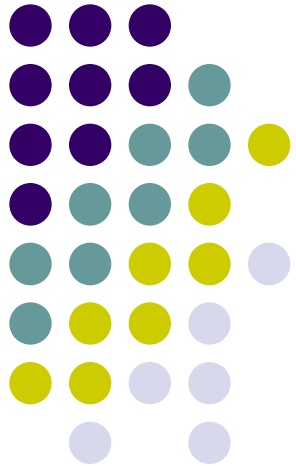
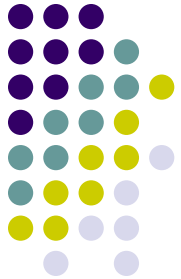
Мова програмування Java та технології J2EE
Модуль “Мова програмування Java”



Лекція 7. Потоки виконання. Паралельне виконання. Синхронізація потоків. Взаємодія потоків.

Сирота О.П.

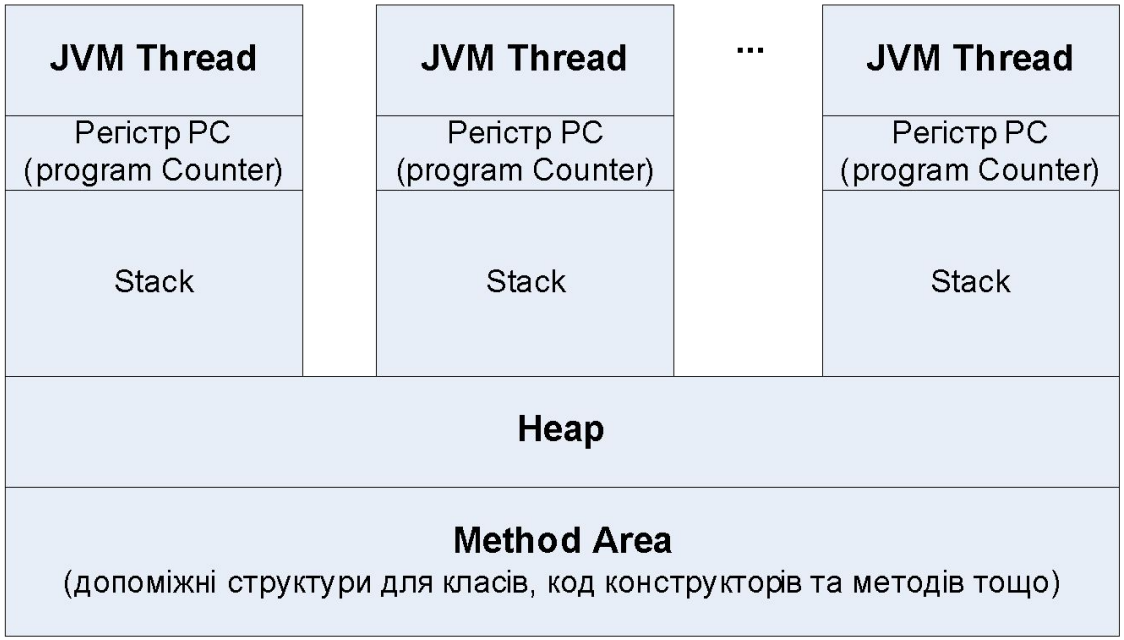
Тема лекції



- Поняття потоку виконання у мові Java
- Запуск потоку виконання
- Стани потоків виконання
- Синхронізація потоків
- Взаємодія потоків

Мова Java та потоки виконання

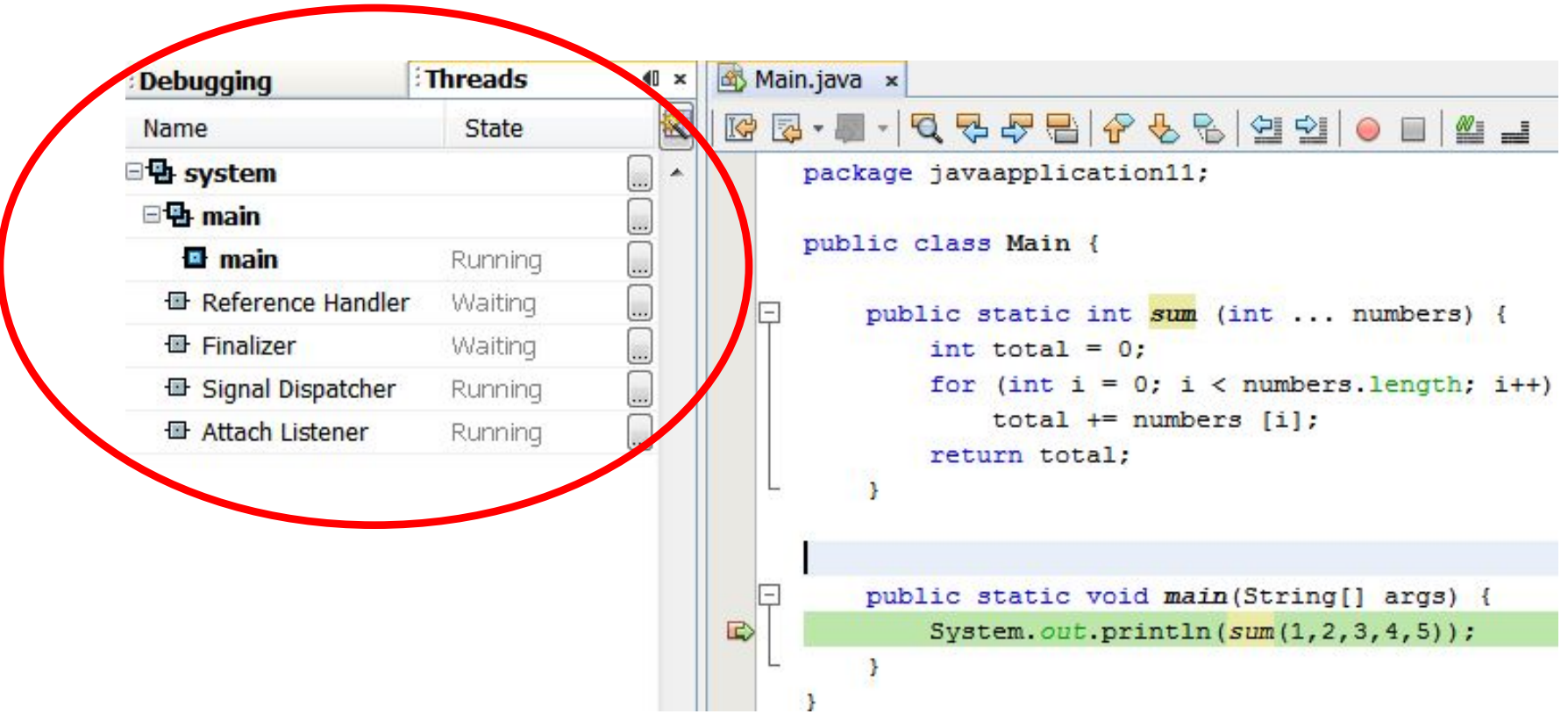
- Потоки виконання - це частини програми, які можуть виконуватись паралельно
- Java на відміну від багатьох інших мов програмування має вбудовані засоби підтримки потоків виконання та управління паралельним виконанням
- Це досягається завдяки тому, що JVM підтримує власні потоки виконання (JVM Thread)
- Потоки виконання JVM відображаються на потоки операційної системи та тим самим отримують обчислювальні потужності



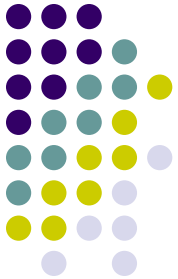
Структура JVM

Потоки виконання для Java-програми

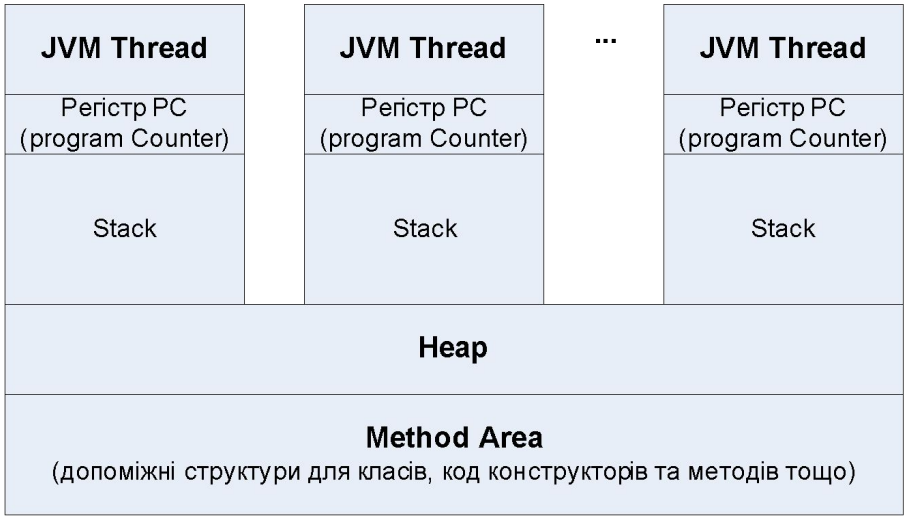
- Потоки виконання для простої Java-програми:
  - Потік “Main” (в якому виконується метод main)
  - Системні потоки (garbage collector, тощо)



Поняття потоку виконання в Java

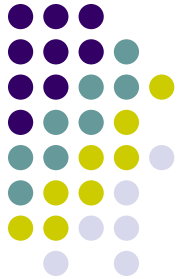


- Потоки виконання в Java
  - кожний потік виконання має свій стек та регістр program counter
  - можуть виконувати один й той самий код (але кожний потік буде виконувати код у своєму стеку)
  - мають доступ до одного й того самого адресного простору (JVM Heap) та можуть маніпулювати одними й тими самими даними



- Потоки виконання в Java – це екземпляри класу `java.util.Thread`

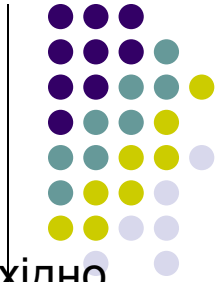
Алгоритм запуску потоку виконання



1. Визначити код, який буде виконуватись.
2. Створити екземпляр потоку та призначити йому код для виконання.
3. Запустити потік.

- Метод `java.util.Thread` для запуску потоку виконання:
  - `start`

Крок 1. Визначення коду для виконання



- Варіант 1 – **визначити код безпосередньо у потоці**. Для цього необхідно розширити клас **`java.lang.Thread`**

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("MyThread");
    }
}
```

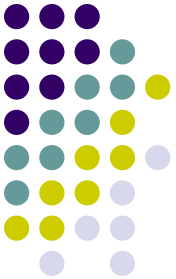
- Варіант2 – **визначити код у окремому класі**. Для цього необхідно реалізувати інтерфейс **`java.lang.Runnable`**

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("MyRunnable");
    }
}
```

Реалізувати інтерфейс **`Runnable`** – це кращий спосіб визначення коду для потоку виконання

- оскільки дозволяє відокремити код потоку від коду «роботи», яку потік виконує

Крок 2. Створення екземпляру потоку виконання та призначення коду для виконання



- Варіант 1 – якщо визначено нащадок класу **Thread**

```
MyThread t = new MyThread();
```

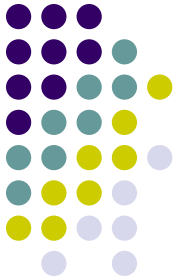
- Варіант 2 – якщо реалізовано інтерфейс **Runnable**

```
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
```

Один екземпляр **Runnable** можна передати декільком об’єктам **Thread**

```
MyRunnable r = new MyRunnable();
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
Thread t3 = new Thread(r);
```

Крок 3. Запуск потоку виконання

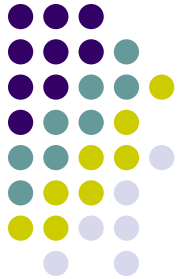


- **За допомогою методу `start()` класу `Thread`**

- **Приклад**

```
public static void main(String[] args) {
    MyRunnable r = new MyRunnable();
    Thread t = new Thread(r);
    t.start();
}
```

Приклад. Запуск декількох потоків



```
public class ThreadStarter {
    public static void main(String[] args) {
        NamedRunnable nr = new NamedRunnable();
        Thread one = new Thread(nr);
        Thread two = new Thread(nr);
        Thread three = new Thread(nr);

        one.setName("Первый");
        two.setName("Второй");
        three.setName("Третий");

        one.start();
        two.start();
        three.start();
    }
}

class NamedRunnable implements Runnable {
    public void run() {
        System.out.println("Запущен " + Thread.currentThread().getName());
        System.out.println("Закончен "+ Thread.currentThread().getName());
    }
}
```

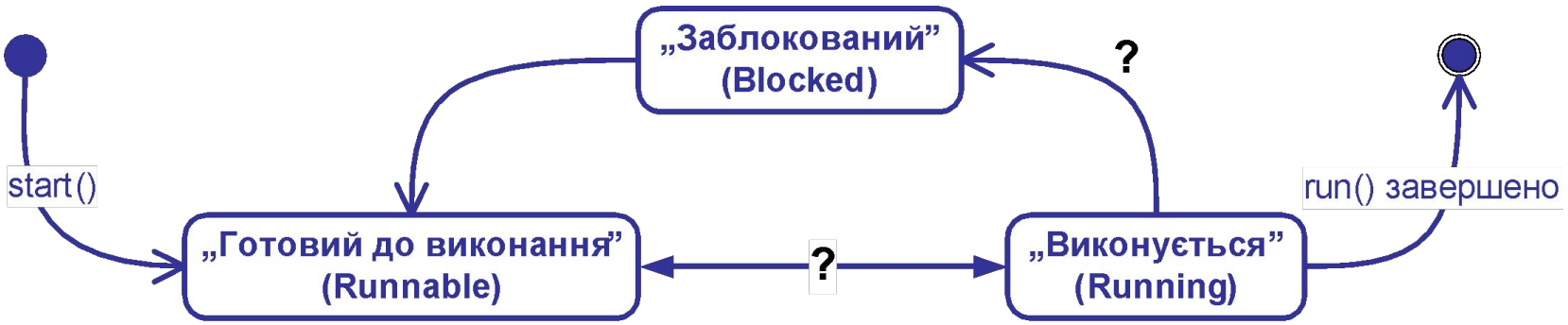
Основні стани потоків виконання

- Після запуску потоку за допомогою методу `start` він не відразу отримує обчислювальні потужності

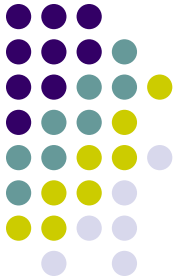
- Основні стани потоків виконання:
  - **Готовий до виконання** – потік переходить в цей стан після виклику методу `start`
  - **Виконується** – такому потоку надано обчислювальні потужності
  - **Заблоковано** – виконання такого потоку призупинено

- Після завершення виконання потік не може бути запущений знову.

- Перевірка стану потоку
  - Клас `Thread`, метод `isAlive` – поток “живий”, якщо він стартований, але виконання методу run ще не завершено.

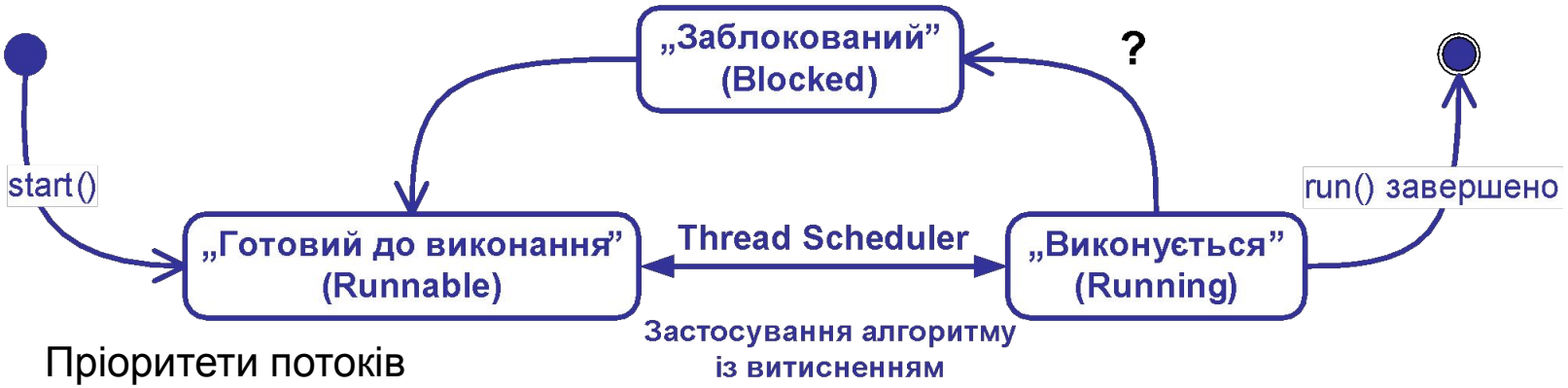


Планувальник потоків виконання



- Планувальник потоків виконання (Thread Scheduler)
  - Складова JVM
  - **Надає потоку виконання обчислювальні потужності**

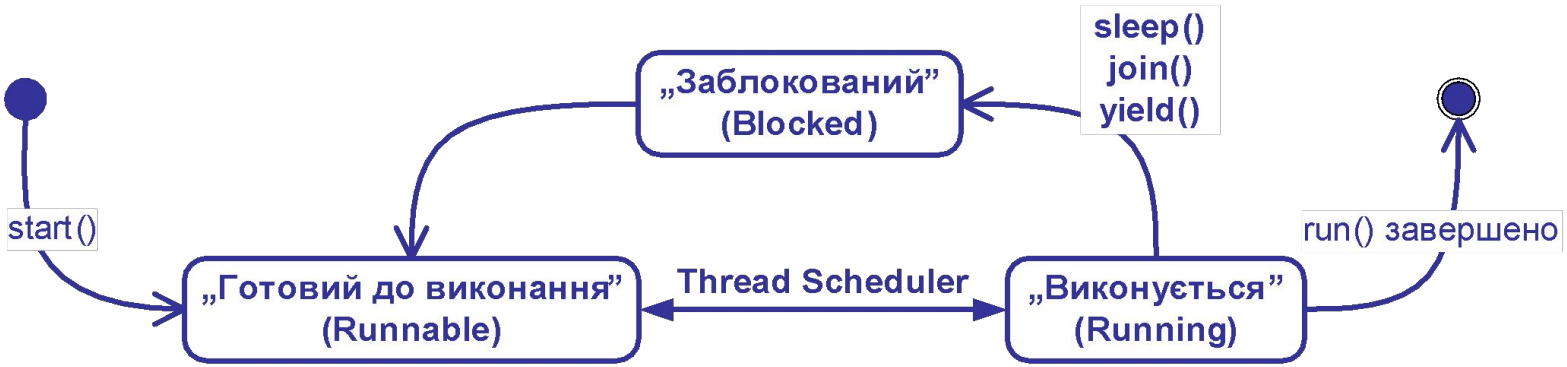
- Планувальник застосовує **алгоритм із витисненням на основі пріоритетів**
  - Планувальник обирає для виконання потік з найвищим пріоритетом, який знаходиться у стані готовності до виконання (runnable)
  - Якщо з'являється інший потік із вищим пріоритетом у стані готовності до виконання, то застосовується витиснення – потік з меншим пріоритетом повертається у стан готовності до виконання, а потік з більшим пріоритетом виконується



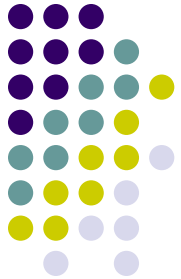
- Пріоритети потоків
  - `Thread.MIN_PRIORITY (1)`
  - `Thread.MAX_PRIORITY (10)`
  - `Thread.NORM_PRIORITY (5)`
- Метод класу `Thread` для встановлення пріоритету потока
  - `public final void setPriority(int newPriority)`

Блокування потоків виконання

- Методи для тимчасового блокування виконання потоку:
  - `public static void sleep(long millis) throws InterruptedException`
  - `public static void yield()`
  - `public final void join() throws InterruptedException`
- **`sleep`** – заснути на мілісекунди. Статичний метод, який діє на потік виконання, в він якому викликаний. Метод **`sleep`** дає змогу виконатися потокам із меншими пріоритетами
- **`join`** – чекати, поки вказаний потік не завершить виконання (див. приклад на наступних слайдах)
- **`yield`** – дати шанс іншим потокам виконатися. Якщо є потоки у стані “готовий до виконання”, то вони будуть переведені у стан “виконуються”. Якщо таких потоків немає, то потік продовжить виконання.

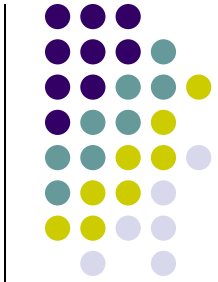


Приклад sleep



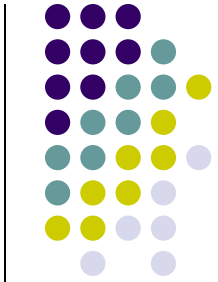
```
public class ThreadStarter {
    public static void main(String[] args) {
        NamedRunnable nr = new NamedRunnable();
        Thread one = new Thread(nr);
        Thread two = new Thread(nr);
        Thread three = new Thread(nr);

        one.setName("Первый");
        two.setName("Второй");
        three.setName("Третий");

        one.start();
        two.start();
        three.start();
    }
}

class NamedRunnable implements Runnable {
    public void run() {
        System.out.println("Запущен " + Thread.currentThread().getName());
        try {
            Thread.sleep(1000); // діє на той потік, в якому викликаний sleep
        }
        catch (InterruptedException ex) {}
        System.out.println("Закончен " + Thread.currentThread().getName());
        }
    }
}
```

Приклад join



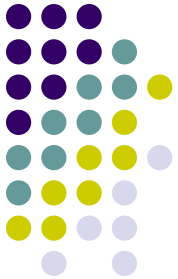
```
public static void main(String[] args) {

    Thread t = new Thread(new NamedRunnable());
    t.start();

    ...

    // почекаємо поки t не закінчиться
    try {
        t.join();
    }
    catch (InterruptedException e) {
    }
    // продовжимо
    ...
}
```

Типи потоків виконання



- Типи потоків:
  - Потоки-“демони” (daemon threads) – «сервісні» потоки. Зазвичай виконуються з низьким пріоритетом. Прикладом є garbage collector thead.
  - Користувацькі потоки (user threads)
- JVM завершує своє виконання, коли завершують виконання усі користувацькі потоки
- Будь-який потік може стати “демоном”
  - Методи `Thread: setDaemon, isDaemon`

Підсумок - клас Thread



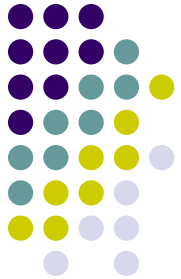
Основні методи

- `static currentThread`
- `static dumpStack`
- `static getAllStackTraces`
- `getId/setId`
- `getName/setName`
- `getPriority/setPriority`
- `getState`
- `interrupt`
- `isAlive`
- `isDaemon/setDaemon`
- `join`
- `run`
- `sleep`
- `start`
- `yield`

Конструктори

```
Thread()
Thread(String name)
Thread(Runnable runnable)
Thread(Runnable runnable,
       String name)
Thread(ThreadGroup g, Runnable
       runnable)
Thread(ThreadGroup g, Runnable
       runnable, String name)
Thread(ThreadGroup g, String
       name)
```

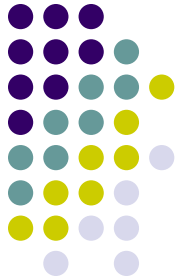
Цілісність даних



```
1 public class NotSyncStack {
2     int idx = 0;
3     char[] data = new char[6];
4
5     public void push(char c) {
6         data[idx] = c;
7         idx++;
8     }
9
10    public char pop() {
11        idx--;
12        return data[idx];
13    }
14 }
```

- Нехай потік 1 та потік 2 одночасно працюють з одним й тим самим екземпляром `NotSyncStack`
  - Уявіть, що потік 1 виконав рядок 11, а потік 2 виконав рядок 6.
  - **В результаті цілісність даних порушено.**

synchronized



- У Java кожний об’єкт має **флаг блокування (об’єктний монітор)**
- Оператор **`synchronized`** захвачує флаг блокування об’єкту, в результаті чого потік отримує ексклюзивний доступ до блоку, захищеного оператором коду

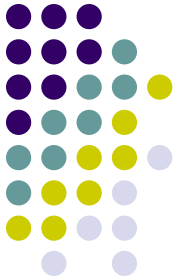
```
synchronized (вираз) {
      …
}
```

Дії потоку виконання:

1) Захват флагу блокування об’єкту, на який вказує *вираз*
2) Виконання блоку команд
3) Звільнення повернення флагу блокування

- Нехай виконуються потік 1 та потік 2 та намагаються виконати один й той самий блок коду, що захищений оператором **`synchronized`**
  - Коли потік 1 намагається виконати **`synchronized`**, то він захоплює флаг блокування.
  - Після цього коли потік 2 намагається виконати **`synchronized`** на тому самому об’єкті, то флаг блокування відсутній (захоплений іншим потоком). В результаті потік 2 стає в чергу очікування, яка асоційована з вказаним об’єктом.

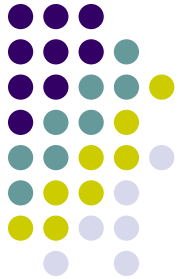
synchronized. Продовження



- Оператор **synchronized** не блокує вказаний об’єкт, а блокує доступ до коду
- Увага! Механізм **synchronized** коректно працює тільки якщо оператором **synchronized** захищати усі ділянки модифікації даних, тому
  - За допомогою **synchronized** повинні бути захищені усі методи, які модифікують дані
  - Усі дані, які модифікуються всередині блоку **synchronized**, повинні бути **private**
- У прикладі оператором **synchronized** захищені усі методи, які модифікують дані - **push** та **pop**

```
public class SyncStack {
    int idx = 0;
    char[] data = new char[6];

    public void push(char c) {
        synchronized(this) {
            data[idx] = c;
            idx++;
        }
    }
    public char pop() {
        synchronized(this) {
            idx--;
            return data[idx];
        }
    }
}
```

Варіанти застосування synchronized

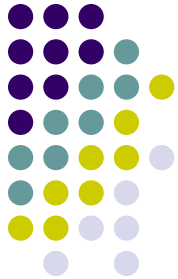


```
public void push(char c) {
    synchronized(this) {
    // The push method code
    }
}
```

Еквівалентна форма:

```
public synchronized void push(char c) {
    // The push method code
}
```

synchronized для статичних методів



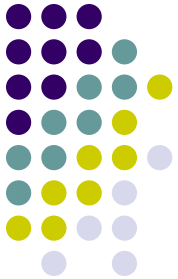
Можливе застосування **synchronized** для статичних методів:

```
class MyStaticSyncClass {
    static int count;
    public static synchronized int getCount() {
        return count;
    }
}
```

- Що є флагом блокування? - Екземпляр **java.lang.Class**
- Еквівалентна форма запису:

```
class MyStaticSyncClass {
    static int count;
    public static int getCount() {
        synchronized(MyStaticSyncClass.class) {
            return count;
        }
    }
}
```

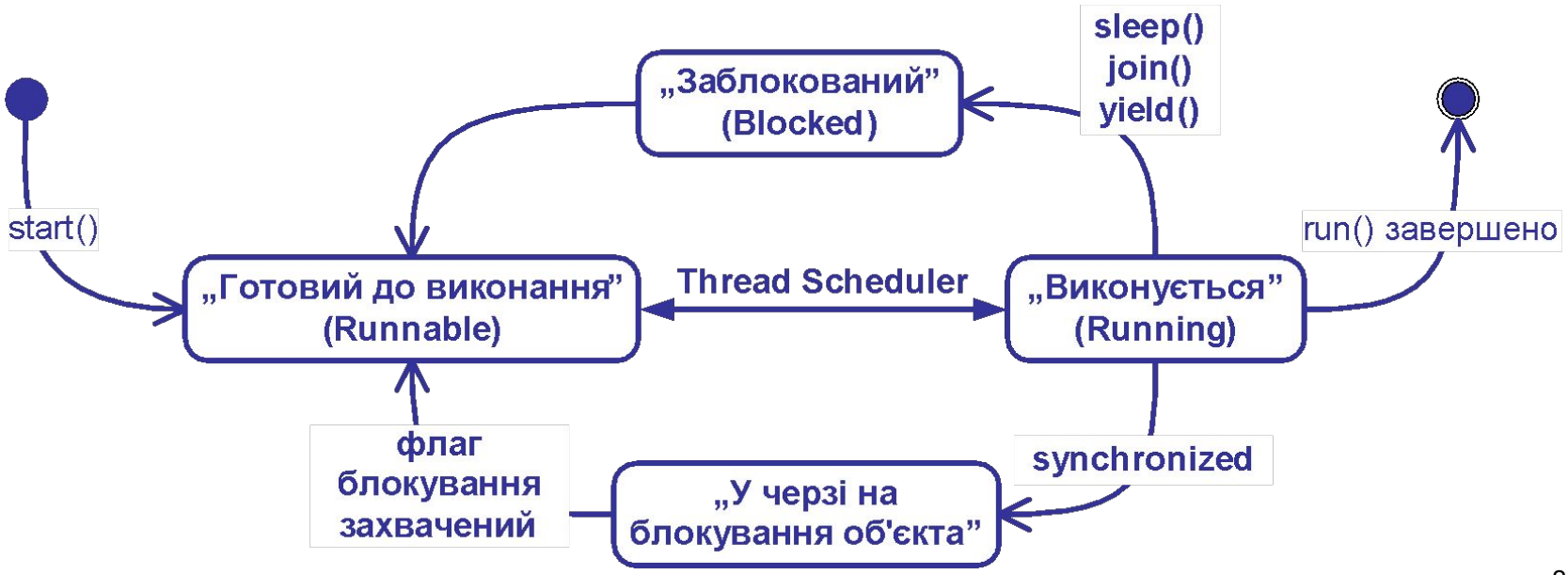
Повернення флагу блокування



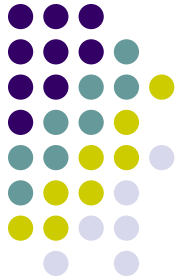
- Технологія Java гарантує, що флаг блокування повертається автоматично у наступних випадках:
  - завершення `synchronized` блоку
  - `return`
  - `break`
  - `throw`

Діаграма станів потоку з врахуванням synchronized

- Механізм синхронізації додає новий стан для потоку виконання – “**у черзі на блокування об’єкта**”. Із цього стану потік виконання виходить, коли захватує флаг блокування об’єкта

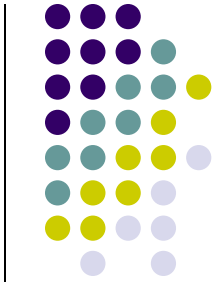


Колекції



- Чи є колекції (похідні від інтерфейсів Collection та Map) thread-safe?
- Ваша відповідь?
- Вірна відповідь:
  - «Нові» колекції – не thread-safe
    - `ArrayList, LinkedList, Queue, Dequeue, HashMap, TreeMap, HashSet, TreeSet` та інші
  - «Старі» колекції – thread-safe
    - `Vector, Hashtable`

Thread-safe-оболонки для колекцій



Матеріал із лекції 6

- Thread-safe оболонки для колекцій

  У класі `Collections` містяться наступні методи:

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

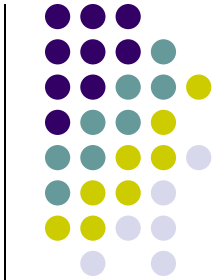
- Приклад:

```
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e : c)
        foo(e);
}
```

Питання: Навіщо `c = Collections.synchronizedCollection` та відразу після цього `synchronized(c)`?

Відповідь: Усі методи для модифікації синхронізованої колекції є thread-safe. Але при ітеруванні колекції колекція може бути змінені (за допомогою метода Iterator.remove). Саме тому будь-яке ітерування необхідно обрамляти synchronized

Взаємне блокування (deadlock)

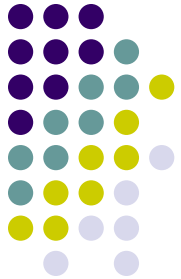


- Взаємне блокування – коли два потоки виконання чекають один від одного, поки інший відпустить флаг блокування того самого об’єкту.

```
public class DeadlockRisk {
    private MyResource resourceA = new MyResource();
    private MyResource resourceB = new MyResource();
    public int read() {
        synchronized (resourceA) { // deadlock может быть здесь
            synchronized (resourceB) {
                return resourceB.value + resourceA.value;
    } } }
    public void write(int a, int b) {
        synchronized (resourceB) { // deadlock может быть здесь
            synchronized (resourceA) {
                resourceA.value = a; resourceB.value = b;
    } } }
}
```

- Рекомендації для запобігання взаємного блокування:
  - Стежте за порядком блокування
- Такі утиліти Java як **jconsole, jstack** дозволяють виявити блокування. Ці утиліти входять у склад JDK та можуть застосовуватись для моніторингу як локальних, так і віддалених процесів.

Паралельне виконання та synchronized

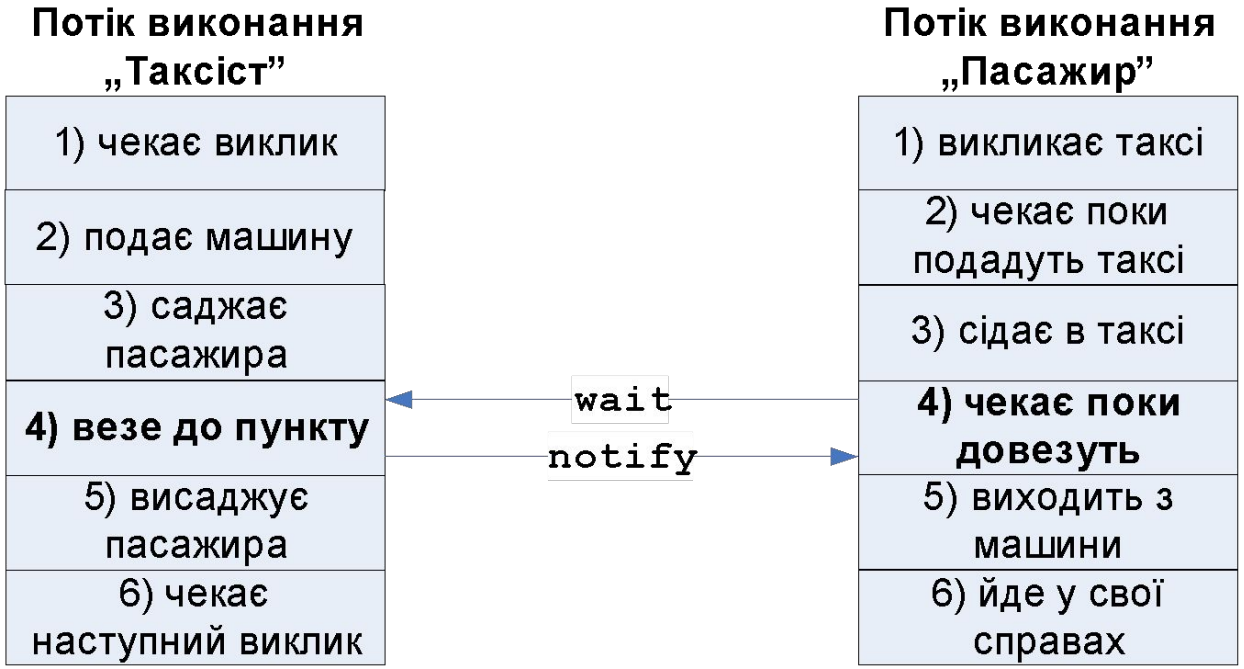


- Увага!
  - Блокування за допомогою synchronized “шкодить” паралельному виконанню
  - Тому “синхронізувати” необхідно найменшу кількість рядків коду

  - Наведіть приклади, які об’єкти необхідно синхронізувати в розподілених високо навантажених системах
    - Connection Pool (БД, LDAP тощо) - методи отримання з’єднання (connection)

Взаємодія потоків

- Сюжет - пасажир їде в таксі
  - Якщо перевести цю ситуацію у потоки Java, то є потік таксиста та потік пасажира



- Розглянемо фрагмент 4) із схеми. Пасажир може дізнатися, що вже приїхали 2 способами:
  - Через кожні 2 секунди пасажир питає “Ми вже приїхали?”
  - Пасажир чекає, поки водій сповістить, що приїхали
- Для взаємодії потоків у стилі “чекати”- “сповістити” призначені методи класу `Object - wait, notify, notifyAll`

Взаємодія потоків `wait-notify`

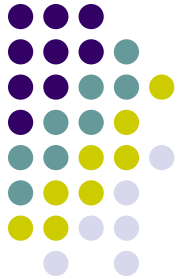
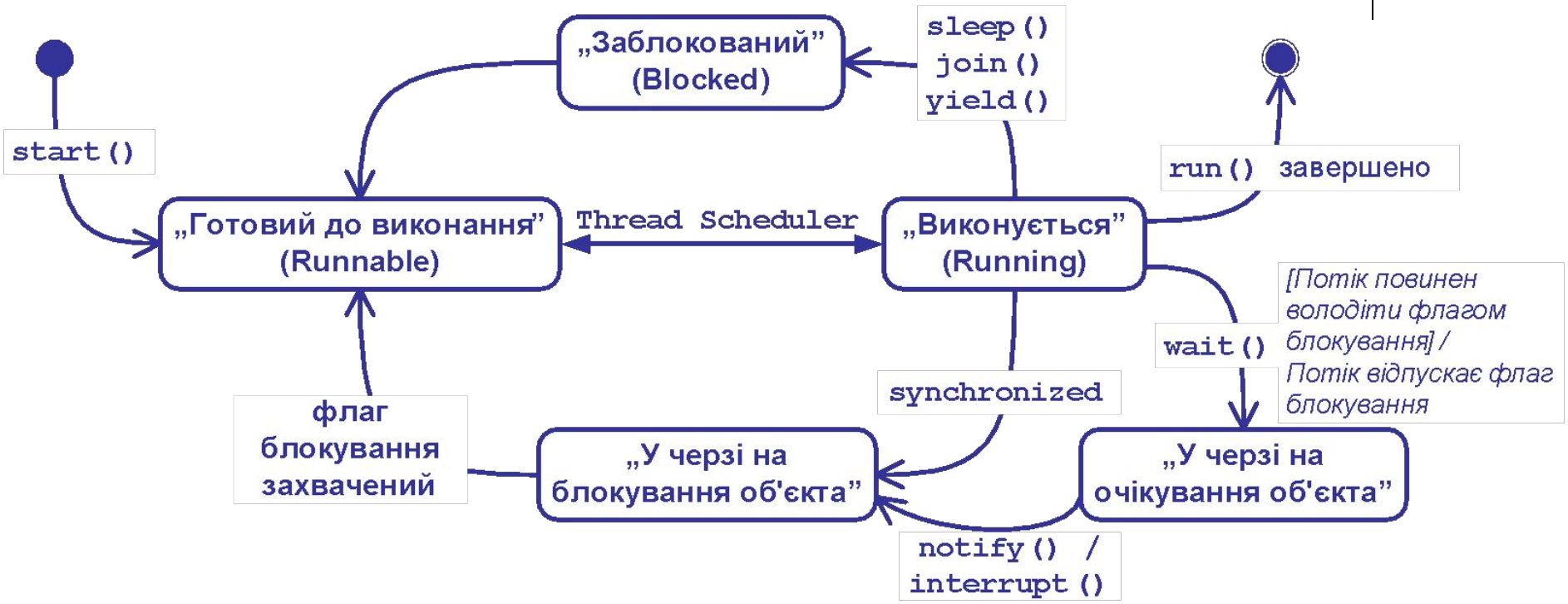


Схема роботи **`wait-notify:`**

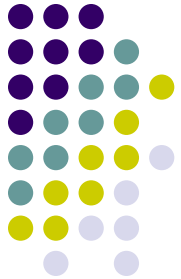
- Якщо в потоці викликано **`obj.wait()`**, то потік призупиняє виконання та потрапляє в чергу очікування об'єкту **`obj`**
- Потік видаляється із черги очікування **`obj`** коли інший потік виконує **`obj.notify()`** або **`obj.notifyAll()`** – для того самого об'єкта **`obj`**
- Якщо **`obj.wait()`** на одному й тому самому об'єкті **`obj`** виконали декілька потоків, то
  - виклик **`obj.notify()`** відновлює роботу тільки одного потоку, що очікує **`obj`**
  - виклик **`obj.notifyAll()`** відновлює роботу усіх потоків, що очікують **`obj`**
- **`wait-notify`** можуть бути застосовані для будь-якого об'єкту, у тому числі об' єкту типу **`Thread`**
- Питання – у прикладі з минулого слайду який об'єкт слід застосувати для **`wait-notify`**

Діаграма станів потоку з врахуванням `wait-notify`



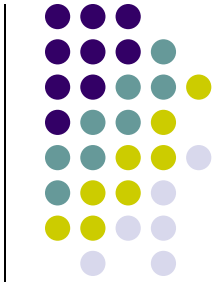
- **`wait, notify, notifyAll`** повинні викликатися із **`synchronized`**-блоку, інакше отримаємо виключну ситуацію
- **`wait`** – звільняє флаг блокування об’єкта

Приклад



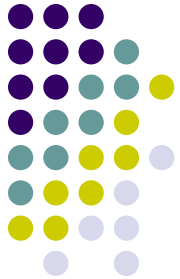
```
public class SyncStack {

    private List<Character> buffer=new ArrayList<Character>(400);

    public synchronized char pop() {
        char c;
        while (buffer.size() == 0) {
            try {
                this.wait();
            } catch (InterruptedException e) {
            }
        }
        c = buffer.remove(buffer.size()-1);
        return c;
    }

    public synchronized void push(char c) {
        this.notify();
        buffer.add(c);
    }
}
```

java.util.concurrent



- Це тема наступної лекції

Література



- **The Java Tutorial.** http://download.oracle.com/javase/tutorial/essential/concurrency/index.html
- **SL-275. Java Programming Language**. – Sun Microsystems. - 2007.
- Брюс Эккель. **Философия Java**. – Питер, 2008. – 640 с.
- Майкл Морган. **Java 2. Руководство разработчика.** – М: Вильямс, 2000. – 720с.
- Кей С. Хорстманн, Гари Корнелл. **Java 2. Библиотека профессионала. Том 1. Основы.** – Вильямс, 2007. – 896с.