

การควบคุมสิ่งผิดพลาด

โปรแกรมที่เราเขียนขึ้นมามักจะคอมไพล์หลายครั้ง กว่าที่จะสามารถนำโปรแกรมนั้นไปใช้งานได้จริงเนื่องจากต้องแก้ไขข้อบกพร่องของโปรแกรม ข้อบกพร่องบางอย่างสามารถตรวจสอบพบได้ในขณะคอมไพล์ เช่นการเขียนผิดหลักไวยากรณ์ของภาษา เมื่อคอมไพเลอร์ตรวจพบข้อบกพร่องมันจะแจ้งข้อบกพร่องที่ตรวจพบนั้นให้ทราบ ข้อบกพร่องที่ตรวจพบขณะคอมไพล์เรียกว่าข้อผิดพลาดขณะแปลโปรแกรม (compile-time error) ผู้เขียนโปรแกรมจะต้องแก้ไขและคอมไพล์ใหม่จนกว่าจะไม่มีพบข้อผิดพลาดชนิดนี้จึงจะนำโปรแกรมไปทดสอบหรือใช้งานได้

ถึงจะทำการแก้ไขโปรแกรมและคอมไพล์ใหม่จนไม่มีข้อผิดพลาดที่เกิดขึ้นขณะแปลโปรแกรมแล้วก็ตาม มิได้หมายความว่าโปรแกรมจะไม่มีข้อบกพร่อง ข้อบกพร่องบางอย่างตรวจไม่พบขณะแปลโปรแกรม แต่เมื่อนำโปรแกรมไปใช้งานจริงอาจทำงานผิดพลาดไม่เป็นไปตามเจตนาหรือทำงานได้เพียงบางส่วนหรือทำงานไม่ได้เลยก็ได้ ข้อผิดพลาดจากการเขียนโปรแกรมที่ทำให้โปรแกรมทำงานผิดไปจากที่ควรจะเป็นเรียกว่าข้อผิดพลาดทางตรรกะ (logic error) โปรแกรมที่ทำงานได้ แต่ทำงานอย่างผิด ๆ ถือว่าโปรแกรมนั้นมีจุดบกพร่อง (bug) ซึ่งเป็นสิ่งไม่พึงประสงค์ ข้อบกพร่องแบบนี้มักตรวจพบได้ยาก ในบางกรณีอาจสร้างความเสียหายอย่างใหญ่หลวงได้

ขณะที่โปรแกรมกำลังทำงาน อาจมีสิ่งผิดปกติเกิดขึ้นจนถึงขั้นที่โปรแกรมไม่สามารถทำงานต่อไปก็เป็นได้ ข้อบกพร่องที่เกิดขึ้นในขณะที่โปรแกรมกำลังทำงานจนเป็นเหตุให้ต้องยุติการทำงานลงกลางคันเช่นนี้เรียกว่าข้อผิดพลาดขณะทำงาน (runtime error)

พิจารณาโปรแกรมในตัวอย่างที่ 1 โปรแกรมนี้มีข้อบกพร่องที่สามารถตรวจพบได้ขณะแปลโปรแกรม

ตัวอย่างโปรแกรมที่ ๑ แสดงถึงสิ่งผิดปกติซึ่งสามารถตรวจสอบได้ในขณะคอมไพล์

```
1: class Main {
2:     public static void main(String []args) {
3:         int a=5;
4:         int c;
5:         c = a/0;
6:         System.out.println(c);
7:     }
8: }
```

ตัวอย่างโปรแกรมที่ ๑ นี้คอมไพล์ไม่ผ่าน เพราะมีข้อผิดพลาดขณะแปลโปรแกรม (compile-time error) ตัวแปรภาษาจะแสดงข้อความระบุสิ่งผิดพลาดตรงบรรทัดที่ ๕ ว่า Arithmetic exception เนื่องจากค่าตัวเลขใด ๆ ถ้าหารด้วยศูนย์จะไม่ได้ผลลัพธ์เป็นค่าตัวเลข จะได้เป็นค่าอนันต์ (infinity) (นักคณิตศาสตร์บางท่านไม่ถือว่าคุณค่าอนันต์เป็นค่าตัวเลข) เมื่อคอมไพเลอร์ตรวจพบว่าการหารด้วยศูนย์จึงไม่ยอมให้ทำเช่นนั้น แต่ถ้าหารด้วยตัวแปรแทนที่จะเป็นค่า ๐ ที่เขียนลงไปตายตัว คอมไพเลอร์จะยอมให้ผ่านไปได้ เนื่องจากว่า ขณะที่กำลังทำงานมาถึงบรรทัด

ที่ ๒ ไม่แน่ว่าค่าของตัวแปร b จะมีค่าเป็นเท่าใด ตัวอย่างที่ ๒ ปรับแก้มาจากตัวอย่างที่ ๑ ตรงบรรทัดที่ ๒ แทนที่จะให้ a หารด้วย 0 เหมือนตัวอย่างที่ ๑ เปลี่ยนเป็นหารด้วยตัวแปร b แทน ขณะคอมไพล์บรรทัดที่ ๒ ตัวแปลภาษาบอกไม่ได้ว่าตัวแปร b มีค่าเป็น 0 หรือไม่ จึงถือว่าไม่มีข้อผิดพลาดขณะแปล

หมายเหตุ

การหารด้วยค่าคงที่จำนวนเต็มศูนย์ จะเกิดข้อผิดพลาดเวลาแปลโปรแกรมหรือไม่ขึ้นอยู่กับรุ่นของคอมไพเลอร์ด้วย บางรุ่นตรวจสอบให้แต่บางรุ่นไม่ตรวจสอบ

ตัวอย่างโปรแกรมที่ ๒ แสดงสิ่งผิดปกติซึ่งเกิดขึ้นขณะที่โปรแกรมกำลังทำงาน

```
1: class Main {
2:     public static void main(String []args) {
3:         int a=5;
4:         int b=0;
5:         int c;
6:         c = a/b;
7:         System.out.println(c);
8:     }
9: }
```

ตัวอย่างโปรแกรมที่ ๒ สามารถคอมไพล์ผ่านและสามารถทำงานได้ไปจนถึงบรรทัดที่ ๒ จะเกิดข้อผิดพลาดขณะทำงาน โดยมีข้อความระบุความผิดพลาดว่า `java.lang.ArithmeticException: / by zero` หมายถึงมีสิ่งผิดปกติทางคณิตศาสตร์ (หารด้วยศูนย์) เมื่อมีสิ่งผิดปกติเช่นนี้เกิดขึ้น โปรแกรมจะยุติการทำงานทันที

หมายเหตุ

สำหรับนิพจน์ที่ให้ผลลัพธ์เป็นเลขจำนวนเต็มที่มีตัวหารเป็นตัวแปร จะเกิดสิ่งผิดปกติขึ้น ถ้าขณะที่กำลังหาค่าของนิพจน์นี้ตัวหารมีค่าเป็นศูนย์ แต่ถ้าเป็นนิพจน์ที่ให้ผลลัพธ์เป็นเลขที่มีจุดทศนิยม การหารด้วยศูนย์ไม่ถือว่าผิดปกติ จะได้ผลลัพธ์เป็นค่าอนันต์ (infinity)

เราจัดการกับสิ่งผิดปกติด้วยการแสดงข้อความบ่งบอกถึงสิ่งผิดปกติที่เกิดขึ้นแล้วหยุดการทำงานของโปรแกรม ทั้งนี้เพื่อไม่ให้ข้อผิดพลาดต่อเนื่องต่อไป แต่โปรแกรมที่ใช้งานจริงหากโปรแกรมยุติการทำงานทุกครั้งที่เกิดสิ่งผิดปกติขึ้นทั้ง ๆ ที่ยังพอจะทำงานต่อไปได้ จะสร้างความไม่สะดวกและรำคาญแก่ผู้ใช้โปรแกรม ดังนั้นในบางครั้งเราอาจต้องการให้โปรแกรมข้ามจุดที่เป็นปัญหา เพื่อจะได้ไปทำอย่างอื่นต่อไป ตัวอย่างที่ ๓ แสดงวิธีแก้ปัญหาที่เกิดขึ้นในโปรแกรมที่ ๒ ถ้าเราไม่ยอมให้โปรแกรมหยุดทำงานเมื่อมีการหารด้วยศูนย์ เราอาจตรวจสอบค่าของตัวแปรทุกครั้งก่อนนำไปใช้เป็นตัวหารว่ามีค่าเป็นศูนย์หรือไม่ ถ้ามีค่าเป็นศูนย์เราจะข้ามส่วนที่มีการหารไปเสีย

ตัวอย่างโปรแกรมที่ ๓ แสดงการตรวจสอบข้อบกพร่องขณะโปรแกรมทำงาน

```
๑. class Main {
๒.     public static void main(String []args) {
๓.         int a=5;
๔.         int b=0;
๕.         int c=0;
๖.         if (b == 0)
๗.             System.out.println("Error: Division by zero");
๘.         else
```

```

๙.         c = a/b;
๑๐.        System.out.println(c);
๑๑.    }
๑๒. }

```

บรรทัดที่ ๖ ในตัวอย่างที่ ๓ มีการตรวจสอบก่อนว่าตัวแปร b มีค่าเป็นศูนย์หรือไม่ ถ้ามีค่าเป็นศูนย์ โปรแกรมจะแสดงข้อความ "Error: Division by zero" เพื่อบอกให้ผู้ใช้โปรแกรมทราบว่า มีข้อบกพร่องเกิดขึ้นที่บรรทัดที่ ๙ ข้อความสั่งในบรรทัดที่ ๙ จะไม่ได้รับการทำงานถ้า b มีค่าเป็นศูนย์ ถ้า b ไม่เป็นศูนย์ บรรทัดที่ ๗ จะไม่ถูกทำงาน บรรทัดที่ ๙ จะได้รับการทำงานตามปกติ

การแก้ปัญหาตามตัวอย่างที่ ๓ นั้น ผู้เขียนโปรแกรมต้องเป็นผู้จัดการตรวจสอบข้อมูลเอง ซึ่งถ้าโปรแกรมมีขนาดใหญ่ มีจุดที่ต้องตรวจสอบเป็นจำนวนมาก การเขียนโปรแกรมจะยุ่งยาก ยื่นเย้อ ไม่น้อย

ภาษาจาวามีวิธีการจัดการกับสิ่งผิดปกติที่เกิดขึ้นในขณะทำงานด้วยวิธีการเชิงวัตถุ (object oriented) เมื่อมีสิ่งผิดปกติเกิดขึ้น อ็อบเจกต์ของคลาส Exception จะถูกสร้างขึ้น และเข้ามาทำงานแทนที่ ณ ตำแหน่งของโปรแกรมที่ก่อให้เกิดสิ่งผิดปกติ

ในภาษาจาวาเราสามารถจัดการกับสิ่งผิดปกติโดยอาศัยคำสำคัญ ๕ คำคือ try catch throw throws และ finally โดยหลักการแล้ว เราพยายาม (try) ที่จะทำงานข้อความสั่งต่าง ๆ ในบล็อกของข้อความสั่งที่ต้องการทำ และถ้ามีข้อผิดพลาดเกิดขึ้น ระบบจะโยน (throws) สิ่งผิดปกติ ซึ่งเราอาจดักจับ (catch) สิ่งผิดปกติแต่ละชนิดเพื่อจัดการเสียเอง หรือในที่สุด (finally) จะให้จัดการด้วยตัวจัดการโดยปริยาย (default handler) ด้วยก็ได้

รูปแบบของบล็อกที่ใช้จัดการสิ่งผิดปกติเป็นดังนี้

```

try {
    // ข้อความสั่งต่าง ๆ ที่ต้องการทำและอาจก่อให้เกิดสิ่งผิดปกติ
} catch (ExceptionType1 e) {
    // ข้อความสั่งที่ใช้จัดการเมื่อเกิดสิ่งผิดปกติ ExceptionType1
} catch (ExceptionType2 e) {
    // ข้อความสั่งที่ใช้จัดการเมื่อเกิดสิ่งผิดปกติ ExceptionType2
    throw e;        // โยนสิ่งผิดปกติไปอีกทอดถ้าต้องการ
} finally {
    // ข้อความสั่งที่ต้องให้ทำไม่ว่าจะเกิดสิ่งผิดปกติชนิดใดขึ้นหรือไม่ก็ตาม
}

```

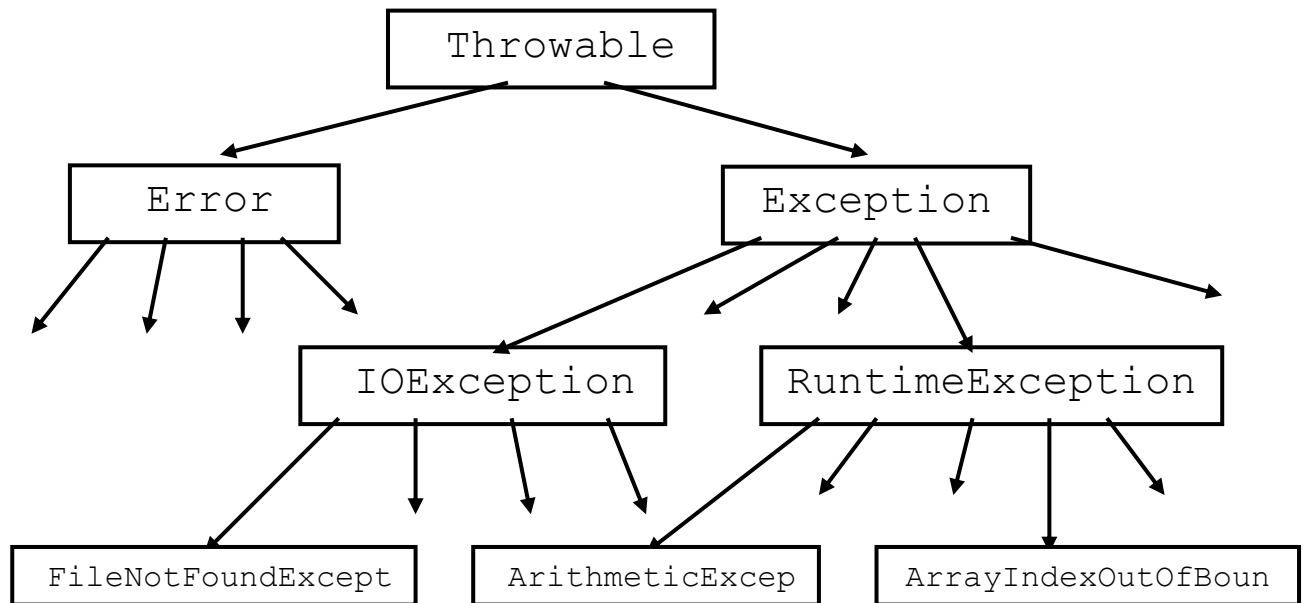
บล็อก try เป็นบล็อกที่บรรจุกลุ่มข้อความสั่งที่เราต้องการให้ทำงาน ซึ่งข้อความสั่งเหล่านี้มีโอกาสที่จะทำให้เกิดสิ่งผิดปกติชนิดต่าง ๆ หลังบล็อก try โดยทั่วไปแล้วจะเป็น บล็อก catch ซึ่งอาจมีเพียงบล็อกเดียวหรือหลาย

บล็อกก็ได้ ขึ้นอยู่กับว่าเราจะดักจับสิ่งผิดปกติที่ชนิด บล็อก catch บรรจุกลุ่มข้อความสั่งที่ใช้สำหรับดำเนินการเมื่อมีสิ่งผิดปกติเกิดขึ้น บล็อกชนิดสุดท้ายคือ บล็อก finally ซึ่งใช้บรรจุกลุ่มข้อความสั่งที่ต้องการให้ทำงานเสมอ ไม่ว่าจะเกิดสิ่งผิดปกติชนิดใดขึ้นหรือไม่ก็ตาม บล็อก finally เป็นบล็อกสุดท้ายที่ต้องทำงานเสมอไม่ว่าจะเกิดสิ่งผิดปกติขึ้นในบล็อก try หรือไม่ ถ้าเกิดสิ่งผิดปกติขึ้นและมีการดักจับสิ่งผิดปกตินั้นในบล็อก catch จะเข้าไปทำข้อความสั่งในบล็อก catch นั้นก่อนแล้วจึงไปทำบล็อก finally

หลังบล็อก try จะต้องมีบล็อกชนิดอื่นตามมาอย่างน้อยหนึ่งบล็อกเสมอ จะเป็นบล็อก catch หรือบล็อก finally อย่างใดอย่างหนึ่ง หรือทั้งสองก็ได้แล้วแต่ความจำเป็น สำหรับบล็อก catch สามารถมีได้มากกว่าหนึ่งบล็อก

ชนิดของสิ่งผิดปกติ

คลาส Throwable เป็นคลาสสูงสุดในลำดับชั้นของคลาส (class hierarchy) ที่ใช้จัดการสิ่งผิดปกติ ชั้นคลาสในลำดับถัดลงคือคลาส Exception และ คลาส Error คลาส Exception ใช้สำหรับกับสถานการณ์ผิดปกติที่ผู้เขียนโปรแกรมควรดักจับสิ่งผิดปกติที่เกิดขึ้น ส่วนคลาส Error ใช้กับสถานการณ์ผิดปกติค่อนข้างร้ายแรงที่ไม่ควรที่จะมีการดักจับสิ่งผิดปกตินั้น โปรแกรมควรยุติการทำงานมากกว่าที่จะพยายามทำงานต่อไป หากจะดักจับผิดปกติที่เกิดขึ้น ชั้นคลาสของ Error ควรทำด้วยความระมัดระวังเพราะอาจเกิดความผิดพลาดอย่างร้ายแรงได้ สำหรับรายละเอียดการเกิดสิ่งผิดปกติ(exception)แต่ละชนิดนั้น แสดงด้วยชั้นคลาสของ คลาส Exception ซึ่งสร้างโดยคลาส RuntimeException อ็อบเจกต์ของคลาส Exception เหล่านี้ โดยทั่วไปแล้วสร้างโดยอัตโนมัติเพื่อสนองตอบต่อข้อผิดพลาดของโปรแกรม



รูปที่ ๑ แผนผังลำดับชั้นของคลาสสิ่งผิดปกติชนิดต่าง ๆ

ตัวอย่างที่ ๔ แสดงการดักจับสิ่งผิดปกติ

```

๑. class Main {
๒.     public static void main(String []args) {
๓.         int a=5;
๔.         int b=0;
๕.         int c=0;
๖.         try {
๗.             c = a/b;
๘.         } catch ( ArithmeticException e) {
๙.             System.out.println("division by zero");
๑๐.        }
๑๑.        System.out.println(c);
๑๒.    }
๑๓. }
  
```

ตัวอย่างที่ ๔ แสดงบล็อก try และ catch ถ้ามีสิ่งผิดปกติเกิดขึ้นในบล็อก try แทนที่โปรแกรมจะยุติการทำงานไปเลยจะหยุดการทำงานเฉพาะในบล็อก try แล้วกระโดดข้ามไปทำงานในบล็อก catch ที่มีการดักจับสิ่งผิดปกติที่มีประเภทของสิ่งผิดปกติตรงกัน เมื่อทำงานในบล็อก catch จบลงแล้วจะย้อนกลับไปทำงานข้อความสั่งถัดไปจากข้อความสั่งที่ก่อให้เกิดสิ่งผิดปกติ

บล็อก catch ทำงานเหมือนเมทอด (method) โดยมีการรับพารามิเตอร์เป็นออบเจกต์ของคลาสที่ทำให้เกิดสิ่งผิดปกติซึ่งในที่นี้คือคลาส ArithmeticException แต่ในตัวอย่างนี้ไม่ได้เอาออบเจกต์ที่เป็นพารามิเตอร์ไปใช้ภายในบล็อกมีเพียงข้อความสั่งสำหรับพิมพ์ข้อความเพื่อบอกความผิดพลาดว่า “division by zero” ให้ผู้ใช้ทราบเท่านั้น ถ้าไม่เกิดสิ่งผิดปกติในบล็อก try จะไม่มีการทำงานในบล็อก catch

ตัวอย่างนี้จะเกิดสิ่งผิดปกติ ชนิด `ArithmeticException` ขึ้นในบรรทัดที่ ๗ เนื่องจาก `b` มีค่าเป็นศูนย์ ไม่สามารถนำไปใช้ในการหารได้ โปรแกรมจึงทำงานในบรรทัดที่ ๗ ไม่สำเร็จ เนื่องจากบรรทัดที่ ๗ อยู่ในบล็อก `try` เมื่อเกิดสิ่งผิดปกติขึ้นโปรแกรมจึงพยายามหาว่ามีบล็อก `catch` ที่คอยดักจับสิ่งผิดปกติชนิดเดียวกันนี้หรือไม่ ถ้ามี บล็อก `catch` ที่สัมพันธ์กัน โปรแกรมจะไหลเข้าไปทำงานในบล็อก `catch` นั้น ตัวอย่างนี้มีบล็อก `catch` ที่คอยดักจับสิ่งผิดปกติชนิด `ArithmeticException` อยู่ที่บรรทัดที่ ๘ ถึง ๑๐ ดังนั้นเมื่อเกิด `ArithmeticException` ขึ้นจึงเข้าไปทำงานในบล็อกนี้ โดยทำงานตามข้อความสั่งในบรรทัดที่ ๙ พิมพ์ข้อความว่า “division by zero” หลังจากนั้นจึงออกจากบล็อก `catch` ไป แล้วไปทำงานต่อข้อความสั่งนอกบล็อกคือ บรรทัดที่ ๑๑ พิมพ์ค่าของ `c` ซึ่งยังคงเป็นศูนย์ออกมา

ตัวอย่างที่ ๕

```

๑. class Main {
๒.     static void doSomething() {
๓.         try {
๔.             int array[] = {1,2,3,4,5};
๕.             int a=5, b=0;
๖.             b = array[a];
๗.         } catch (ArrayIndexOutOfBoundsException e) {
๘.             System.out.println("Error: index out of range");
๙.         }
๑๐.    }
๑๑.    public static void main(String []args) {
๑๒.        try {
๑๓.            int a=5, b=0, c=0;
๑๔.            doSomething();
๑๕.            c = a/b;
๑๖.        } catch ( ArithmeticException e) {
๑๗.            System.out.println("Error: division by zero");
๑๘.        }
๑๙.    }
๒๐. }
```

ตัวอย่างที่ ๕ แสดงโปรแกรมที่มีการดักจับสิ่งผิดปกติของแถวลำดับ ถ้าค่าของดัชนี (index) ที่ใช้ในการเข้าถึง ส่วนย่อย (element) ของแถวลำดับมีค่าต่ำกว่าศูนย์หรือมีค่ามากกว่าจำนวนส่วนย่อยลบด้วยหนึ่งจะเกิดสิ่งผิดปกติ ชนิด `ArrayIndexOutOfBoundsException` เนื่องจากค่าของดัชนีนั้นไม่สามารถอ้างอิงถึงส่วนย่อยใดของ แถวลำดับได้ เพราะเป็นค่าที่อยู่นอกช่วงที่เป็นไปได้ของแถวลำดับนั้น ตัวอย่างนี้เกิดสิ่งผิดปกติขึ้นที่บรรทัดที่ ๑๕ มีการหารด้วยศูนย์ จึงกระโดดข้ามไปพิมพ์ข้อความ “Error: division by zero” และเกิดสิ่งผิดปกติขึ้นอีกครั้งเมื่อทำงานมาถึงบรรทัดที่ ๖ เมื่อทำงานมาถึงบรรทัดนี้ ตัวแปร `a` มีค่าเป็น 5 ในบรรทัดนี้มีการเข้าถึงส่วนย่อยของ แถวลำดับโดยใช้ค่าของตัวแปร `a` เป็นดัชนีในการเข้าถึง ตัวแปร `a` มีค่าเป็น 5 จึงหมายถึงการเข้าถึงส่วนย่อยตัวที่ ๖

ของแถวลำดับซึ่งไม่มีเนื่องจากแถวลำดับนี้มีส่วนย่อยเพียง ๕ ตัวเท่านั้น จึงเกิดสิ่งผิดปกติชนิด

`ArrayIndexOutOfBoundsException` เมื่อเกิดสิ่งผิดปกตินี้ขึ้นจึงเข้าไปทำงานในบล็อก `catch` ที่ดักจับสิ่งผิดปกตินี้ โปรแกรมจะกระโดดไปทำงานในบรรทัดที่ ๘ พิมพ์ข้อความว่า "Error: index out of range" ออกมา

การดักจับสิ่งผิดปกติหลายชนิด

ในบล็อก `try` หนึ่ง ๆ อาจเกิดสิ่งผิดปกติได้หลายชนิด เราสามารถดักจับสิ่งผิดปกติต่าง ๆ ที่เกิดขึ้นในบล็อก `try` เดียวกัน ได้หลายกรณี ดังตัวอย่างที่ ๖

ตัวอย่างที่ ๖ แสดงการดักจับสิ่งผิดปกติหลายชนิดในบล็อก `try` เดียวกัน

```
๑. class Main {
๒.     public static void main(String []args) {
๓.         int a=5, b=0, c=0;
๔.         int array[] = {1,2,3,4,5};
๕.         try {
๖.             b = array[a];
๗.             c = a/b;
๘.         } catch ( ArithmeticException e) {
๙.             System.out.println("Error: division by zero");
๑๐.        } catch ( ArrayIndexOutOfBoundsException e) {
๑๑.            System.out.println("Error: index out of range");
๑๒.        }
๑๓.        System.out.println(c);
๑๔.    }
๑๕. }
```

ตัวอย่างที่ ๖ แสดงการดักจับสิ่งผิดปกติ ๒ ชนิดที่อาจเกิดขึ้นในบล็อก `try` เดียวกัน ในบล็อก `try` มีการใช้แถวลำดับ ถ้าใช้ดัชนีที่ไม่เหมาะสมอาจเกิด `ArrayIndexOutOfBoundsException` ขึ้นได้ และในบล็อกนี้มีการหารด้วยตัวแปรรวมอยู่ด้วย ซึ่งค่าของตัวแปรที่ใช้ในการหารอาจมีค่าเป็นศูนย์ได้ จึงอาจเกิด `ArithmeticException` ขึ้นได้ ดังนั้นเพื่อให้โปรแกรมสามารถทำงานต่อไปได้เมื่อมีสิ่งผิดปกติดังกล่าวเกิดขึ้น เราจึงควรจัดการกับสิ่งผิดปกติเหล่านี้เสียเอง โดยการสร้างบล็อก `try` ขึ้นมารับมือกับสิ่งผิดปกติทั้ง ๒ กรณี ตัวอย่างนี้เกิด `ArrayIndexOutOfBoundsException` ขึ้นในบรรทัดที่ ๖ โปรแกรมจึงกระโดดเข้าไปทำงานในบล็อก `catch` ในบรรทัดที่ ๑๑ แล้วจึงออกจากบล็อก `try` ไปทำงานต่อในบรรทัดที่ ๑๓

การโยนสิ่งผิดปกติ (Throwing Exceptions)

ในสถานการณ์ที่เมทอดไม่สามารถรับมือสิ่งผิดปกติได้ หรือไม่อยากจัดการสิ่งผิดปกติทั้งหมดเองนั้น เมทอดสามารถโยนความรับผิดชอบไปให้อ็อบเจกต์ของชั้นคลาสของคลาส **Throwable** จัดการแทนได้ โดยใช้ข้อความสั่ง `throw` รูปแบบโดยทั่วไปของข้อความสั่ง `throw` เป็นดังนี้

```
throw ThrowableInstance;
```

หลังจากทำข้อความสั่ง `throw` แล้ว โปรแกรมจะยุติการทำงานทันที จะทำงานไปไม่ถึงข้อความสั่งที่อยู่ถัดจากข้อความสั่ง `throw`

ถ้าเมทอดใด สามารถเป็นต้นเหตุที่ทำให้เกิดสิ่งผิดปกติในการทำงาน แต่ไม่จัดการความเป็นปรกตินั้นด้วยตัวมันเองเมทอดนั้นควรประกาศเพื่อให้ผู้ที่เรียกใช้เมทอดนี้รู้ เพื่อให้ผู้เรียกได้ป้องกันการเกิดสิ่งผิดปกตินั้น เราใช้คำสำคัญ `throws` ในการบ่งบอกรายการสิ่งผิดปกติต่าง ๆ ที่อาจเกิดขึ้นและจะถูกโยนโดยเมทอดนั้น ๆ

รูปแบบในการประกาศเมทอดที่อาจโยนสิ่งผิดปกติเป็นดังนี้

```
type method-name (parameter-list) throws exception-list
{
    . . . // ข้อความสั่งที่ต้องการดำเนินการเมื่อเกิดสิ่งผิดปกติ
}
```

เมทอดใดที่มีการใช้ข้อความสั่ง `throw` ผู้เขียนโปรแกรมควรประกาศเมทอดนั้นโดยระบุคำสำคัญ `throws` ไว้ด้วยเสมอ ถ้ามีการใช้ข้อความสั่ง `throw` กับสิ่งผิดปกติชนิดต่าง ๆ หลายชนิด ควรใส่ชื่อสิ่งผิดปกติไว้หลังคำสำคัญ `throws` ให้ครบถ้วน เพื่อเป็นการบอกให้ผู้ที่จะนำเมทอดนี้ไปใช้ได้ระมัดระวัง

ตัวอย่างที่ 7 เป็นตัวอย่างโปรแกรมที่มีการโยนสิ่งผิดปกติ โดยที่ไม่ได้เขียนข้อความสั่งสำหรับดักจับ (catch) สิ่งผิดปกติมาจัดการเสียเอง โปรแกรมนี้ตั้งใจโยนสิ่งผิดปกติไปให้อ็อบเจกต์ของคลาส `ArithmeticException` จัดการให้ ตัวอย่างนี้เป็นตัวอย่างโปรแกรมสั้นที่ไม่ได้เกิดสิ่งผิดปกติขึ้นจริง แต่ประสงค์ที่จะแสดงให้เห็นวิธีการโยนสิ่งผิดปกติ เมื่อโปรแกรมทำงานมาถึงบรรทัดที่ 4 จะโยนสิ่งผิดปกติไปให้อ็อบเจกต์ของคลาส `ArithmeticException` ซึ่งสร้างขึ้นใหม่ด้วยตัวดำเนินการ `new` พร้อมทั้งส่งข้อความที่เป็นสายอักขระว่า "test" ไปให้ constructor ของคลาส `ArithmeticException`

`ArithmeticException` เป็นชื่อคลาสที่สร้างไว้แล้วใน API คลาสนี้เป็น subclass ของคลาส `Exception` ซึ่งเป็น subclass ของคลาส `Throwable` อีกทอดหนึ่ง

ตัวอย่างที่ ๗ แสดงการโยนสิ่งผิดปกติไปให้อ็อบเจกต์ สำหรับจัดการสิ่งผิดปกติทำการแทน

```
๑. class Main {
๒.     static void doSomething() throws ArithmeticException {
๓.         System.out.println("inside procedure");
```



```

๔.         throw new ArithmeticException("test");
๕.     }
๖.     public static void main(String []args) {
๗.         doSomething();
๘.     }
๙. }

```

```

inside procedure
java.lang.ArithmeticException: test
    at java.lang.Throwable.<init>(Compiled Code)
    at java.lang.RuntimeException.<init>(Compiled Code)
    at java.lang.ArithmeticException.<init>(ArithmeticException.java:54)
    at Main.doSomething(Exception6.java:4)
    at Main.main(Exception6.java:7)

```

ตัวอย่างที่ ๗ นี้พิมพ์ข้อความระบุสิ่งผิดปกติดังนี้

ตั้งแต่บรรทัดที่ ๒ เป็นต้นมา เป็นผลลัพธ์ที่ได้จากการโยนสิ่งผิดปกติไปให้ อ็อบเจกต์ ของคลาส `ArithmeticException` ดำเนินการ ข้อความ "test" ทำยบรรทัดที่ ๒ นั้นเป็นข้อความที่ได้มาจากการส่งเป็นพารามิเตอร์ไปให้ตัวสร้างในบรรทัดที่ ๔ ของตัวอย่างที่ ๖ คลาสต่าง ๆ ที่ เอพีไอ มีไว้เพื่อจัดการสิ่งผิดปกตินั้นสามารถรับ `String` เป็นพารามิเตอร์ได้ทั้งสิ้น ทั้งนี้เพื่อใช้เป็นข้อความแสดงสิ่งผิดปกติตามที่ผู้เขียนโปรแกรมอยากให้แสดงนั่นเอง

ข้อความในบรรทัดต่าง ๆ หลังบรรทัดที่ ๒ นั้นเรียกว่า stack trace จาวาจะพยายามไต่ย้อนหลังให้ดูว่ามีการเข้าไปทำงานผ่านเมทอดใดมาบ้าง

บล็อก finally

เมื่อมีการโยนสิ่งผิดปกติ การไหลของข้อความสั่งในเมทอดจะไม่เป็นไปตามเส้นทางเชิงเส้นไปจนตลอดเมทอด จะมีการข้ามบางบรรทัด หรืออาจย้อนกลับ (return) ไปก่อนที่จะทำงานจบเมทอดก็ได้ถ้าไม่มีบล็อก catch ที่เข้าคู่กับสิ่งผิดปกติที่เกิดขึ้นนั้นคอยจัดการอยู่ ในบางครั้งเราต้องการความแน่ใจว่า ข้อความสั่งส่วนหนึ่งจะต้องทำงานเสมอไม่ว่าสิ่งผิดปกติใดจะเกิดและถูกดักจับหรือไม่ก็ตาม เราสามารถใช้คำสำคัญ finally ในการบ่งบอกบล็อกของข้อความสั่งที่ต้องการให้ทำงานเสมอไม่ว่าจะเกิดสิ่งผิดปกติขึ้นหรือไม่ ในกรณีที่เกิดสิ่งผิดปกติขึ้น สิ่งผิดปกตินั้นอาจถูกดักจับหรือไม่ก็ได้ ขึ้นอยู่กับว่ามีบล็อก catch ดักจับสิ่งผิดปกตินั้นหรือไม่ ถ้ามีการดักจับสิ่งผิดปกตินั้น จะเข้าไปทำข้อความสั่งในบล็อก catch ที่ดักจับมันก่อน แล้วจึงไปทำบล็อก finally แต่ถ้าเกิดสิ่งผิดปกติที่ไม่มีบล็อก catch ใดจับไว้ได้ จะเข้าไปทำข้อความสั่งในบล็อก finally ต่อจากนั้นสิ่งผิดปกตินั้นจะถูกโยนไปให้ java run-time จัดการต่อซึ่งจะแสดงข้อความผิดพลาดแล้วโปรแกรมจะยุติการทำงาน

ตัวอย่างที่ ๘ แสดงการใช้ บล็อก finally

```

๑. class Main {

```

```

๒.     public static void main(String []args) {
๓.         int a=5, b=0, c=0;
๔.         int array[] = {1,2,3,4,5};
๕.         try {
๖.             b = array[a];
๗.             c = a/b;
๘.         } catch ( ArithmeticException e) {
๙.             System.out.println("Error: division by zero");
๑๐.        } catch ( ArrayIndexOutOfBoundsException e) {
๑๑.            System.out.println("Error: index out of range");
๑๒.        } finally {
๑๓.            System.out.println("Message from finally block");
๑๔.        }
๑๕.        System.out.println("Message from the line after the whole try
        block");
๑๖.    }
๑๗. }

```

ตัวอย่างที่ ๘ แสดงการใช้ บล็อก finally ซึ่งเขียนไว้ที่บรรทัด ๑๒ - ๑๔ ไม่ว่าจะเกิดสิ่งผิดปกติขึ้นหรือไม่ก็ตาม บรรทัดที่ ๑๑ จะทำงานเสมอก่อนที่จะออกจากบล็อก try เมื่อโปรแกรมนี้ทำงานมาถึงบรรทัดที่ ๖ จะเกิดสิ่งผิดปกติเกี่ยวกับดัชนี (index) ของแถวลำดับเป็น 5 อันหมายถึงการเข้าถึงส่วนย่อยตัวที่ ๖ ซึ่งไม่มีอยู่ในแถวลำดับ เมื่อเกิดสิ่งผิดขึ้น โปรแกรมจะค้นหามีบล็อก catch ที่จะจัดการสิ่งผิดปกตินั้นหรือไม่ ถ้ามีจะเข้าไปทำในบล็อก catch นั้น ซึ่งในที่นี้มีบล็อก catch สำหรับจัดการสิ่งผิดปกตินี้อยู่แล้วที่บรรทัดที่ ๑๐ จึงเข้าไปทำงานที่บรรทัดที่ ๑๑ พิมพ์ข้อความว่า "Error: index out of range" ออกมา หลังจากทำข้อความสั่งในบล็อก catch เสร็จแล้ว จะไม่ไปทำข้อความสั่งต่อจากจุดที่ก่อให้เกิดสิ่งผิดปกติ มันจะยุติการทำงานในบล็อก try แต่เนื่องจากโปรแกรมนี้มี บล็อก finally ดังนั้นมันจึงเข้าไปทำงานในบล็อก finally ซึ่งพิมพ์ข้อความว่า "Message from finally block" ออกมา เป็นอันสิ้นสุดการจัดการสิ่งผิดปกติ เนื่องจากกรณีนี้สามารถดักจับสิ่งผิดปกติโดยบล็อก catch ได้ ดังนั้น โปรแกรมจึงทำงานต่อไปได้ถึงแม้จะมีสิ่งผิดปกติเกิดขึ้นก็ตาม เพียงแต่ไม่ทำในส่วนของบล็อก try ที่ก่อให้เกิดสิ่งผิดปกติเท่านั้น ข้อความสั่งต่าง ๆ ที่อยู่นอกบล็อก try ยังคงทำงานต่อไป ดังนั้นโปรแกรมจะข้ามบรรทัดที่ 7 ไปทำงานต่อบรรทัดที่ 15 ผลลัพธ์จากการทำงานของโปรแกรมนี้เป็นดังนี้

```

Error : index out of range
Message from finally block
Message from the line after the whole try block

```

ตัวอย่างที่ ๙ แสดงการดักจับสิ่งผิดปกติและมีบล็อก finally แต่ดักจับสิ่งผิดปกติไม่ครบถ้วน เมื่อโปรแกรมทำงานไปถึงบรรทัดที่ ๖ เกิดสิ่งผิดปกติชนิด ArrayIndexOutOfBoundsException แต่ไม่มีการดักจับสิ่งผิดปกติชนิดนี้ ระบบจึงจัดการสิ่งผิดปกติชนิดนี้เสียเองแล้วยุติการทำงาน แต่เนื่องจากโปรแกรมนี้มี บล็อก

finally ควบคู่กับบล็อก try ด้วย ดังนั้นโปรแกรมจึงไปทำบล็อก finally ก่อนที่จะไปจัดการสิ่งผิดปกติแล้ว จึงจบการทำงาน

ตัวอย่างที่ ๕

```

๑. class Main {
๒.     public static void main(String []args) {
๓.         int a=5, b=0, c=0;
๔.         int array[] = {1,2,3,4,5};
๕.         try {
๖.             b = array[a];
๗.             c = a/b;
๘.         } catch ( ArithmeticException e) {
๙.             System.out.println("Error: division by zero");
๑๐.        } finally {
๑๑.            System.out.println("Message from finally block");
๑๒.        }
๑๓.        System.out.println("Message from the line after the whole try
        block");
๑๔.    }
๑๕. }

```

ผลลัพธ์จากการทำงานตัวอย่างโปรแกรมที่ ๕ แสดงข้างล่างนี้ จะเห็นว่าเมื่อข้อความจากบล็อก finally และมีข้อความที่เกิดจาก ข้อบกพร่อง ของระบบที่จัดการสิ่งผิดปกตินี้ ส่วนบรรทัดที่ ๑๓ ของโปรแกรม ไม่มีการทำงาน

```

Message from finally block
java.lang.ArrayIndexOutOfBoundsException: 5
    at Main.main(Exception7.java:6)

```

เนื่องจากโปรแกรมยุติการทำงานไปก่อน

จะเห็นว่าจะมีการเข้าไปทำงานในบล็อก finally เสมอ จึงเหมาะที่จะใช้ดำเนินการสะสางงานบางอย่าง เช่น การ ปิดแฟ้มข้อมูลที่เปิดไว้ การคืนทรัพยากรต่าง ๆ ที่มีการขอใช้ค้างไว้ ก่อนที่จะให้โปรแกรมยุติการทำงานไป