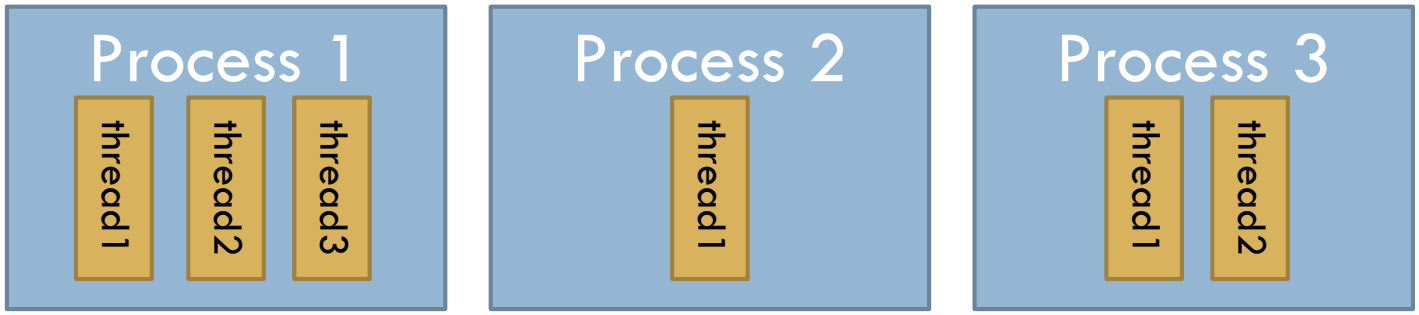
JAVA THREAD

030523313 - Network programming

Asst. Prof. Dr. Choopan Rattanapoka

อะไรคือ Thread

- ในระบบปฏิบัติการสมัยใหม่ เวลาเรียกใช้งานโปรแกรมก็คือก็สร้าง **process** ของโปรแกรมนั้น
- Process คือ โปรแกรมที่กำลังทำงานอยู่ในระบบ
- แต่ละ Process จะมีอย่างน้อย 1 Thread ที่ทำงานอยู่
- Thread เรียกอีกอย่างว่า **lightweight process**



Multiple Threads ?

- แล้วทำไมถึงต้องมีการใช้ Thread หลาย Thread ใน process เดียว
  - การตอบสนองของโปรแกรมที่ดีกับผู้ใช้
    - โปรแกรม GUI เช่น winzip ขณะที่ทำการบีบอัดแฟ้มข้อมูลถ้าไม่มี Thread ตัวหน้าโปรแกรม GUI จะค้างจนกว่าการบีบอัดแฟ้มข้อมูลจะเสร็จสิ้น
  - สำหรับงานที่สามารถทำพร้อมกันได้
    - โปรแกรมเซิฟเวอร์ เช่น Web server สามารถรองรับผู้ใช้หลายคนพร้อมกันได้
  - ใช้ประโยชน์จาก CPU แบบ multicore, หรือ Multiple CPU อย่างเต็มประสิทธิภาพ
    - โปรแกรมเช่น สร้างภาพ 3 มิติ สามารถใช้หลาย Thread ช่วยกันประมวลผลได้

การใช้งาน Thread

- Thread มีข้อดีหลายอย่าง แต่ก็ไม่เหมาะสำหรับงานบางประเภท
- การใช้ Thread เป็นการเพิ่มการใช้งานทรัพยากรของระบบ
  - RAM ที่ใช้เก็บตัวแปรต่างๆ ของแต่ละ thread
  - Overhead ของการทำ context switch ของ CPU
- ตัวอย่าง โปรแกรมที่ไม่จำเป็นต้องใช้ Thread
  - โปรแกรมตรวจสอบ e-mail ทุกๆ 5 นาที
  - เราสามารถจะใช้ 1 thread หลับรอแล้วตื่นขึ้นทุกๆ 5 นาทีเพื่อตรวจสอบ e-mail ซึ่งจะดีกว่า สร้าง thread ใหม่ทุกครั้งเมื่อครบ 5 นาที
- Thread เป็นเรื่องง่ายในการเริ่มต้นใช้งาน แต่ยากมากที่จะชำนาญ

การสร้าง Thread ใน Java

- การสร้างและใช้งาน Thread ในภาษา Java มีด้วยกัน 2 วิธี
  - สร้าง class ที่ extends มาจาก Class Thread
    - เป็นวิธีที่ง่ายที่สุดในการเขียนโปรแกรมเพื่อใช้ Thread ในภาษา Java
  - สร้าง class ที่ implements Runnable Inteface
    - เป็นวิธีที่ถูกใช้งานมากที่สุดในโปรแกรมทั่วไป

Extend จาก Class java.lang.Thread

- การสร้าง Class ที่สามารถทำงานเป็น Thread ให้เพิ่มคำว่า **extends Thread** ไปข้างหลังชื่อ Class

```
public  class TwoThread extends  Thread {
                ....
                ....
}
```

- จะต้องทำการ Override เมธอดที่ชื่อว่า run( )
  - public void run( ) { }
  - เมธอดนี้เป็นจุดเริ่มต้นการทำงานของ Thread

ตัวอย่างการสร้าง Class ที่สามารถทำ Thread

```
import java.io.*;

public class TwoThread extends Thread {
        public void run( ) {
                for(int i = 0; i < 10; i++) {
                        System.out.println("New Thread");
                }
        }
}
```

- Class TwoThread มีการ extends Thread
- มีการ override เมธอด run( )
- การทำงานของ Thread จะวนลูป 10 รอบเพื่อพิมพ์คำว่า “New Thread”

การเรียกใช้งาน Thread

- การสร้าง Thread ขึ้นในโปรแกรม (Spawning) จะต้องสร้างขึ้นจาก Thread ที่กำลังทำงานอยู่
- การสร้าง Thread นั้นเหมือนกับการสร้าง Object ปกติในภาษาจาวา

```
TwoThread  tt  =  new TwoThread( );
```

- เมื่อต้องการให้ Thread ทำงาน ก็เรียกใช้เมธอด start( ) ของ Object นั้น

```
tt.start();
```

- หลังจากนั้น Thread จะเริ่มทำงานในเมธอด run( ) ในขณะที่ผู้เรียกก็ทำงานคำสั่งถัดไปต่อได้ทันที

ตัวอย่างการเรียกใช้งาน Thread

```
import java.io.*;

public class TwoThread extends Thread {
        public void run( ) {
                for(int i = 0; i < 10; i++) {
                        System.out.println("New Thread");
                }
        }

        public static void main(String[] args) {
                TwoThread tt = new TwoThread();
                tt.start();

                for(int i = 0; i < 10; i++) {
                        System.out.println("Main Thread");
                }
        }
}
```

คิดว่าผลการรันจะเป็นอย่างไร

การ delay หรือ sleep ในภาษาจาวา

- ในการทำงานบางอย่าง โปรแกรมจำเป็นต้องมีการ delay หรือ sleep รอ

```
long startTime = System.currentTimeMillis();
long stopTime  = System.currentTimeMillis()+6000;
while(System.currentTimeMillis() < stopTime) { }
```

- การทำแบบนี้จะให้ CPU ทำงานตลอดเวลา และสิ้นเปลืองทรัพยากร
- ถ้าต้องการให้โปรแกรม delay หรือ sleep จะต้องสั่งผ่าน Class Thread

Thread.sleep(เวลาเป็น ms);

- เนื่องจากเมธอด sleep จะมีการโยน Exception ดังนั้นต้องนำ try-catch มาครอบคำสั่งเอาไว้

ตัวอย่างการใช้ Sleep

```
import java.io.*;

public class TestSleep {
        public static void main(String[] args) {
                System.out.println("Hello");

                try {
                        Thread.sleep(3000);
                } catch (Exception e) {}

                System.out.println("Bye Bye");
        }
}
```

ตัวอย่างการทำงานหลาย Thread กับ Sleep

```
import java.io.*;

public class MultiThread extends Thread {
        String myName;
        long sleepTime;

        public MultiThread(String myName, long sleepTime) {
                this.myName    = myName;
                this.sleepTime = sleepTime;
        }

        public void run() {
                for(int i = 0; i < 5; i++) {
                        System.out.println(myName);
                        try {
                                Thread.sleep(sleepTime);
                        } catch(Exception e) {}
                }
        }

        public static void main(String[] args) {
                MultiThread t1 = new MultiThread("-1-", 1000);
                MultiThread t2 = new MultiThread("-2-", 2000);
                MultiThread t3 = new MultiThread("-3-", 3000);

                t1.start();
                t2.start();
                t3.start();
        }
}
```

ผลการรัน

```
-1-
-3-
-2-
-1-
-2-
-1-
-3-
-1-
-2-
-1-
-3-
-2-
-2-
-3-
-3-
```

ข้อเสียของการใช้ Extends Thread

- ในการใช้งาน Thread ผ่านวิธี extends Thread นั้นใช้งานง่าย
- แต่มีข้อจำกัดของภาษาจาวา **ที่กัน** การสืบทอดจากหลาย Class
  - ภาษาจาวาอนุญาตให้มีการสืบทอด Class มาจาก Class แม่เพียง Class เดียวเท่านั้น ซึ่งเป็นการแก้ปัญหาที่เกิดขึ้นจากการสืบทอด Class หลาย Class จากภาษา C++
- ทำให้การทำใช้งานบางประเภทไม่สามารถทำได้
  - เช่น Class ที่ทำ GUI เช่น JComponent โปรแกรมจำเป็นต้องเขียน extends JComponent เพื่อใช้งาน ทำให้ไม่สามารถเขียน extends Thread ได้
- วิธีแก้ไขคือต้องเขียน Thread ด้วยวิธีที่ 2 คือ implements Runnable

การสร้าง Thread ด้วยวิธี implements Runnable

- การสร้าง Class ที่สามารถทำงานเป็น Thread ให้เพิ่มคำว่า **implements Runnable** ไปข้างหลังชื่อ Class

```
public  class TwoThread implements  Runnable {
                ....
                ....
}
```

- จะต้องทำการ Override เมธอดที่ชื่อว่า run( ) [เหมือนวิธี extends]
  - public void run( ) { }
  - เมธอดนี้เป็นจุดเริ่มต้นการทำงานของ Thread

ตัวอย่างการสร้าง Class ที่สามารถทำ Thread

```
import java.io.*;

public class TwoThread implements Runnable {
        public void run( ) {
                for(int i = 0; i < 10; i++) {
                        System.out.println("New Thread");
                }
        }
}
```

- Class TwoThread มีการ implements Runnable
- มีการ override เมธอด run( )
- การทำงานของ Thread จะวนลูป 10 รอบเพื่อพิมพ์คำว่า “New Thread”

การเรียกใช้งาน Thread จาก Class ที่ implements Runnable

- สร้าง Object ของ Class นั้นตามปกติ

```
TwoThread  tt  =  new TwoThread( );
```

- สร้าง Object Thread สำหรับ Class นั้น

```
Thread   t  =  new Thread(tt);
```

- เรียกใช้งาน Thread ผ่าน Object Thread ที่สร้างขึ้น

```
t.start();
```

- หลังจากนั้น Thread จะเริ่มทำงานในเมธอด **run( )** ในขณะที่ผู้เรียกก็ทำงานคำสั่งถัดไปต่อได้ทันที

ตัวอย่างการเรียกใช้งาน Thread

```
import java.io.*;

public class TwoThread implements Runnable {
        public void run( ) {
                for(int i = 0; i < 10; i++) {
                        System.out.println("New Thread");
                }
        }

        public static void main(String[] args) {
                TwoThread tt = new TwoThread();
                Thread t = new Thread(tt);
                t.start();

                for(int i = 0; i < 10; i++) {
                        System.out.println("Main Thread");
                }
        }
}
```

ข้อแตกต่างของ 2 วิธีในการสร้างและใช้งาน Thread

(1) วิธี extends Thread

```
import java.io.*;

public class TwoThread extends Thread {
        public void run( ) {
                for(int i = 0; i < 10; i++) {
                        System.out.println("New Thread");
                }
        }

        public static void main(String[] args) {
                TwoThread tt = new TwoThread();
                tt.start();

                for(int i =  ; i < 10; i++) {
                        Sys m.out.println("Main Thr
                }
        }
}
```

การเขียน Class

(1) extends Thread

(2) implements Runnable

```
import java.io.*;

public class TwoThread implements Runnable {
        public void run( ) {
                for(int i = 0; i < 10; i++) {
                        System.out.println("New Thread");
                }
        }

        public static void main(String[] args) {
                TwoThread tt = new TwoThread();
                Thread t = new Thread(tt);
                t.start();

                for(int i = 0; i < 10; i++) {
                        System.out.println("Main Thread");
                }
        }
}
```

การสร้างและเรียกใช้งาน Object ของ Thread

(2) วิธี implements Runnable

ตัวอย่าง: โปรแกรมหา summation

```
import java.io.*;

public class Sum {
        int from;
        int where;
        int result = 0;

        public Sum(int from, int where) {
                this.from  = from;
                this.where = where;
        }

        public void run() {
                for(int i = from; i <= where; i++) {
                        result += i;
                }
        }

        public int getResult() {
                return result;
        }


        public static void main(String[] args) {
                Sum s = new Sum(0, 1000000);
                s.run();
                System.out.println("Result = " + s.getResult());
        }
}
```

นำ Thread มาประยุกต์ใช้

- จากตัวอย่าง เป็นการหา Summation ของตัวเลข 0-1000000 โดยใช้
- เราสามารถใช้นำ Thread มาประยุกต์ใช้ได้ เช่น
  - แบ่งการทำงานของ Summation ของเป็น 2 ส่วน คือ
  - Summation ของ 0 – 499999 (Thread 1)
  - Summation ของ 500000 – 1000000 (Thread 2)
  - จากนั้นนำผลลัพธ์ของทั้ง 2 Thread มารวมกันเป็นคำตอบ
- **คำเตือน** ประสิทธิภาพที่ได้จากการแบ่งการทำงานเป็น 2 Thread ของ Summation นี้ เวลาที่ใช้ประมวลผลจะไม่แตกต่างกันมากนัก แต่ถ้าเปลี่ยนจากการทำ Summation เป็นการประมวลผลอย่างอื่นที่ใช้เวลานานๆ การทำงานแบบ Multi-thread จะใช้เวลาน้อยกว่า

การนำ Thread มาประยุกต์ใช้

```java
import java.io.*;

public class SumThreadWrong implements Runnable {
        int from;
        int where;
        int result = 0;

        public SumThreadWrong(int from, int where) {
                this.from  = from;
                this.where = where;
        }

        public void run() {
                for(int i = from; i <= where; i++) {
                        result += i;
                }
        }

        public int getResult() {
                return result;
        }
```

```java
        public static void main(String[] args) {
                int s = 0;
                SumThreadWrong s1 = new SumThreadWrong(0, 499999);
                SumThreadWrong s2 = new SumThreadWrong(500000, 1000000);
                Thread t1 = new Thread(s1);
                Thread t2 = new Thread(s2);
                try {
                        t1.start();  t2.start();
                        s = s1.getResult() + s2.getResult();
                } catch(Exception e){}
                System.out.println("Result = " + s);
        }

}
```

มีปัญหาในการรวมค่าของ Thread 1 และ Thread 2

การหยุดรอ Thread ให้ทำงานเสร็จ

- ในการทำงานกับ Thread ใน Java ในกรณีที่เราต้องการรอจนกว่า Thread จะทำงานเสร็จ จะใช้ เมธอด join()
- Thread ที่เรียกใช้งาน join() จะหยุดรอจนกระทั่ง Thread ที่รอนั้นเสร็จสิ้นการทำงาน หรือ ตาย
- จากนั้น Thread ที่เรียกใช้งาน join() จึงจะสามารถทำงานต่อไปได้

การใช้งาน join( )

```java
import java.io.*;

public class SumThread implements Runnable {
        int from;
        int where;
        int result = 0;

        public SumThread(int from, int where) {
                this.from  = from;
                this.where = where;
        }

        public void run() {
                for(int i = from; i <= where; i++) {
                        result += i;
                }
        }

        public int getResult() {
                return result;
        }
```

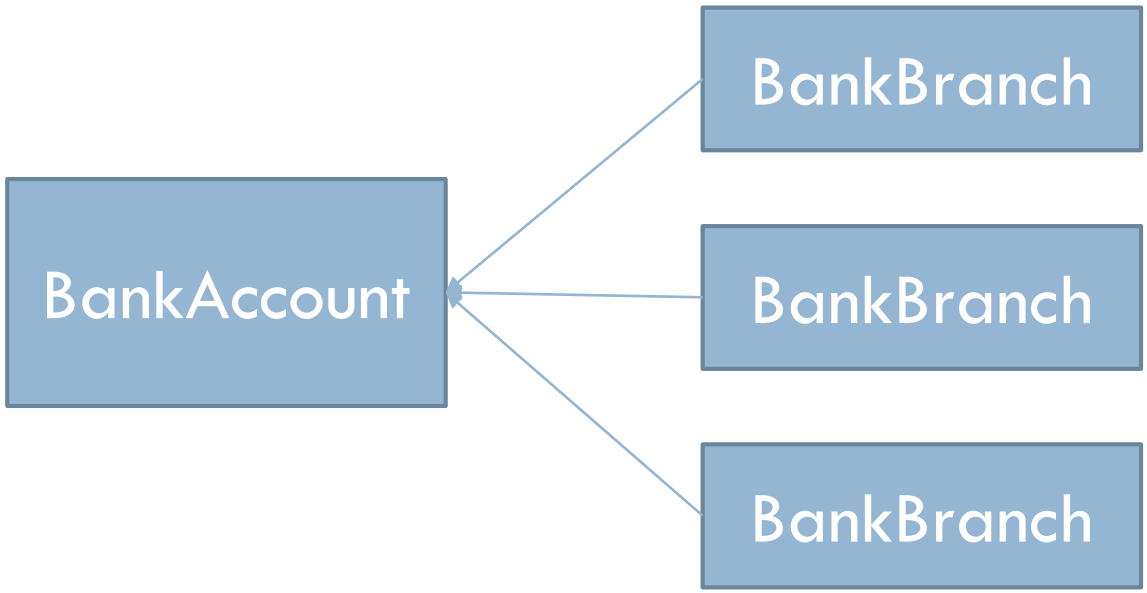
```java
        public static void main(String[] args) {
                int s = 0;
                SumThread s1 = new SumThread(0, 499999);
                SumThread s2 = new SumThread(500000, 1000000);
                Thread t1 = new Thread(s1);
                Thread t2 = new Thread(s2);

                try {
                        t1.start();  t2.start();
                        t1.join();  t2.join();
                        s = s1.getResult() + s2.getResult();
                } catch(Exception e){}
                System.out.println("Result = " + s);
        }

}
```

Thread Concurrency

- การทำงานแบบ **Multithread** ควรจะให้ความสำคัญกับคำว่า **“Thread safe”**
- ซึ่งหมายถึงเมื่อมี **Thread** ทำงานพร้อมกันหลายตัว และแต่ละตัวมีการเปลี่ยนแปลงค่าตัวแปรตัวเดียวกัน ถ้าไม่มีการจัดการให้ดี อาจจะทำให้ค่าตัวแปรนั้นผิดไป
- ในภาษา **Java** มีคำขยายชื่อ **synchronized** ขึ้นมาเพื่อทำงานกับ **critical region.**
- **Critical region** คือส่วนของโปรแกรมที่ควรอนุญาตให้ **Thread** สามารถเข้าช่วงนี้ได้ทีละ 1 **thread** เท่านั้น

ตัวอย่างปัญหาของ Race Condition

- ตัวอย่างการทำงานของธนาคารโดยจะมี Class อยู่ 3 Class
  - **BankAccount :** เก็บเงินในบัญชี
  - **BankBranch :** เหมือนสาขาหรือตู้ ATM ที่รับฝาก/ถอนเงินในบัญชี
  - **BadThread :** คลาส **main** ยกตัวอย่างการทำงาน



Class : BankAccount

```java
public class BankAccount {
    volatile int money = 0;


    public BankAccount(int money) {
        this.money = money;
    }

    public void deposit(int money) {
        for(int i = 0; i < money; i++) {
            this.money++;
        }
    }
    public  void withdraw(int money) {
        for(int i = 0; i < money; i++) {
            this.money--;
        }
    }

    public int getBalance() {
        return money;
    }
}
```

Class : BankBranch

```
public class BankBranch extends Thread {
    BankAccount bankAcct = null;
    String method = null;
    int money = 0;

    public BankBranch(BankAccount bankAcct, String method, int money) {
        this.bankAcct = bankAcct;
        this.method = method;
        this.money = money;
    }

    public void deposit(int money) {
        bankAcct.deposit(money);
    }

    public void withdraw(int money) {
        bankAcct.withdraw(money);
    }
    public void run() {
        if(method.equals("deposit")) deposit(money);
        else withdraw(money);
    }
}
```

Class: BadThread (main)

```
public class BadThread {
    public static void main(String[] args) {
        BankAccount bankAcct = new BankAccount(1000);

        BankBranch b1 = new BankBranch(bankAcct, "deposit", 10000);
        BankBranch b2 = new BankBranch(bankAcct, "withdraw", 10000);

        b1.start();
        b2.start();

        try {
            b1.join(); b2.join();
        } catch(Exception e) {}
        System.out.println("Balance = " + bankAcct.getBalance());

    }
}
```

ผลการทำงานไม่เหมือนกันในแต่ละครั้ง

```
Balance = 1000
Balance = 941
Balance = 937
Balance = 937
Balance = -4906
```

```
Balance = -3820
Balance = -2888
Balance = -386
Balance = -386
```

การแก้ไขปัญหาการแย่งกันเข้าถึงข้อมูลของ Thread

- **วิธีที่ 1** ใส่ synchronized ในเมธอดที่ต้องการทำ Critical Region
  - ทุก method ที่มี synchronized จะถูก block ทั้งหมดเพื่อให้ thread เพียงตัวเดียวเข้าใช้งานเพียง method ใด method หนึ่ง
  - รูปแบบตัวอย่าง

    public void **synchronized** deposit(int money)
  - ควรจะใส่ที่ตรงไหนบ้าง เพื่อป้องกันปัญหา race condition ?
- **วิธีที่ 2** เลือกทำ Critical Region เฉพาะบางส่วนของเมธอด วิธีนี้จะทำให้ยืดหยุ่นและสามารถเลือก lock แยกสำหรับกรณีที่เข้าถึงตัวแปรคนละตัวได้
  - จะต้องสร้าง Class ที่เป็น Object เพื่อใช้ในการ lock
  - ตัวอย่าง :

```
public void deposit(int money) {
    synchronized(o) {
        for(int i = 0; i < money; i++) {
            this.money++;
        }
    }
}
```

Keyword

- ทบทวนเรื่อง **Keyword** หน้าชื่อตัวแปร
  - **volatile**
    - **ตัวอย่าง: volatile int X;**
    - เป็นการประกาศตัวแปรที่จะบังคับให้เหมือนมีการเปลี่ยนแปลงข้อมูลใน **cache** ให้เขียนกลับลงไปในหน่วยความจำหลักทันที และเตือน **thread** อื่นที่มีการใช้งานตัวแปรนี้ให้อัพเดพข้อมูลจากหน่วยความจำหลัก
    - สามารถใช้ในการควบคุม **race condition** เบื้องต้นได้ แต่ในงานทั่วไปควรใช้คู่กับ **synchronized** แต่ถ้าใช้มากไปจะทำให้การทำงานโดยรวมของโปรแกรมช้าลง
  - **static**
    - ตัวอย่าง: **static int X;**
    - เป็นการประกาศตัวแปรที่มีที่เก็บข้อมูลเพียงที่เดียว และทุก **Object** จะอ้างอิงถึงที่เก็บข้อมูลที่เดียวกัน