
        System.out.println("Default: " + defaultName);
        System.out.println("Protected: " + protectedName);
        System.out.println("Public: " + publicName);
    }
}
```

```
package animals;

public class Dog extends Animal {
    public void showAccess() {
        // privateName is not accessible
        // defaultName is accessible because we are in the same package
        System.out.println(defaultName);
        System.out.println(protectedName);
        System.out.println(publicName);

        // privateMethod(); // not accessible
        defaultMethod();
        protectedMethod();
        publicMethod();
    }
}
```

```
package zoo;

import animals.Animal;

public class ZooKeeper extends Animal {
    public void showAccess() {
        // defaultName is NOT accessible in another package
        // privateName is not accessible
        // protectedName is accessible because we are a subclass
        System.out.println(protectedName);
        System.out.println(publicName);

        // defaultMethod(); // not accessible
        protectedMethod();
        publicMethod();

        public void tryAccessAnimal() {
            Animal a = new Animal();
            // a.protectedName; // NOT accessible via object reference from another package
            // a.publicName; // accessible
            System.out.println(a.publicName);
        }
}
```

```
package ua.university;

import animals.Animal;
import animals.Dog;
import zoo.ZooKeeper;

public class Main {
    public static void main(String[] args) {
        System.out.println("Project works");

        Animal a = new Animal();
        Dog d = new Dog();
        ZooKeeper z = new ZooKeeper();

        System.out.println("Animal access:");
        a.showNames();

        System.out.println("\nDog access:");
        d.showAccess();

        System.out.println("\nZooKeeper access:");
        z.showAccess();
        z.tryAccessAnimal();
    }
}
```

# Практичні рекомендації (доступ)

## 1 Поля / змінні

- Завжди **private** за замовчуванням.
- Винятки:
  - Якщо плануємо доступ у підкласах → можна **protected**.
  - Якщо створюємо бібліотеку і очікуємо наслідування → теж **protected**.
- Мета: інкапсуляція → контроль доступу до внутрішнього стану об'єкта.

# Практичні рекомендації (доступ)

## 2 Геттери та сеттери

Робимо **public** лише якщо дійсно потрібно.

```
private int age;
```

```
public int getAge() { return age; }
```

```
public void setAge(int age) { this.age = age; }
```

Уникаємо непотрібних сеттерів, щоб об'єкти залишалися контролюваними або незмінними.

# Практичні рекомендації (доступ)

## 3 Default / package-private

Використовуємо без модифікатора для членів/класів, які повинні бути видимі тільки в пакеті.

Мета:

Інкапсуляція пакету → приховуємо внутрішню реалізацію від зовнішнього світу.

Добре для допоміжних класів, внутрішньої логіки чи утиліт, не призначених для публічного використання.

# Практичні рекомендації (доступ)

4 `protected`

Використовуємо для класів, що планується наслідувати.

Уникаємо `protected` для полів, якщо підклас не потребує прямого доступу.

5 `public`

Використовуємо тільки для:

API, яке повинно бути доступне скрізь.

Методів, які призначені для використання поза пакетом.

Поля не робимо `public`; краще через геттери/сеттери.

# Практичні рекомендації (доступ)

## 6 Загальні рекомендації

Мінімізуємо доступ → починаємо з **private**.

Відкриваємо доступ тільки коли необхідно.

Інкапсуляція допомагає з підтримкою, рефакторингом і відлагодженням.

Запитання, яке варто ставити: “Хто дійсно потребує доступ до цього поля, методу, класу?”

# Підсумок

- Java – мова С-подібна, але зі спрощеним і безпечнішим синтаксисом.
- Ми розглянули створення класів та використання конструкторів для ініціалізації об'єктів.
- Познайомились зі статичними методами, які можна викликати без створення об'єкта.
- Вивчили наслідування, яке дозволяє створювати підкласи і повторно використовувати код батьківського класу.
- Розібралися з модифікаторами доступу (`private`, `default`, `protected`, `public`) і їх впливом на видимість полів та методів.
- Підкреслили принцип інкапсуляції: обмежуємо доступ до внутрішнього стану об'єкта і забезпечуємо контролюваний доступ через методи.