# ใบงานการทดลองที่ 13

## เรื่อง การใช้งาน Inner Class และการใช้งาน Thread

**1. จุดประสงค์ทั่วไป**

1.1. รู้และเข้าใจการโปรแกรมเชิงวัตถุ การกำหนดวัตถุ การใช้วัตถุ

1.2. รู้และเข้าใจการทำหลายงานพร้อมกัน

**2. เครื่องมือและอุปกรณ์**

เครื่องคอมพิวเตอร์1 เครื่อง ที่ติดตั้งโปรแกรม Eclipse

**3. ทฤษฎีการทดลอง**

3.1. Nest Class คืออะไร? มีวัตถุประสงค์เพื่ออะไร? อธิบายพร้อมยกตัวอย่างประกอบ

- เป็น Class ที่ประกาศภายใน body ของ Class หรือ Interface อื่นๆ

การ group Class และ Interface ที่เกี่ยวข้องกันให้อยู่ภายใน File เดียวกัน ถึงแม้ว่าการทำ Package ก็ช่วยในเรื่องดังกล่าวแล้วแต่การทำ Nested Classes ทำให้การ group แข็งแรงมากขึ้นอีกขั้น

- Nested Classes จะถือว่าเป็นสมาชิกของ Class ที่ล้อมรอบมัน ดังนั้นเราสามารถที่จะระบุ Access Modifier ให้มันได้ และ Nested Classes นี้ยังถูกสืบทอดไปให้ SubClass อีกทั้งยังใส่ Abstract หรือ final ให้มันได้อีกด้วย

3.2. จงยกตัวอย่างการสร้าง Inner Class

```
private int x;
public class InnerClass { public void printX() {
 System.out.println("x = " + x); } }
public void createInnerClass() { InnerClass inner = new InnerClass();
 inner.printX(); }}
```

3.3. จงยกตัวอย่างการเรียกใช้งาน Instance ที่มีการเรียกใช้งาน Properties ภายใน Inner Class

```
public class OuterClass {
   private int x; public class InnerClass {
    public void printX() { System.out.println("x = " + x);
}
```

3.4. จงยกตัวอย่างการเรียกใช้งาน Instance ที่มีการเรียกใช้งาน Method ภายใน Inner Class

```
public class OuterClass {
private int x; public class InnerClass {
public void printX() {
System.out.println("x = " + x); }
}
```

3.5. Thread คืออะไร? มีประโยชน์อย่างไร? อธิบายพร้อมยกตัวอย่างประกอบ

- Thread คือเทรด (thread) ที่ใช้ในการทำงานแบบพร้อมกันกับเทรดอื่นๆ ในโปรแกรม หรือก็คือเทรดเป็นเหมือนเส้นทางที่มีการทำงานของ โปรแกรมซึ่งแตกต่างจากกระบวนการ (process) ที่เป็นเหมือนห้องทดลองหนึ่งห้องที่มีสายงานหลายสายงานแต่แตกต่างกันตรงที่กระบวนการมีการ จัดการแยกกันของหน่วยงานเป็นก้อนๆ ทำให้การทำงานของส่วนต่างๆ ของโปรแกรมไม่สามารถเข้าถึงหรือแชร์กันได้ในขณะที่ Thread สามารถใช้ แบ่งแยกและประมวลผลงานได้พร้อมกัน

3.6. การเริ่มต้นใช้งาน Thread มีขั้นตอนอย่างไรบ้าง?

```
public class MyThread extends Thread {
    public void run() {
        // โค้ดที่ต้องการให้ Thread ทำงาน
    }
}

MyThread myThread = new MyThread();
myThread.start();
```

3.7. ระหว่าง Thread และ Runnable มีรูปแบบการใช้งานที่เหมือนหรือแตกต่างกันอย่างไร?

- Thread และ Runnable มีรูปแบบการใช้งานที่แตกต่างกันอย่างมาก โดยส่วนใหญ่แล้วการสร้าง Thread จะเป็นการสร้าง Object ของคลาส Thread และนำ Runnable มาใช้เป็นพารามิเตอร์ของ Thread

3.8. สถานะ Deadlock มีลักษณะเป็นอย่างไร? อธิบายพร้อมยกตัวอย่างประกอบ

- Deadlock คือสถานะที่เกิดขึ้นเมื่อสองหรือมากกว่าสอง Thread หรือโปรเซสต้องการเข้าถึงทรัพยากรเดียวกัน และไม่สามารถปล่อยทรัพยากรที่ถูกจองไว้ให้ Thread หรือโปรเซสอื่นใช้ได้ ทำให้ทั้งหมดติดอยู่ในสถานะ blocked โดยเป็นอย่างน้อย 2 Thread ที่ถูก block และไม่สามารถดำเนินการต่อไปได้
- มีลักษณะเป็นวงกลมและเกิดขึ้นเมื่อมีการล็อก (lock) ทรัพยากรที่สอง Thread หรือโปรเซสต้องการเข้าถึงเดียวกัน โดยทั้งสอง Thread จะรอกันไม่รู้จักว่าอีกฝั่งจะปล่อยทรัพยากรออกมาหรือไม่ ทำให้เกิดสถานการณ์ที่ Thread หรือโปรเซสทั้งสองไม่สามารถดำเนินการต่อไปได้

Thread 1:

```
synchronized(A) {
    synchronized(B) {
        // โค้ดที่ต้องการทำงาน
    }
}
```

Thread 2:

```
synchronized(B) {
    synchronized(A) {
        // โค้ดที่ต้องการทำงาน
    }
}
```

4. ลำดับขั้นการปฏิบัติการ

4.1. จงสร้างหน้า GUI เพื่อทำการทดสอบสร้าง Thread ที่มีส่วนประกอบดังต่อไปนี้

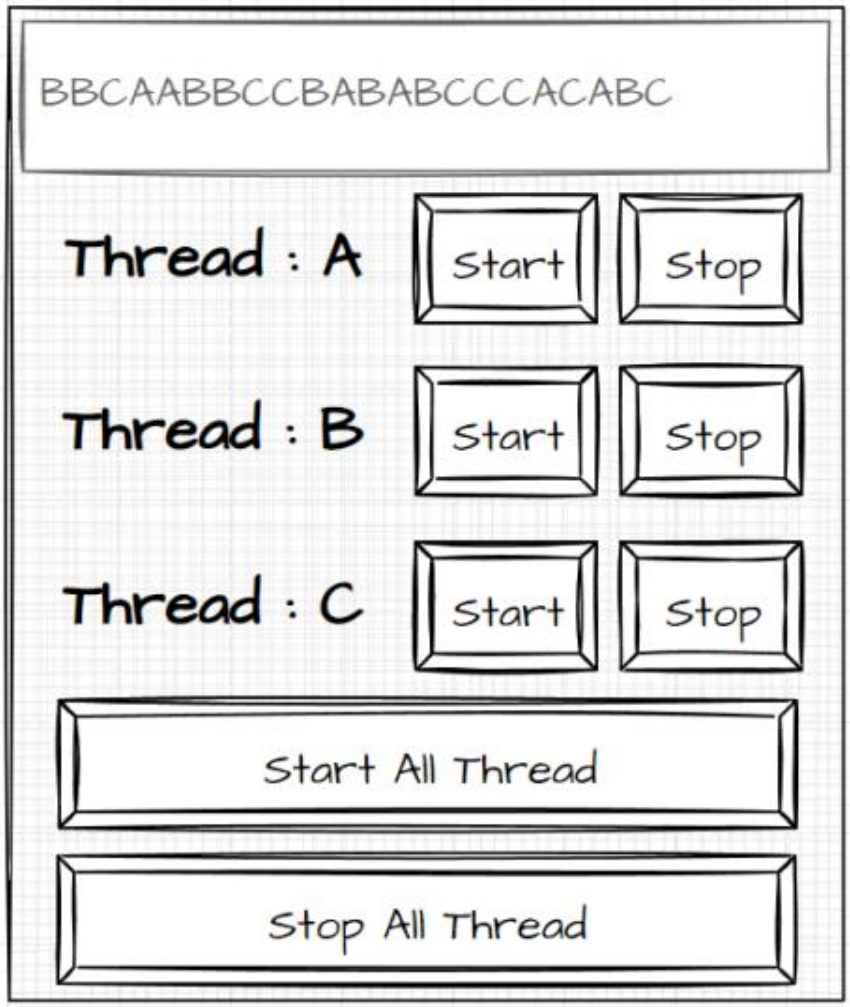
4.1.1. สร้าง Thread A ที่สร้างจาก Inner Class

4.1.2. สร้าง Thread B และ C จาก Class ปกติ

4.1.3. แต่ละ Thread จะมีปุ่ม Start เพื่อเริ่มต้นพิมพ์ตัวอักษรของ Thread ลงในช่อง Textbox และ Stop เพื่อหยุดการพิมพ์ตัว อักษรของ Thread ในช่อง Textbox

4.1.4. สร้างปุ่ม Start All Thread เพื่อทำให้Thread แต่ละตัวทำงานพร้อมกัน

4.1.5. สร้างปุ่ม Stop All Thread เพื่อให้Thread แต่ละตัวหยุดทำงานพร้อมกัน




\

| โค้ดโปรแกรมของปุ่ม Start และ Stop ของ Thread A |
|---|

```
private class ThreadA implements Runnable {
    private Thread thread;
    private volatile boolean isRunning = false;
    public void startThread() {
        isRunning = true;
        thread = new Thread(this);
        thread.start();
    }
    public void stopThread() {
        isRunning = false;
}
```

| โค้ดโปรแกรมของปุ่ม Start และ Stop ของ Thread B |
|---|

```
private class ThreadB extends Thread {
    private volatile boolean isRunning = false;

    public void startThread() {
        isRunning = true;
        start();
    }
    public void stopThread() {
        isRunning = false;
    }
    public void run() {
        while (isRunning) {
        textArea.append("B");
```

```
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

โค้ดโปรแกรมของปุ่ม Start และ Stop ของ Thread C

```
private class ThreadC extends Thread {
    private volatile boolean isRunning = false;
    public void startThread() {
    isRunning = true;
    start();
    }
    public void stopThread() {
    isRunning = false;
    }
    public void run() {
        while (isRunning) {
            textArea.append("C");
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
}
```

โค้ดโปรแกรมของปุ่ม Start All Thread

```
public void actionPerformed(ActionEvent e) {
        if (e.getSource() == startAllBtn) {
                threadA.startThread();
                threadB.startThread();
                threadC.startThread();
        } else if (e.getSource() == stopAllBtn) {
                threadA.stopThread();
                threadB.stopThread();
                threadC.stopThread();
        }
}
```

โค้ดโปรแกรมของปุ่ม Stop All Thread

```
        } else if (e.getSource() == stopAllBtn) {
        threadA.stopThread();
        threadB.stopThread();
        threadC.stopThread();
        }
}
```

5. สรุปผลการปฏิบัติการ

- การทำงานในชุดคำสั่งสามารถแทรกแทรงเข้าหากันได้

6. คำถามท้ายการทดลอง

6.1. Inner Class แตกต่างจาก Class แบบปกติอย่างไร?

- การเข้าถึง: Inner Class สามารถเข้าถึงตัวแปรและเมธอดของ Outer Class ได้โดยตรง ในขณะที่ Outer Class ไม่สามารถเข้าถึงตัวแปรและเมธอดของ Inner Class ได้โดยตรง
- การใช้งาน: Inner Class สามารถเรียกใช้งานตัวแปรและเมธอดของ Outer Class ได้โดยตรง ซึ่งช่วยให้ Inner Class สามารถเข้าถึงและปรับเปลี่ยนค่าของตัวแปรใน Outer Class ได้ ในขณะที่ Outer Class ไม่สามารถเรียกใช้งานตัวแปรและเมธอดของ Inner Class ได้โดยตรง
- การสร้าง: Inner Class สามารถสร้างได้โดยไม่ต้องมีการประกาศอย่างชัดเจน ในขณะที่ Outer Class จะต้องประกาศโดยชัดเจน
- ความสามารถ: Inner Class สามารถเข้าถึง private ตัวแปรและเมธอดของ Outer Class ได้ ในขณะที่ Outer Class ไม่สามารถเข้าถึง private ตัวแปรและเมธอดของ Inner Class ได้

6.2. เมื่อใดจึงเป็นช่วงเวลาที่ดีที่สุดในการใช้งาน Inner Class

-การใช้งาน Inner Class เพื่อเข้าถึงตัวแปรและเมธอดของ Outer Class: Inner Class สามารถเข้าถึงตัวแปรและเมธอดของ Outer Class ได้โดยตรง ซึ่งเหมาะสำหรับการใช้งานในสถานการณ์ที่ต้องการใช้งานตัวแปรหรือเมธอดของ Outer Class ใน Inner Class
- การใช้งาน Inner Class เพื่อสร้างคลาสที่มีความซับซ้อน: Inner Class สามารถใช้เพื่อสร้างคลาสที่มีความซับซ้อนได้ โดยที่ไม่จำเป็นต้องประกาศคลาสอื่นขึ้นมาเพิ่มเติม ซึ่งเหมาะสำหรับการใช้งานในสถานการณ์ที่ต้องการสร้างคลาสที่มีลักษณะพิเศษ หรือมีการเชื่อมโยงกับคลาสอื่น
- การใช้งาน Inner Class เพื่อเข้าถึง private ตัวแปรและเมธอดของ Outer Class: Inner Class สามารถเข้าถึง private ตัวแปรและเมธอดของ Outer Class ได้ ซึ่งเหมาะสำหรับการใช้งานในสถานการณ์ที่ต้องการเข้าถึง private ตัวแปรและเมธอดของ Outer Class ใน Inner Class
- การใช้งาน Inner Class เพื่อการสืบทอด: Inner Class สามารถใช้เพื่อสืบทอดคุณสมบัติจากคลาสอื่น ๆ ได้ ซึ่งเหมาะสำหรับการใช้งานในสถานการณ์ที่ต้องการสร้างคลาสที่ม

6.3. ข้อควรระวังในการใช้งาน Thread คืออะไร?

- Race Condition: เป็นการแข่งขันกันในการเข้าถึงข้อมูลหรือแชร์แหล่งข้อมูลร่วมกัน ทำให้เกิดปัญหาในการอ่านหรือเขียนข้อมูลได้ผลลัพธ์ไม่ถูกต้อง หรือเกิดปัญหาเซ็งเซพในกรณีที่ Thread ต่างกันต้องการเข้าถึงข้อมูลเดียวกัน
- Deadlock: เกิดขึ้นเมื่อสองหรือมากกว่า Thread มีการถือครองทรัพยากรและต้องการทรัพยากรของกันและกัน ทำให้เกิดปัญหาเกี่ยวกับการรอคอยทรัพยากรและเป็นปัญหาที่ยากต่อการตรวจหาและแก้ไข
- Starvation: เกิดขึ้นเมื่อ Thread ไม่ได้รับการพิจารณาส่วนใหญ่ในการแบ่งปันทรัพยากร ทำให้ Thread นั้นๆ ต้องรอคอยการเข้าถึงทรัพยากรอยู่เสมอ และอาจจะไม่ได้รับการใช้งานเลย
- Context Switching Overhead: เป็นการสลับการทำงานระหว่าง Thread ที่ใช้ทรัพยากรของ CPU ซึ่งการสลับนี้อาจเป็นเรื่องที่ใช้เวลาและทรัพยากรในการทำงาน
- Memory Synchronization: เป็นการจัดการกับการแชร์หน่วยความจำร่วมกัน ทำให้เกิดปัญหาการเข้าถึงข้อมูล