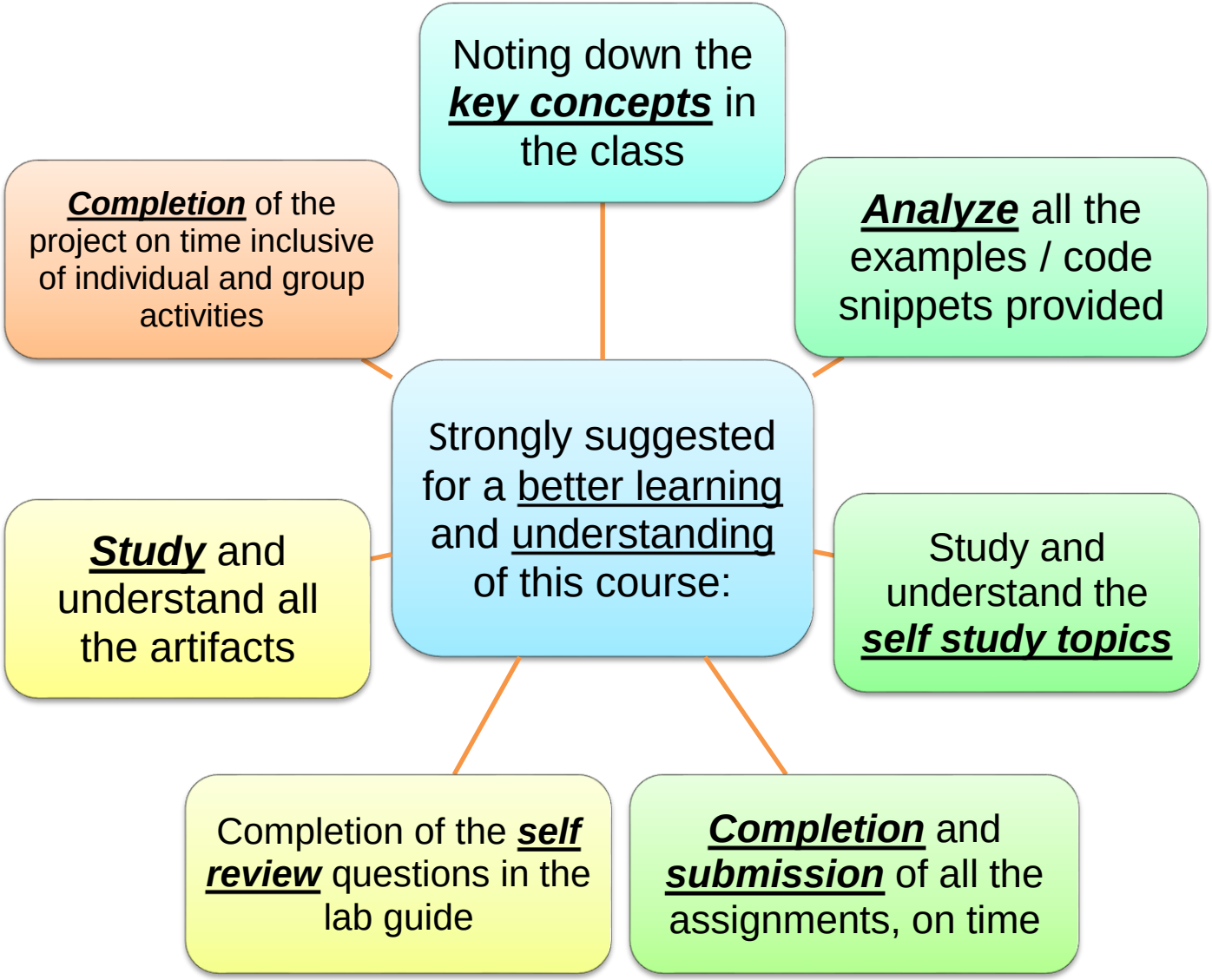
MULTITHREAD & THREAD SYNCHRONIZATION

Instructor:

Table of contents

- Multithread
  - ✓ Multi-Thread vs. Multitasking
  - ✓ Practical time
- Thread Synchronization

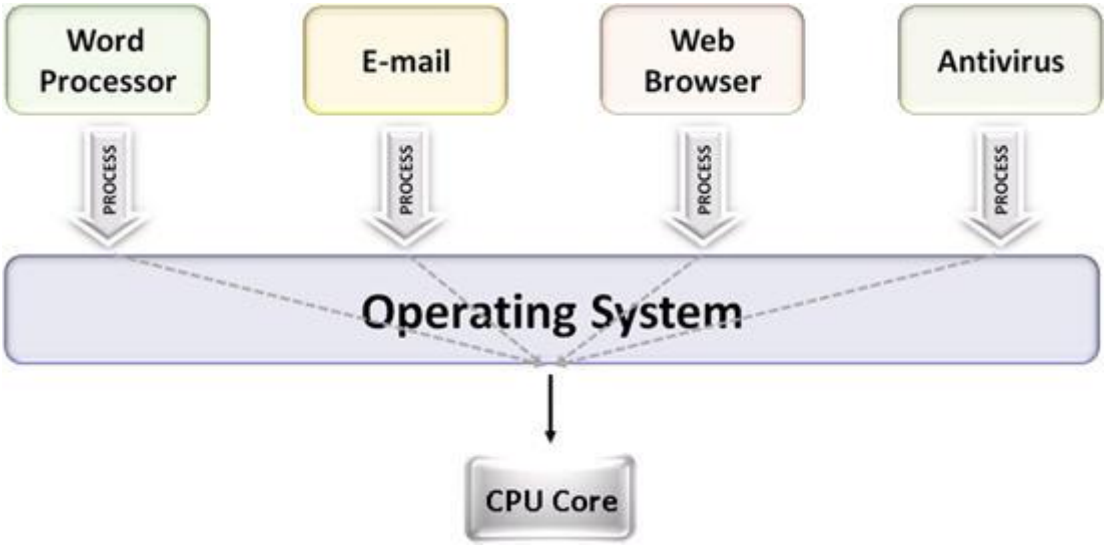
Learning Approach



Section 1

Multitasking

- Multitasking is the ability of the operating system to perform two or more tasks concurrently.
- Multitasking can be process-based or thread-based.



Thread & Multi-threading

- A thread performs a certain task and is the smallest unit of executable code in a program.
- Multithreading can be defined as the concurrent running of the two or more parts of the same program.

Multithreading vs. Multitasking-process based

Multithreading	Multitasking process based
In a multithreaded program two or more **threads can run concurrently.**	In a multitasking environment two or more **processes run concurrently.**
Multithreading requires **less** overhead[chi phí].	Multitasking requires **more** overhead.
Threads are lightweight processes.	Processes are heavyweight tasks that require their own address space.
Threads can share same address space and inter-thread communication is **less expensive** than inter-process communication.	Interprocess communication is very expensive and the context switching from one process to another is costly

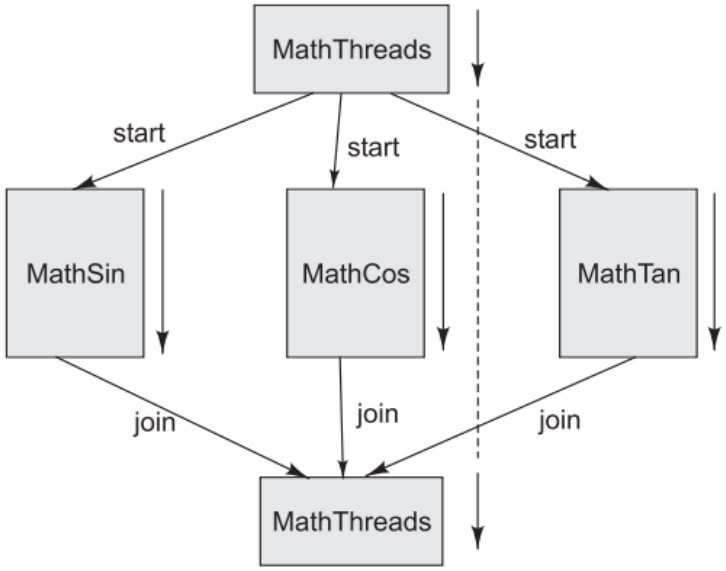
Need for Multithreading

- To increase performance of single-processor systems, as it reduces the CPU idle time.
- Faster execution of a program when compared to an application with multiple processes.
- Parallel processing of multiple threads in an application which services a huge number of users.

Practical time

- To illustrate creation of multiple threads in a program performing concurrent operations, let us consider the processing of the following mathematical equation:

$$p = \sin(x) + \cos(y) + \tan(z)$$



Flow of control in a master and multiple workers threads application

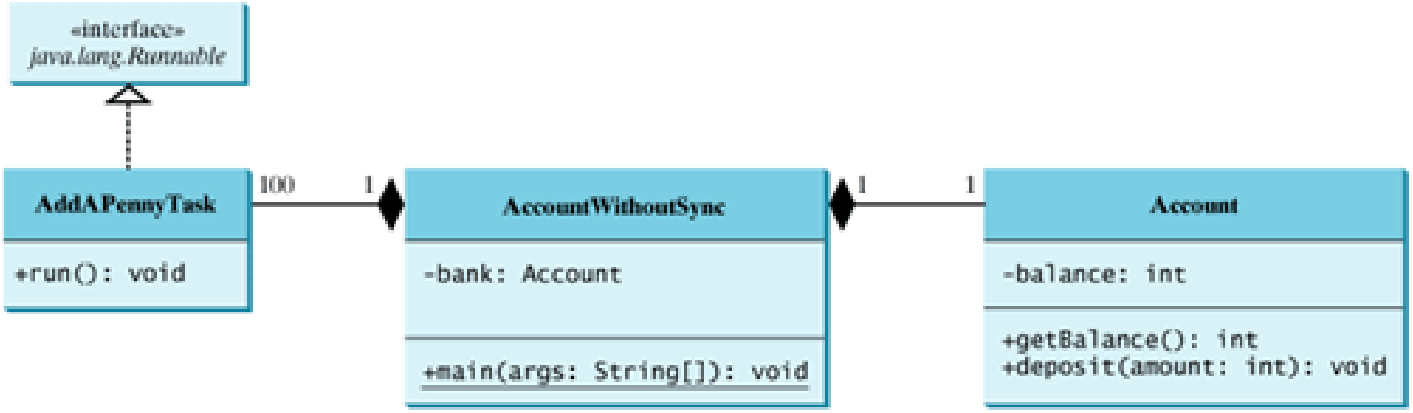
Section 2

Overview

- What happens when **two different threads** have **access to a single instance** of a class, and both threads invoke methods on that object…and those methods modify the state of the object?
- **Example,**
  - ✓ Imagine that two people each have a checkbook for a single checking account (or two people each have ATM cards, but both cards are linked to only one account).
  - ✓ We have a class called Account that represents a bank account. This account starts with a balance of 50, and can be used only for withdrawals. The withdrawal will be accepted even if there isn't enough money in the account to cover it.
  - ✓ Imagine what would happen if Lucy checks the balance and sees that there's just exactly enough in the account, 10. *But before she makes the withdrawal, Fred checks the balance and also sees that there's enough for his withdrawal.*
  - ➔ This problem is known as a "**race condition**,"

Thread synchronization

- Several threads may simultaneously try to **update the same resource**, such as a file. This leaves the resource in an **undefined** or **inconsistent**[không nhất quán] state. This is called **race condition**.
- In general, race conditions in a program occur when
  - Two or more threads share the same data between them.
  - Two or more threads try to read and write the shared data simultaneously.



Technical

- So how do you protect the data? You must do two things:
  - ✓ Mark the **variables private**.
  - ✓ **Synchronize** the code that modifies the variables.

Synchronized method

- **Synchronized method** is used to **lock an object** for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Code snippet

```
public class Data {
   synchronized void printData(int n) {
        // synchronized method
       for (int i = 1; i <= 5; i++) {
           System.out.println(n * i);
           try {
               Thread.sleep(400);
           } catch (Exception e) {
               System.out.println(e);
           }
       }
   }
}
```

Synchronized block

- Synchronized blocks are **used to prevent the race conditions.**
- A lock allow only one thread at a time to access the code.
- Points to remember for Synchronized block:
  - ✓ Synchronized block is used to lock an object for any shared resource.
  - ✓ Scope of synchronized block is smaller than the method.
- **Syntax:**

```
synchronized (object reference expression) {
    //code block
}
```

- **Example:** see **TestSynchronizedBlock.java**

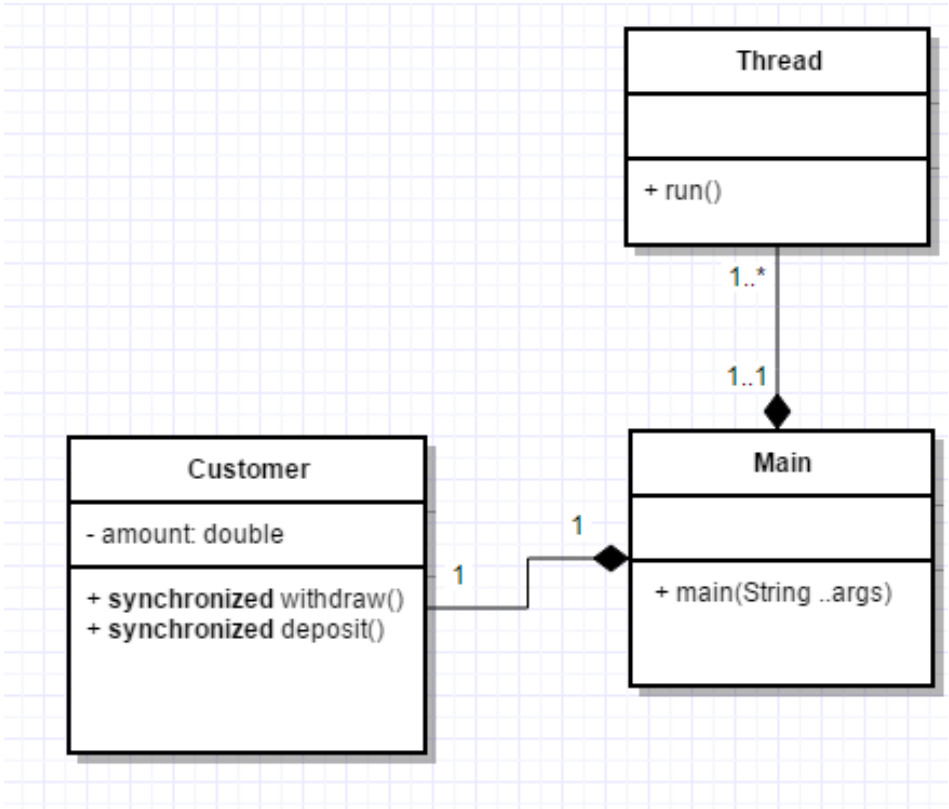
Wait-notify mechanism

- Ensures smooth transition of a particular resource between 2 competitive thread.
- Allowed to wait for the lock of a-synchronized block of resource currently used by another thread.
- Notified to end its waiting state and get the lock of that synchronized block of resource.

`wait()` and `notify()` methods

- They are methods of Object class:
  - ✓ `wait()`: methods provide a way for a shared object to pause a thread when it becomes unavailable to that thread and the calling thread gives up the CPU and lock
  - ✓ `notify()`: methods provide a way for a shared object to allow the thread to continue when appropriate.
- wait – notify mechanism: Demo

- Sử dụng wait() và notify()



interrupt() Method

- An **interrupt** is an indication to a thread that it should stop what it is doing and do something else.
- An interrupted thread can **die, wait** for another task or **go to next step** depending on the requirement of the application:
  - ✓ If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state **throwing InterruptedException (die or wait case)**.
  - ✓ If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behavior **(go to next step)**.

interrupt() Method

Example 1

- Interrupting a thread that stops working:

Code snippet

```
public class InterruptSample1 extends Thread {
    public void run() {
        try {
         Thread.sleep(1000);
         System.out.println("Task");
        } catch (InterruptedException e) {
            throw new RuntimeException("Thread interrupted..." +e);
        }
    }
    public static void main(String args[]) {
        InterruptSample1 t1 = new InterruptSample1();
        t1.start();
        try {
         t1.interrupt();
        } catch (Exception e) {
         System.out.println("Exception handled " + e);
        }
    }
}
```

interrupt() Method

Example 2

- Interrupting a thread that doesn't stops working:

Code snippet

```
public class InterruptSample2 extends Thread {
   public void run() {
   try {
       Thread.sleep(1000);
       System.out.println("task");
   } catch (InterruptedException e) {
       System.out.println("Exception handled " + e);
   }
       System.out.println("thread is running...");
   }
   public static void main(String args[]) {
       InterruptSample2 t2 = new InterruptSample2();
       t2.start();
       t2.interrupt();
   }
}
```

interrupt() Method

Example 3

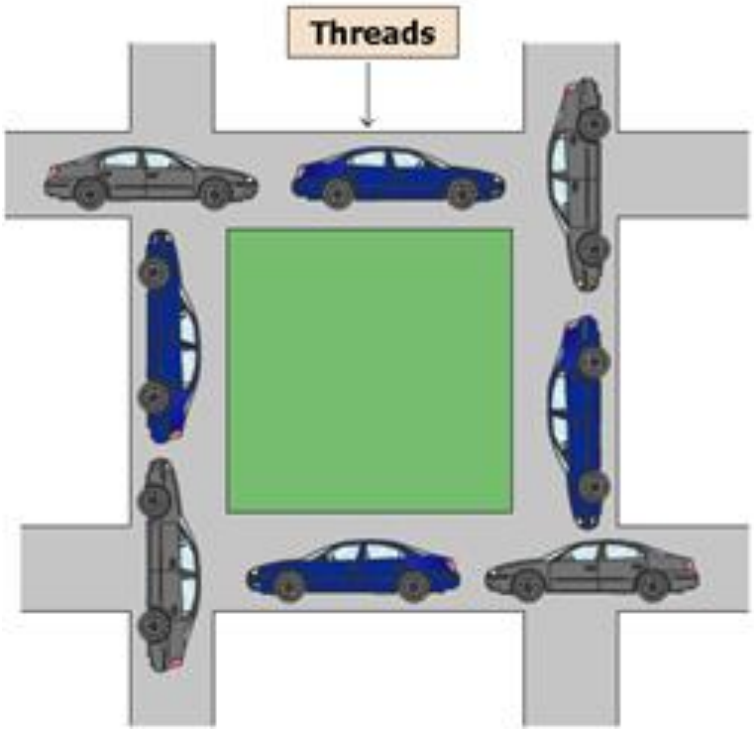
- Interrupting thread that behaves normally:

```
public class InterruptSample3 extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++)
         System.out.println(i);
    }

    public static void main(String args[]) {
        InterruptSample3 t3 = new InterruptSample3();
        t3.start();
        t3.interrupt();
}}
```

Deadlock

- Deadlock describes a situation where two or more threads are block forever, waiting for the other to release a resource.



Overcoming of Deadlock

Prevention of deadlock :

- Avoid acquiring more than one lock at a time.
- Ensure that in a Java program, you acquire multiple locks in a consistent and defined order.

Overcoming of Deadlock

- If a thread holds a lock and goes in **sleeping** state, it does **not loose the lock**. However, when a thread goes in the **blocked** state, it **releases the lock**. This eliminates potential deadlock situations.
- Java does not provide any mechanism for detection or control of potential deadlock situations. The programmer is responsible for avoiding them.

Lesson summary

- Multithreading
  - ✓ Multithread vs. Multitasking
  - ✓ Practical time
- Thread synchronization
  - ✓ Synchronized method
  - ✓ Synchronized block
  - ✓ Wait/Notify mechanism

Thank you