

# ВСТУП ДО ПРОГРАМУВАННЯ МОВОЮ JAVA

Лекція 01  
Java-програмування

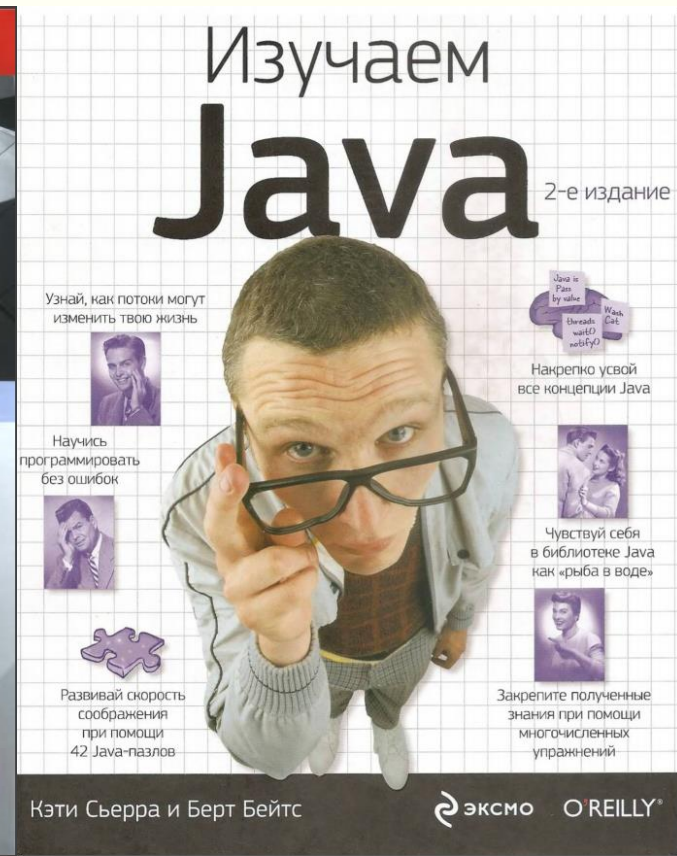
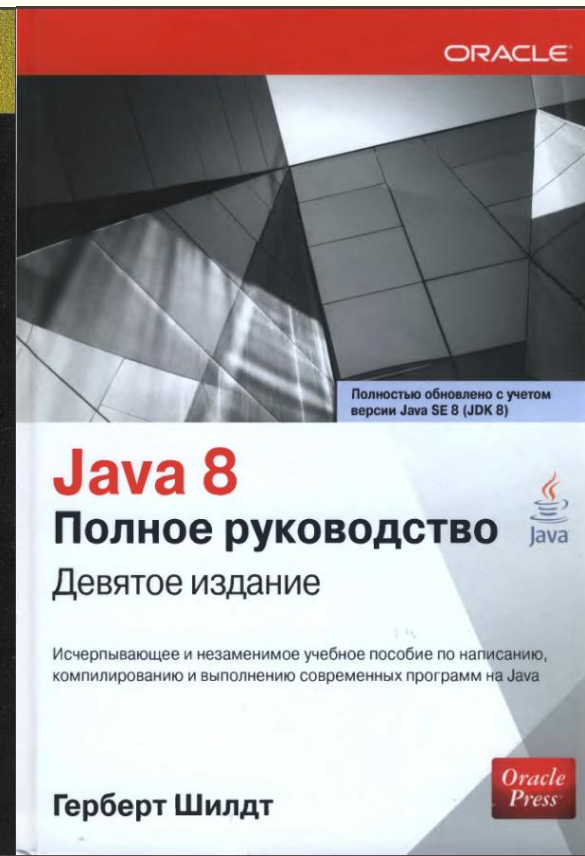
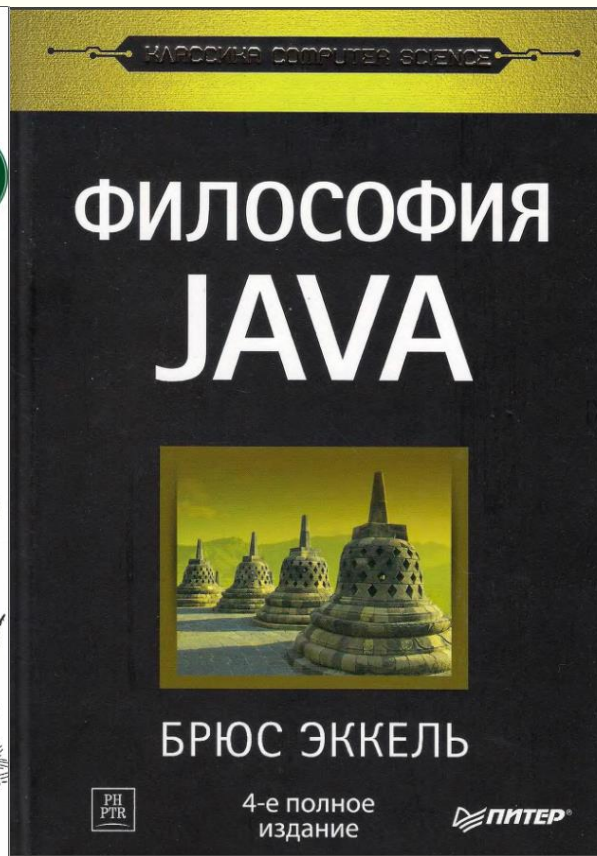


# План лекції

---

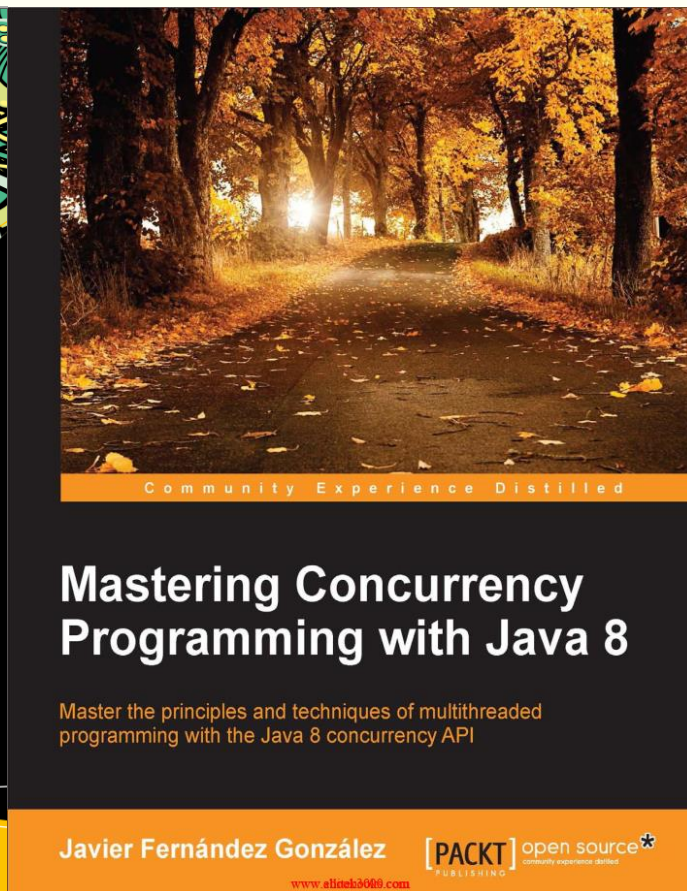
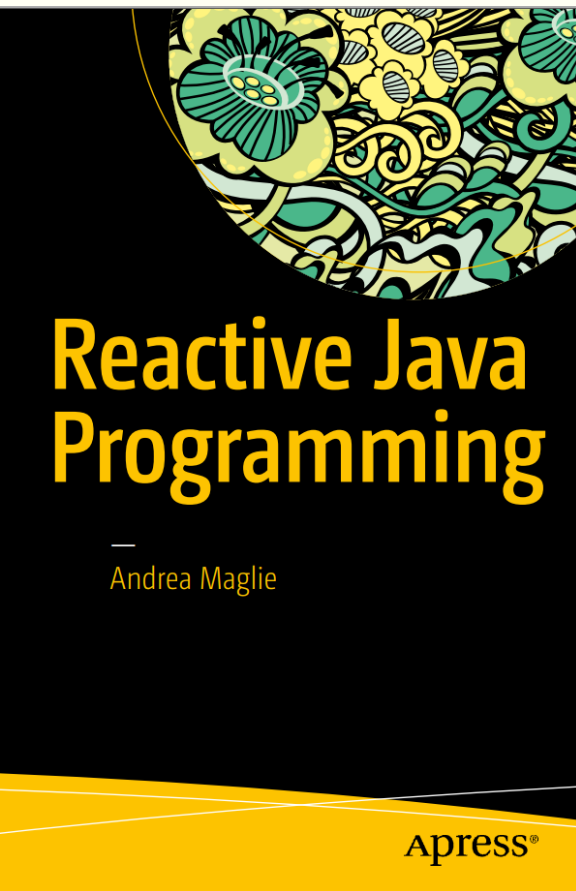
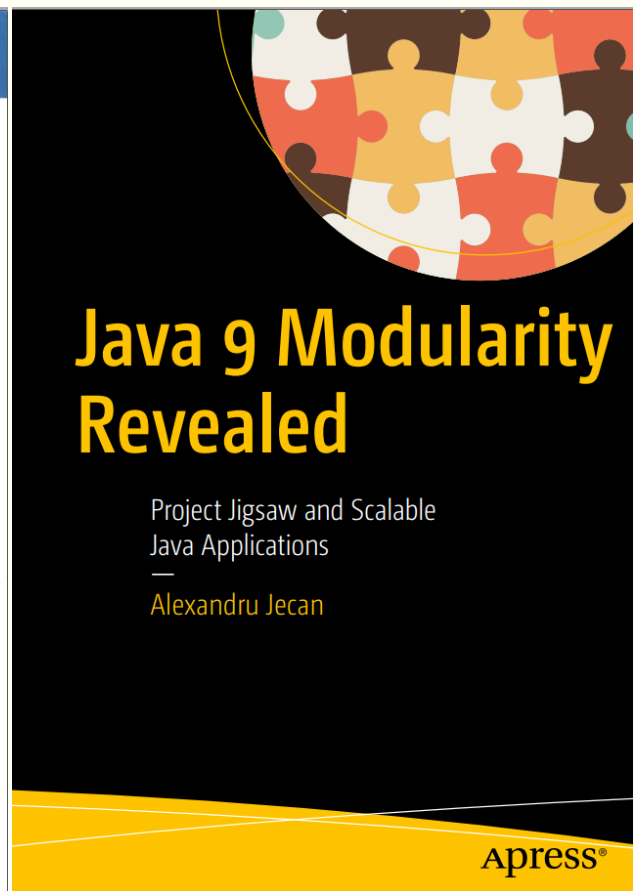
- **Java як мова та платформа**
- **Базові типи даних у мові Java**
- **Лексичні основи мови програмування Java**
- **Управління ходом виконання Java-програми**
- **Основи роботи з винятками та твердженнями**

# Література з предмету





# Додаткова література



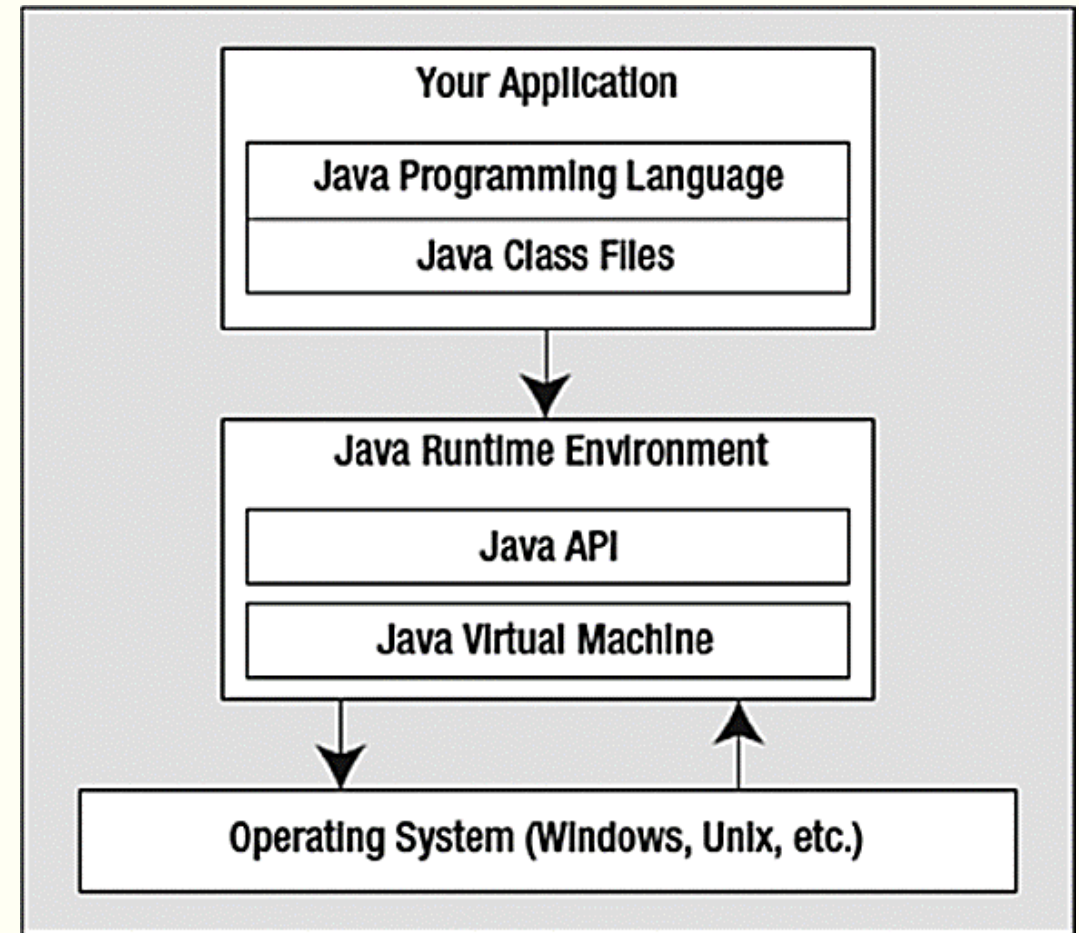


# JAVA ЯК МОВА ТА ПЛАТФОРМА

Питання 1.1.

# Архітектура платформи Java

- Комбінація 4 компонентів:
  - Мова програмування Java
  - Формат Java class-файл
  - Прикладні інтерфейси Java API
  - Java Virtual Machine
- Написаний код компілюється в Java class файли, які виконуються JVM
- Комбінація JVM та core classes формує Java-платформу, також відому як Java Runtime Environment (JRE)



# Основні платформи Java API

---



Java Platform, Standard Edition (Java SE): містить базові класи Java та класи GUI

Java Platform, Enterprise Edition (Java EE): включає класи для розробки більш складних «корпоративних» застосунків; містить сервлети, JavaServer Pages, Enterprise JavaBeans та ін.

Java Platform, Micro Edition (Java ME): забезпечує оптимізоване runtime-середовище для такої техніки, як Blu-ray плеєри, мобільні телефони та ін.

# Java Virtual Machine

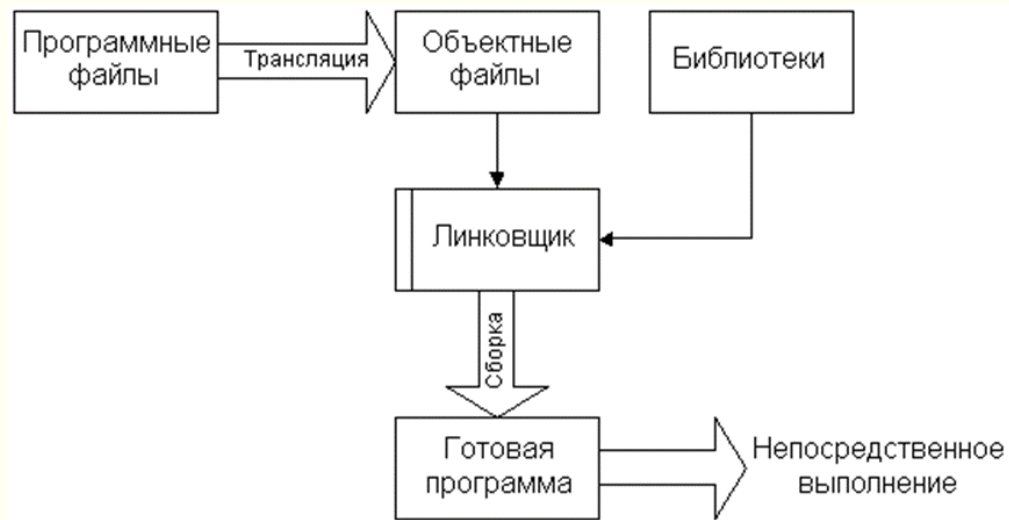
---

- Компілятор прив'язаний до конкретних процесорів та ОС.
  - Перевага – самодостатність скомпільованого файлу.
  - Недолік – нестача портативності, необхідно перекомпілювати код для різних систем.
- Для vendor-specific компіляторів код може не запуститись на іншому типі процесора, навіть якщо він підтримується ОС.
  - Особливо ця проблема стає помітною при написанні інтернет-застосунків.
  - Єдине вирішення – створити платформонезалежну мову.
- Компілятор Java створює не стандартний виконуваний код, а оптимізований набір інструкцій – байткод.
  - *Байткод* – послідовність байтів, яка інтерпретується runtime системою – JVM.
  - Загалом, байткод, згенерований на одній платформі, можна запустити на іншій, якщо там встановлено JVM.

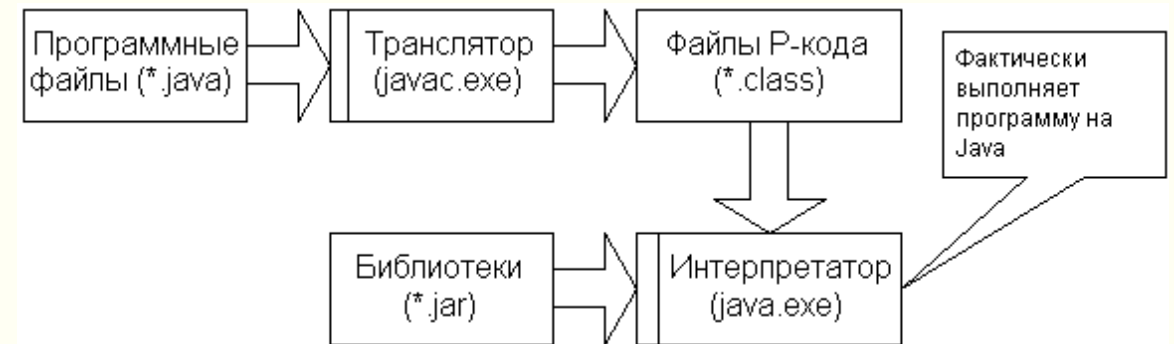


# Портабельність Java

- Первинний Java-код компілюється в байткод, який кожна версія JVM інтерпретує в нативні виклики, специфічні для кожної платформи (мову, яку може зрозуміти конкретний процесор)

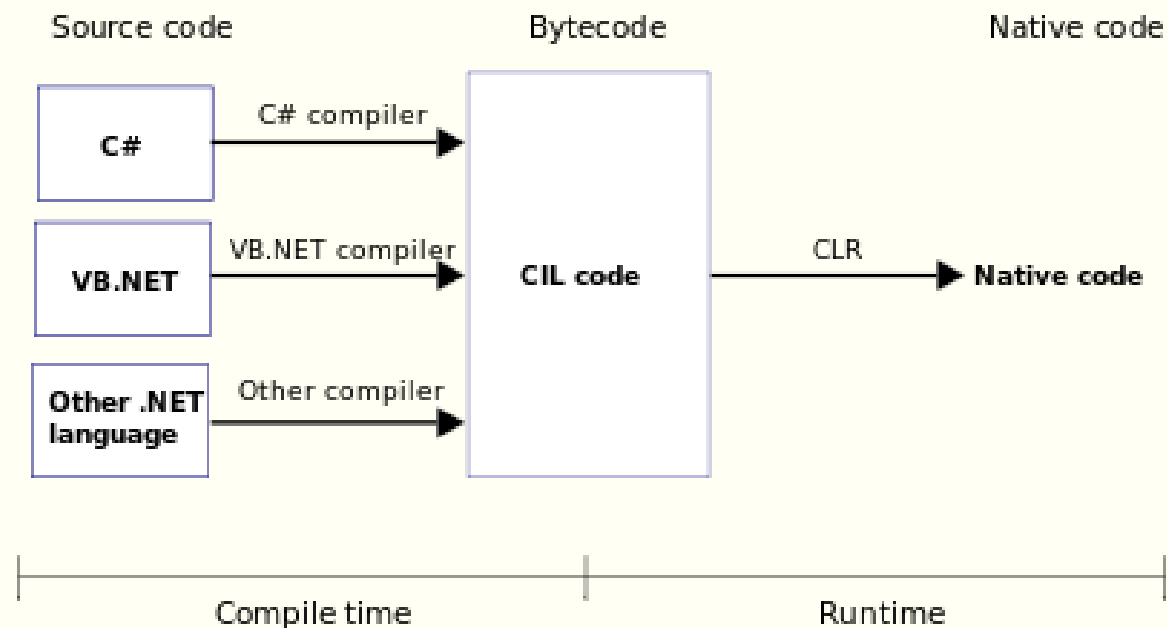
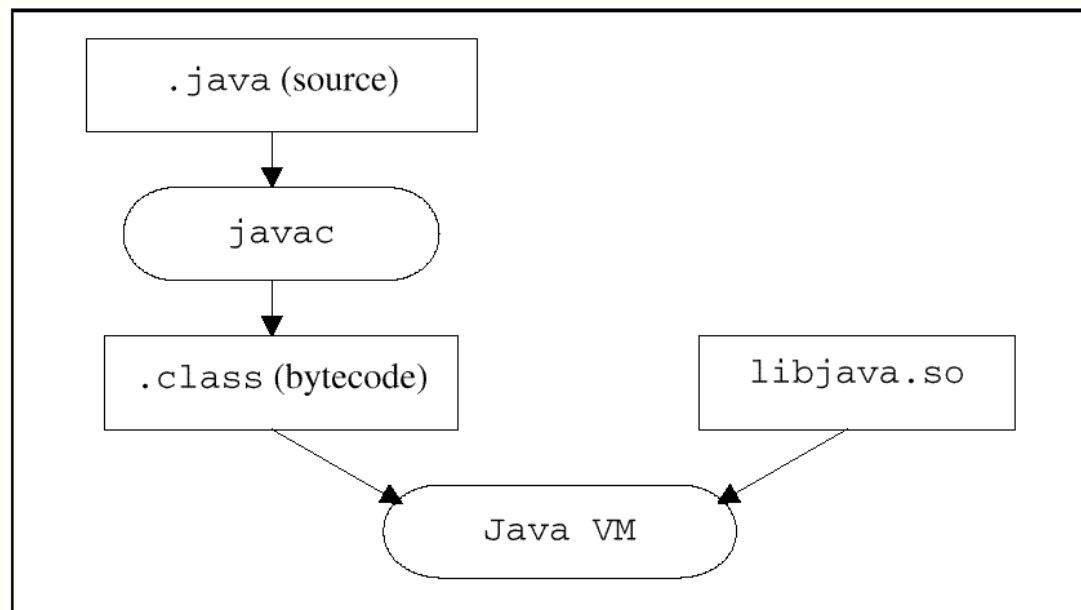


**Стандартна програма C/C++**



**Java-програма**

# Java i C# - один принцип



## Порівняння з C++

---

- Особливість: байткод напряму не виконується процесором, а проходить через JVM, яка його інтерпретує.
- Архітектура більшості реалізацій JVM – суміш інтерпретації та компіляції.
  - Чиста інтерпретація завжди повільніша за компіляцію, тому Java має репутацію повільної мови.
  - Гібридна модель значно зменшує відставання, роблячи мову підходящою для більшості застосунків, крім resource-intensive.

Мова	Комп. / Інтерпр.	Портабельність	Мінімальні витрати на виконання
C++	Компільована	Ні	Так
Java	Інтерпретована	Так	Ні

# Java Native Interface (JNI)

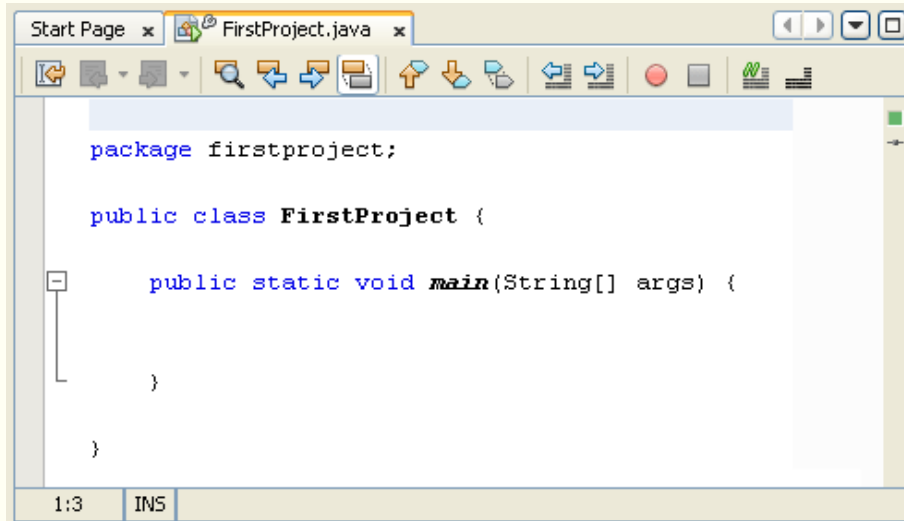
---

- Дозволяє викликати іншомовний код у Java-програмі, і навпаки.
  - Можна використовувати, наприклад, C/C++ для функцій, в яких важлива продуктивність.
  - Страждає переносимість через прив'язку нативного коду

```
package my.mega.pack;  
  
public class NativeCallsClass  
{  
    static  
    {  
        System.loadLibrary("megalib");  
    }  
  
    native public static void printOne();  
    native public static void printTwo();  
}
```



# Структура простої Java-програми



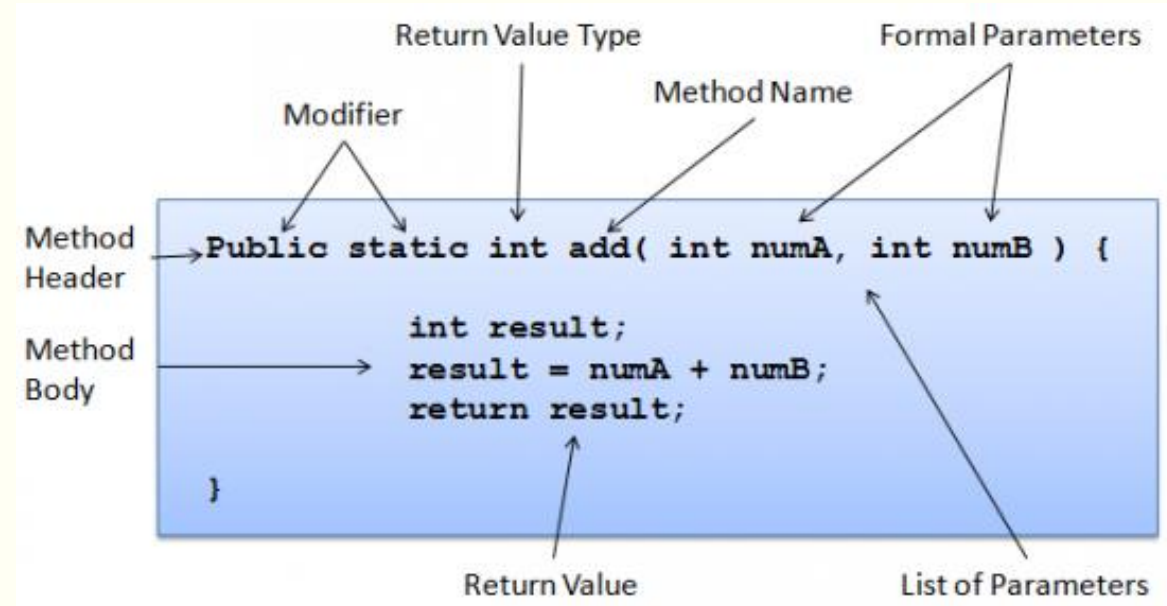
```
package firstproject;

public class FirstProject {

    public static void main(String[] args) {

    }

}
```



- JVM виконує лінування та ініціалізацію класу, викликає метод `main()`.
- `main()` управляє завантаженням (loading), лінковою (linking) та ініціалізацією будь-яких додаткових класів та інтерфейсів, на які є посилання

# Пакети (package)

---



- Пакет є спеціальним бібліотечним модулем, який містить групу класів, об'єднаних в одному просторі імен.
- Загальноприйнята схема іменування пакету: перша частина назви повинна складатись з перевернутого доменного імені розробника класу.
  - Оскільки доменні імена в інтернеті унікальні, забезпечується унікальність назв пакетів.
  - Це попереджує конфлікти.
  - Якщо власного доменного імені немає, придумайте власну унікальну комбінацію.

# Приклад для Android

---

- Існує системна бібліотека `android.widget`, до складу якої входить клас `Button`.
  - Щоб використовувати клас у програмі, можна навести його повну назву `android.widget.Button`.
  - Зручніше використовувати ключове слово `import`.
  - `import android.widget.Button;`
  - Тепер до класу `Button` можна звертатись без вказівки повного імені.
- Вказувати один і той же пакет можна в різних файлах, він просто вказує, кому належить клас.
- Пишуться повні імена, якщо, наприклад, в одному проекті використовують `java.sql.Date` і `java.util.Date`.

# Приклад для IBM

---

- Для комерційних проектів
  - директорія зазвичай починається з префіксу “com”,
  - за ним слідує назва компанії – наприклад “mysompany”,
  - далі йде назва проекту,
  - Потім іде більш чіткий поділ за будь-якою ознакою - найчастіше функціональною, але ніхто не заважає їх розбивати навіть в алфавітному порядку.
  - Коли компанія IBM створює проект education, то повний шлях до класів цілком може виглядати так: com/ibm/education.
- Для проектів OpenSource часто починається все з org.



# Байткод

---

- Байткод class-файлу складається з набору 1-байтних opcode інструкцій, які вказують, яку операцію необхідно виконати.
  - Після кожного опкоду йде 0 або більше операндів, що постачають аргументи чи дані для відповідної операції.

```
class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

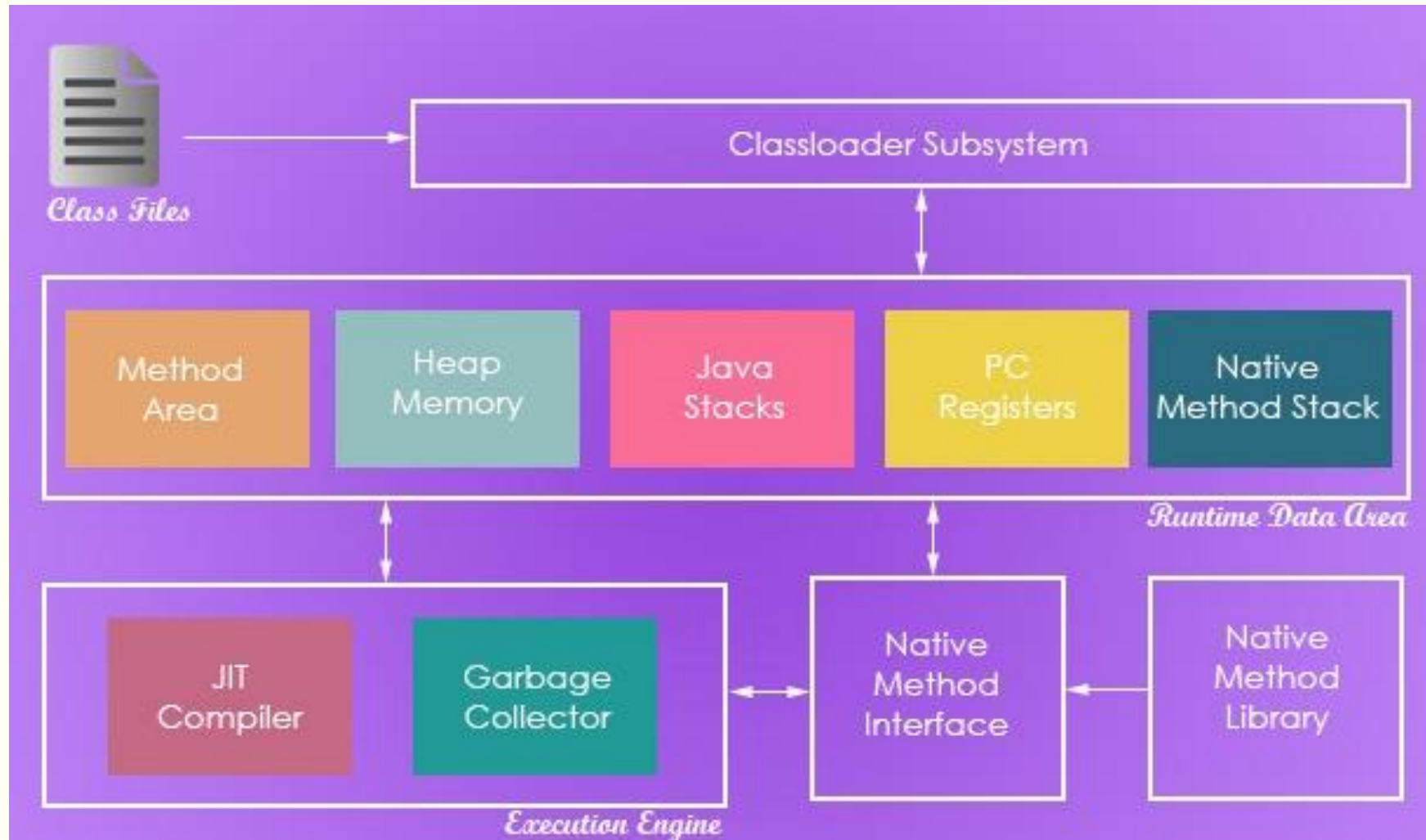
```
Compiled from "Hello.java"  
class Hello {  
    Hello();  
    Code:  
        0: aload_0  
        1: invokespecial #1           // Method java/lang/Object."<init>":()V  
        4: return  
  
    public static void main(java.lang.String[]);  
    Code:  
        0: getstatic     #2           // Field java/lang/System.out:Ljava/io/  
                                PrintStream;  
        3: ldc          #3           // String Hello World!  
        5: invokevirtual #4           // Method java/io/PrintStream.println:  
                                (Ljava/lang/String;)V  
        8: return  
}
```

# Віртуальна машина Java

---

- Виконує операції:
  - Завантаження потрібних .class- та jar-файлів
  - Присвоєння посилань та верифікація коду
  - Виконання коду
  - Постачає середовище виконання для Java-байткоду
  - Збирає сміття
- Має 2 різні реалізації:
  - **Java Hotspot Client VM:** віртуальна машина за умовчанням, починаючи з JDK 2.0. Налаштована для максимальної продуктивності при роботі додатків у клієнтському середовищі шляхом зменшення тривалості запуску додатку та кількості «слідів» у пам'яті
  - **Java Hotspot Server VM:** спроектована для покращеної швидкодії при роботі на серверному обладнанні. Дана VM викликається за допомогою опції серверного командного рядка.

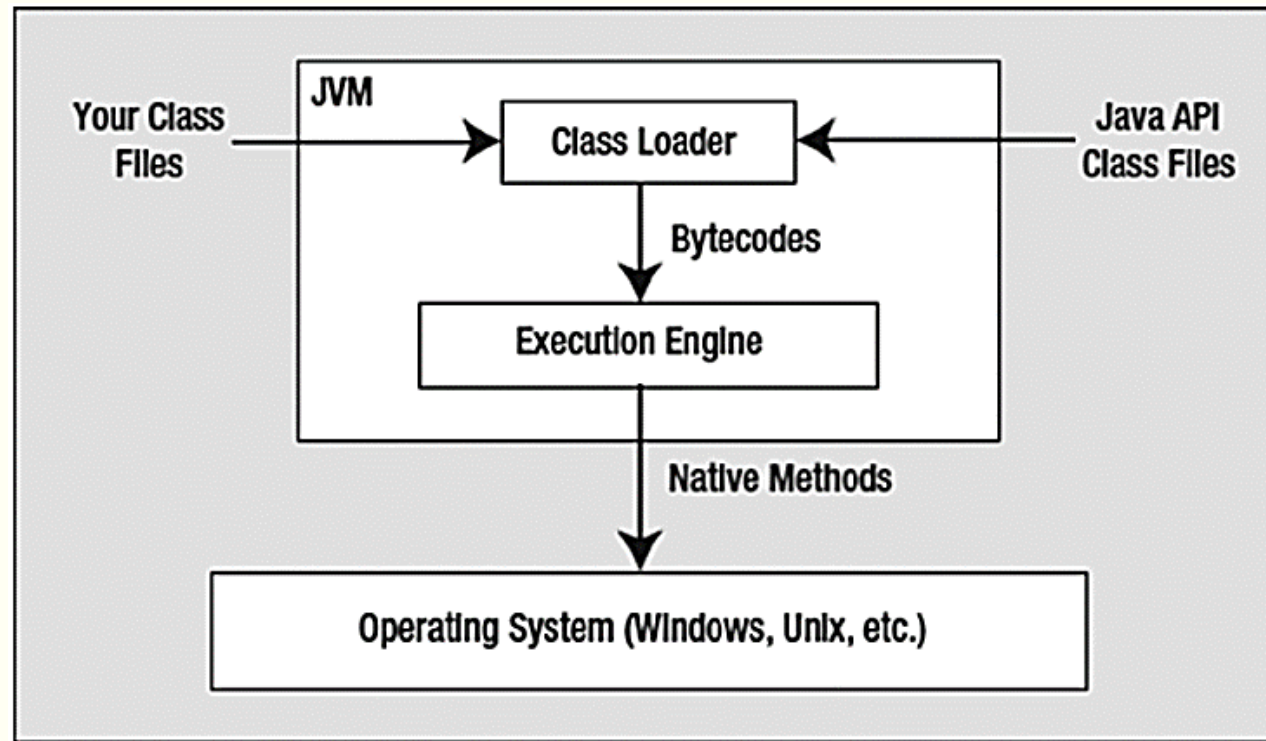
# Загальний вигляд архітектури



# JVM як середовище виконання

---

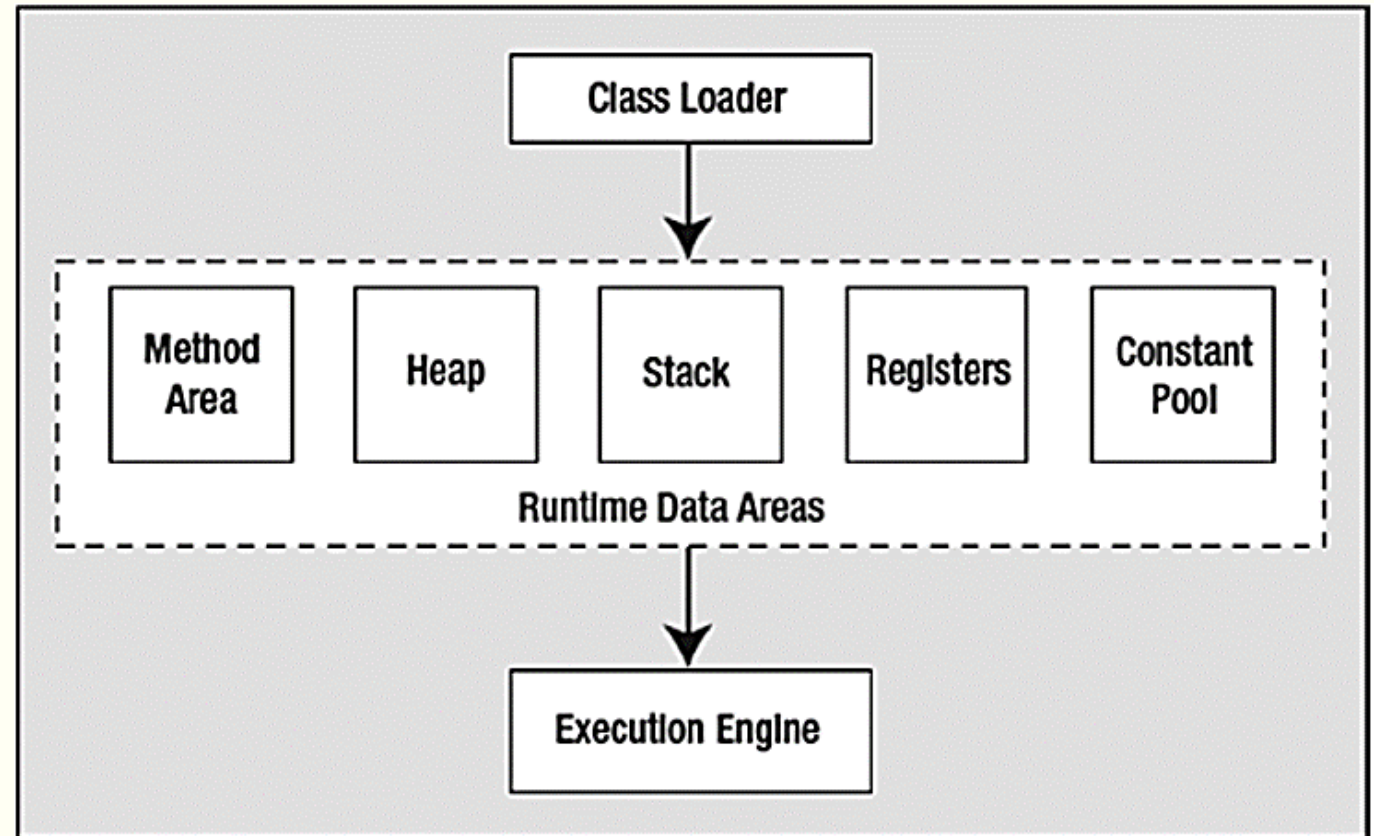
- Разом із запуском застосунку запускається і JVM.
  - Високорівневе представлення: байткод з файлу \*.class виконується за допомогою Execution Engine (JVM взаємодіє з нативними методами ОС)





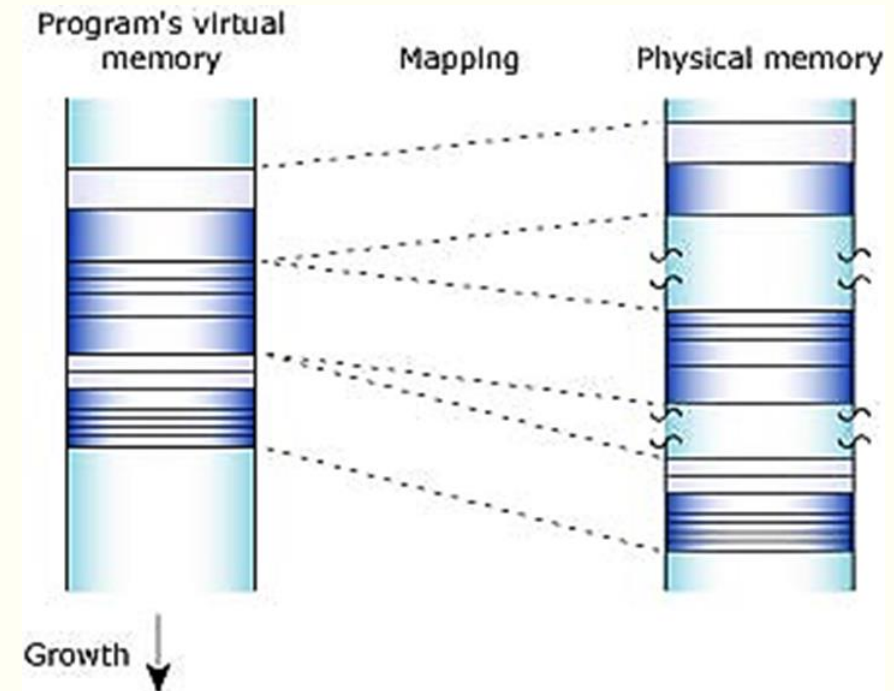
# Структури даних часу виконання (Runtime Data Areas) віртуальної машини Java

- JVM також потребує пам'яті, щоб зберігати тимчасові дані, що пов'язані з виконанням коду, наприклад, локальні змінні та ін.
  - Ці дані зберігаються в структурах даних часу виконання віртуальної машини Java



# Структури даних часу виконання у JVM

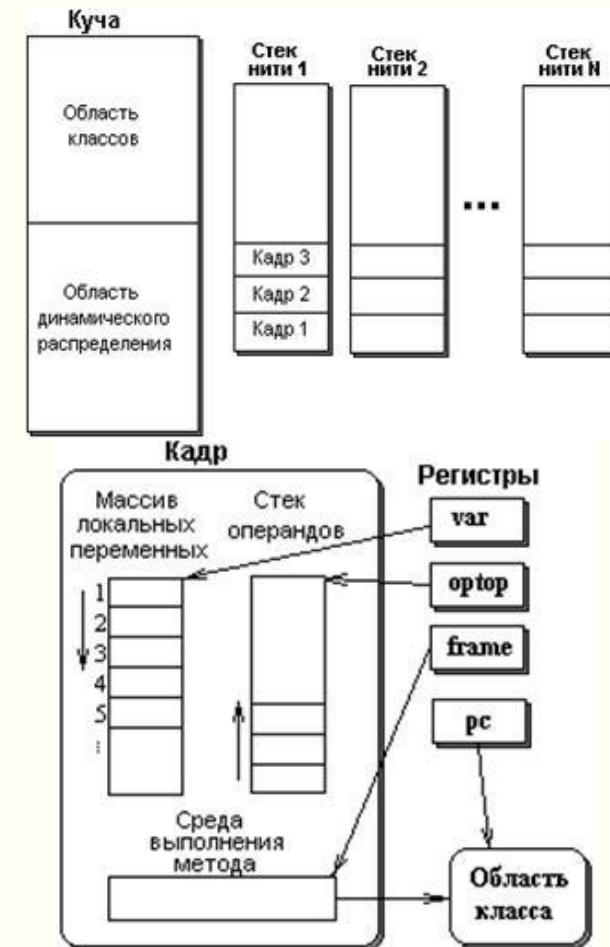
- Куча (Heap) – область вільної пам'яті, з якої виділяється пам'ять для класів та масивів.
  - Створюється при запуску JVM.
  - Виділяється класам та масивам.
  - Повертається при відсутності посилань на об'єкт (система збірки сміття).
  - Реалізація кучі не прописана в специфікаціях.
  - Розмір може бути як статичним, так і динамічним.
  - Не обов'язково розташовується в суміжних комірках.



Якщо кучі не вистачає пам'яті, а додаткову виділити неможливо, система згенерує виключення `OutOfMemoryError`

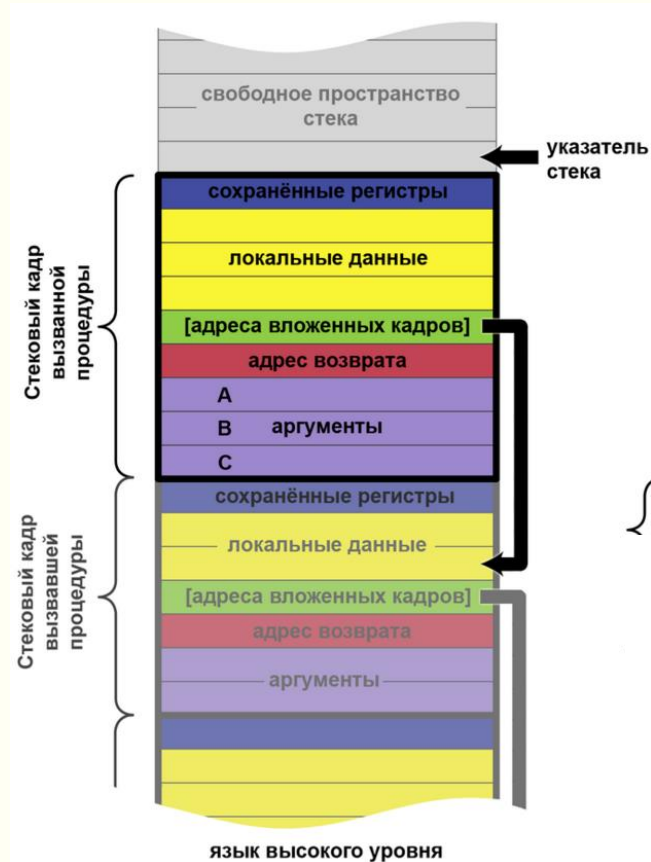
# Структури даних часу виконання у JVM

- **Стек (Stack)** містить стекові кадри (фрейми) та забезпечує вкладені виклики методів, представляючи кожен метод кадром у стеку.
  - Новий кадр створюється та поміщається у вершину стеку при виклику методу.
- Кожен кадр містить:
  - набір локальних змінних екземпляру класу, на котрий посилається регістр `var`;
  - стек операндів, на який посилається регістр `optr`;
    - Містить параметри та значення, що повертаються, для більшості інструкцій байткоду
  - структури середовища виконання, на яку посилається регістр `frame`.
    - Містить вказівники на різні аспекти виклику методу



<http://ermak.cs.nstu.ru/trans/Trans472.htm>

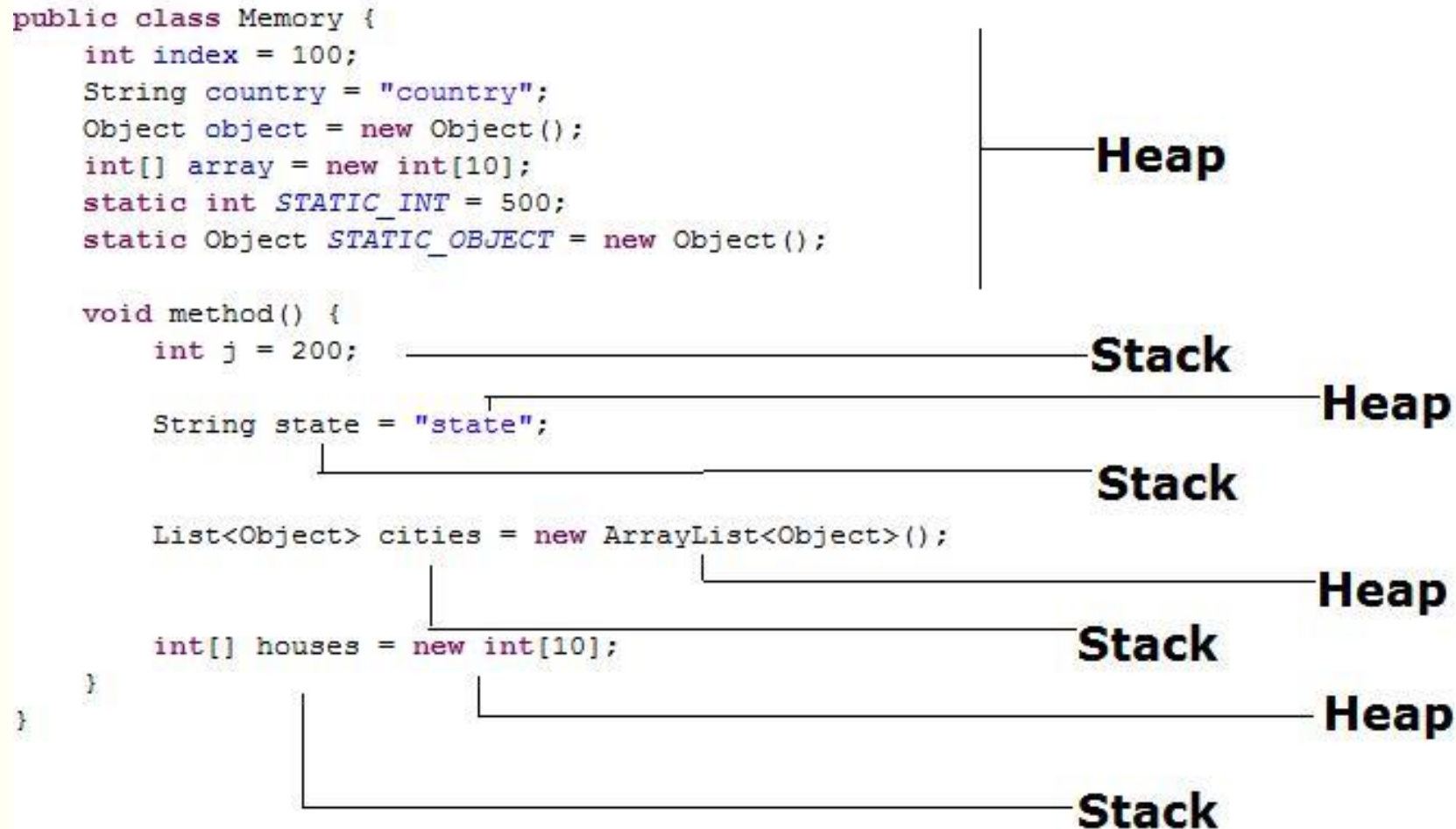
# Вигляд стекового кадру



- Стекові кадри (frame) виконують динамічне лінування та «викидають» винятки часу виконання (runtime exceptions)
  - Стековий кадр створюється при виклику методу та знищується при закінченні його роботи.
- Коли JVM виконує Java-код, в кожен момент часу активним є лише один кадр, що відповідає методу, який виконується.
  - Такий кадр називають **поточним**, відповідні метод та клас теж вважають **поточними**.
  - Коли нитка (thread) викликає метод, JVM створює новий (поточний) кадр та додає (push) його в стек нитки.
  - Якщо обчислення вимагають більшого розміру стеку, ніж можливо, генерується виключення `StackOverflowError`.



# Поля та методи в пам'яті



# Стек-трейс

---

```
public static void main(String[] args)
```

```
{  
    method1();  
}
```

```
public static void method1()
```

```
{  
    method2();  
}
```

```
public static void method2()
```

```
{  
    method3();  
}
```

```
public static void method3()
```

```
{  
    StackTraceElement[] stackTraceElements = Thread.currentThread().getStackTrace();  
    for (StackTraceElement element : stackTraceElements)  
    {  
        System.out.println(element.getMethodName());  
    }  
}
```

```
"C:\Program Files\Java\jdk1.8.0_66\bin\java" ...
```

```
getStackTrace
```

```
method3
```

```
method2
```

```
method1
```

```
main
```

```
invoke0
```

```
invoke
```

```
invoke
```

```
invoke
```

```
main
```

# Решта компонентів структури

---

- **Область методу (*Method area*)** зберігає пул runtime-констант, дані методу, поля та байткод для методів та конструкторів.
  - У специфікації чітко не прописані місце знаходження та способи реалізації області, фіксований чи динамічний розмір.
- **Регістри** відображають поточний стан машини та оновлюються в процесі виконання байткоду.
  - Головний регістр – pc (program counter) – вказує на адресу JVM-інструкції, що зараз виконується.
  - Якщо метод нативний, значення регістра – undefined.
  - Решта регістрів вказують на компоненти стекового кадру.
- **Пул констант часу виконання (Runtime Constant Pool)** містить числові літерали та константні поля.
  - Пам'ять виділяється з області методу, а пул конструюється при завантаженні class-файлу для класу або інтерфейсу.

# Збирач сміття (Garbage Collector)

---

- Java автоматично виділяє пам'ять при створенні об'єкту та вивільняє її, коли посилань на об'єкт більше не існує.
  - Збирач сміття переглядає Java-програму в процесі її роботи
- Java використовує набір м'яких посилань (soft pointers) для відстежування посилань на об'єкт та таблицю об'єкта (object table) для відображення м'яких посилань та посилання на об'єкт.
  - М'які посилання називають так, оскільки вони вказують не напряму на об'єкт, а на посилання на об'єкт.
  - Використання м'яких посилань дозволяє збирачеві сміття працювати в фоні, використовуючи окрему нитку, а також переглядати один об'єкт за раз.
  - Збирач сміття може відмітити (mark), видалити, перемістити або оглянути (examine) об'єкти, змінюючи object table entries.
- Збирач сміття працює окремо, тому викликати його явно зазвичай немає необхідності.
  - Він спорадично перевіряє посилання на об'єкт під час виконання програми і затребує області пам'яті, на які немає посилань.
  - Можна явно викликати збирач сміття за допомогою статичного методу gc() з класу System, проте гарантії виконання запиту немає.

---

---

Если бы в Java действительно работала  
сборка мусора, большинство программ бы  
удаляли сами себя при первом же запуске.

// tproger.ru

*Роберт Сьюэлл*



# ДЯКУЮ ЗА УВАГУ!

Наступне запитання: базові типи даних у мові Java