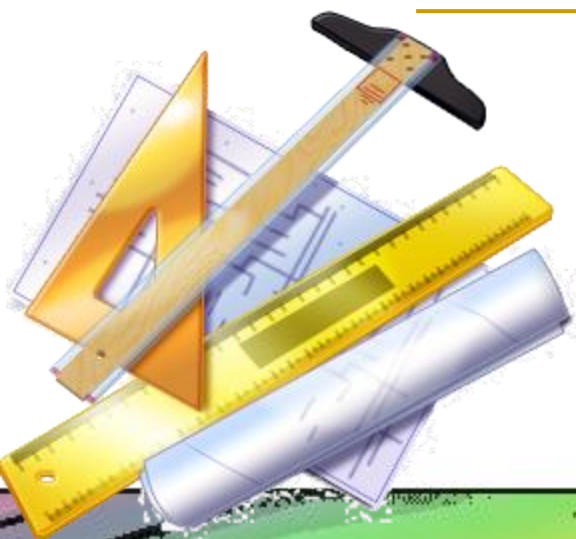


# Exception และการอ่านเท็กซ์ไฟล์พื้นฐาน

Lecture 10

Yaowadee Temtanapat

เยาวดี เต็มธนาภักตร์



# วัตถุประสงค์ของการเรียนในวันนี้

- เรียนรู้การใช้และการสร้างคลาสของความผิดพลาด (Exception) ได้
- เข้าใจข้อแตกต่างระหว่าง checked และ unchecked exception
- เรียนรู้ในการส่งต่อและการจับความผิดพลาด (Exception)
- เรียนรู้เกี่ยวกับ File object
- สามารถอ่าน Text file ได้
- เรียนรู้ว่าจะควรจับความผิดพลาดเมื่อใดและที่ไหน

# ความผิดพลาดกับการจัดการ

## ■ การจัดการกับความผิดพลาดแบบดั้งเดิม:

- ❑ method คืนค่าที่บอกว่ามีการทำงานผิดพลาด เช่น คืน false หรือ -1

## ■ ปัญหา

- ❑ ผู้เรียก ลืมที่จะตรวจสอบ ซึ่งอาจส่งผลกระทบตามมาภายหลัง?
- ❑ ผู้เรียก ตรวจสอบผลลัพธ์ที่คืนมา แต่ไม่รู้จะจัดการกับความผิดพลาดอย่างไร อาจจำเป็นต้องส่งต่อ โดยคืนค่า error code กลับออกมา เช่น

```
if ( !x.doSomething() ) return false;
```

# การจัดการกับความผิดพลาด

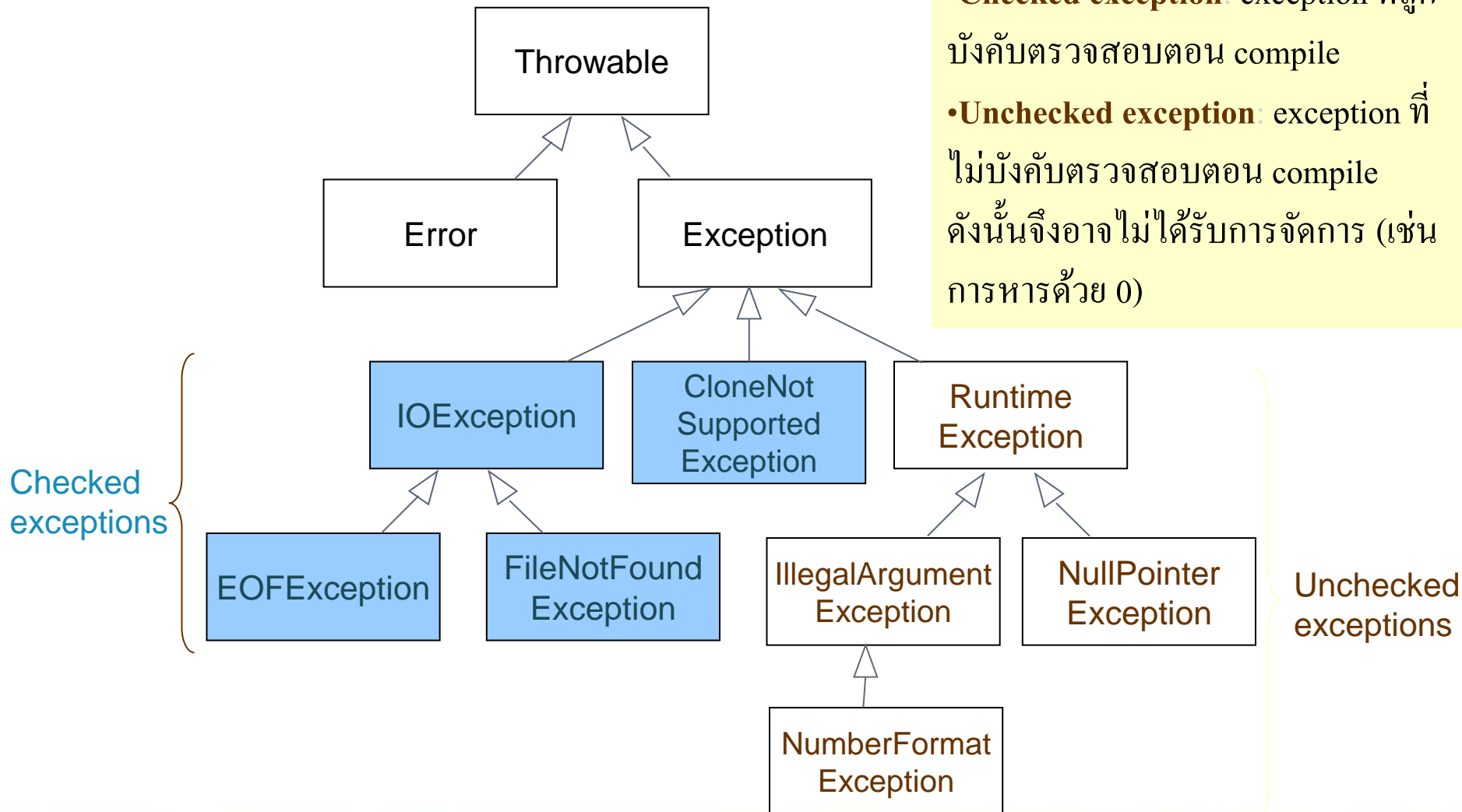
- Exception Handling: เป็นกลไกที่ถูกออกแบบไว้เพื่อให้
  - ความผิดพลาดไม่ถูกละเลย
  - ความผิดพลาดถูกจัดการได้อย่างเหมาะสม (ไม่จำเป็นต้องจัดการโดยผู้ทำขณะนั้น)

- เมื่อเกิดความผิดพลาด โปรแกรมสามารถแจ้งให้รู้ว่าเกิดความผิดพลาด

- ตัวอย่างเช่น โดยการสร้างวัตถุแสดงความผิดพลาดแล้วส่งให้ผู้เรียก

```
if (failure) {  
    XxxException e = new XxxException(...); // สร้างวัตถุ exception  
    throw e; // โยน exception ต่อเพื่อให้ผู้เรียกรู้และจัดการอย่างเหมาะสม  
}
```

# Exception Class Hierarchy



ชนิดของ Exception:

- **Checked exception:** exception ที่ถูกบังคับตรวจสอบตอน compile
- **Unchecked exception:** exception ที่ไม่บังคับตรวจสอบตอน compile ดังนั้นจึงอาจไม่ได้รับการจัดการ (เช่น การหารด้วย 0)

# ชนิดของความผิดพลาด (1)

## ■ ชนิดของความผิดพลาด: Checked VS Unchecked

### ■ Checked Exception

- ❑ ตรวจสอบโดย Compiler → ผู้พัฒนาต้องรับรู้เกี่ยวกับความผิดพลาดนั้น
- ❑ ใช้โดยทั่วไปสำหรับการผิดพลาดที่อาจมีได้ แม้ในโปรแกรมที่พัฒนาอย่างถูกต้อง
  - ตัวอย่างเช่น IOException และ subclasses ของมัน เป็น checked exceptions
- ❑ Checked exceptions เป็น subclasses ของ Exception ที่ไม่ใช่ subclasses ของ RuntimeException

# ชนิดของความผิดพลาด (2)

## ■ Unchecked Exception

### □ ไม่ตรวจสอบโดย Compiler

- ทั่วไปเป็นความผิดพลาดในการเขียนของผู้พัฒนา
  - ตัวอย่างเช่น NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException
- ความผิดพลาดบางประเภทที่ไม่ถูกตรวจสอบโดย compiler
  - ตัวอย่างเช่น Integer.parseInt จะส่งต่อ unchecked NumberFormatException
- Error จากการทำงานของ Virtual machine
  - ตัวอย่างเช่น OutOfMemoryError เป็น unchecked

# การจัดการกับความผิดพลาด (Revisited)

- **Exception Handling:** ความผิดพลาดไม่ควรถูกละเลยและควรต้องได้รับการจัดการที่เหมาะสม
  - Method ส่งความผิดพลาดกลับไปยังผู้เรียก → **throwing exception**
    - อาจสร้างวัตถุแสดงความผิดพลาดแล้วส่งให้ผู้เรียก
    - อาจส่งต่อความผิดพลาดที่ได้รับจากการเรียก method อื่น
      - เมื่อก่อนโยนความผิดพลาดมา ส่งต่อความผิดพลาดนั้นให้ผู้เรียก
  - Method สามารถจัดการความผิดพลาดได้เองเหมาะสม (ไม่ต้องส่งต่อจัดการเองได้ที่ method ที่ทำงานขณะนั้น) → **catching exception**
  - ผสมทั้ง 2 แบบเข้าด้วยกัน



# การสร้าง ส่งและประกาศส่งความผิดพลาดไปยังผู้เรียก

- ตัวอย่าง ควรตรวจสอบว่า player พยายามเดินออกนอกขอบเขตของ room หรือไม่

```
public void walk(int x, int y) {  
    if (room.inbound(player.getPositionX(),  
                      player.getPositionY()))  
        player.move(x, y);  
}
```

ประกาศเพื่อให้ผู้เรียกรู้ว่าอาจมีการ  
ส่งความผิดพลาด จำเป็นในกรณี  
checked (ควรมี ในกรณี unchecked)

- หากเกิน ควรจัดการต่ออย่างไร

```
public void walk(int x, int y)  
    throws IllegalArgumentException {  
    if (room.inbound(player.getPositionX()+x,  
                      player.getPositionY()+y))  
        player.move(x, y);
```

สร้างและส่งความผิดพลาด  
method หยุดและส่งคืนการ  
ทำงานไปยังผู้เรียกเพื่อให้อำนาจจัดการ

```
    else {  
        IllegalArgumentException ex = new  
            IllegalArgumentException("Walk out of bound");  
        throw ex;  
    }
```

# Syntax: การส่งความผิดพลาด (Throwing Exception)

## ■ Syntax ของการส่งความผิดพลาด

**throw** *exceptionObject*

## ■ ตัวอย่างเช่น:

```
throw new IllegalArgumentException();
```

## ■ จุดมุ่งหมาย:

- ❑ เพื่อส่งความผิดพลาดพร้อมกับการควบคุม/ทำงานกลับไปยังผู้เรียก เพื่อให้ดำเนินการต่ออย่างเหมาะสม

# Syntax: การประกาศการส่ง/ส่งต่อ Exception

## ■ Syntax ของการประกาศเมทอดที่มีการส่งความผิดพลาด

*accessSpecifier returnType methodName (Type variable, ...)*

**throws ExceptionClass1, ExceptionClass2 . . .**

## ■ ตัวอย่างเช่น:

```
public void walk(int x, int y)
    throws IllegalArgumentException{
    // method body ...
```

## ■ จุดมุ่งหมาย:

- ❑ เพื่อระบุเป็นข้อกำหนดให้รู้ว่า method นั้น ๆ มีการส่ง/โยนความผิดพลาด (โดยเฉพาะที่ต้องตรวจสอบในกรณี Checked Exception)

# การทำงานกับ File Object: ใช้ระบุ/ตรวจสอบ file/directory

- การสร้าง File Object (จาก `java.io package`) โดย

```
File file = new File("sample.data");
```


- โดยใช้ชื่อของไฟล์ที่ต้องการเป็น argument ของ constructor

```
File file = new File  
("Lecture/lecture10", "sample.data");
```

- หากอ้างถึง file ที่ไม่มีอยู่, ตรวจสอบด้วยเมทอด `exists` คืนค่า `false`
- Method บางส่วนเช่น `delete`, `renameTo`, `exists`

# การอ่าน Text File โดยใช้ Scanner

FileNotFoundException  
เป็น checked exception  
อยู่ใน java.io package



- Constructor ของ Scanner ที่อ่านข้อมูลจากไฟล์

```
public Scanner(File source) throws FileNotFoundException
```

- การอ่าน textfile:

- สร้าง Scanner object ที่เชื่อมต่อกับไฟล์

```
File file = new File("sample.txt");  
Scanner scanner = new Scanner(file);
```

- จากนั้นอ่านข้อมูลจากในไฟล์ ได้ตามเมทอดที่เคยใช้

- `nextLine()`
    - `nextInt()`
    - `nextDouble();`

# การประกาศส่งต่อความผิดพลาดไปยังผู้เรียกโดยไม่จัดการ

## ■ สมมติอ่านข้อมูลจากเท็กซ์ไฟล์

```
public void read(File inFile) {  
    Scanner scan = new Scanner(inFile);  
    double amount = scan.nextDouble();  
    :  
}
```

Constructor ของ  
Scanner อาจโยน  
FileNotFoundException

## ■ การประกาศเมทอดเพื่อส่งต่อความผิดพลาด (read ไม่รู้ว่าควรจัดการอย่างไร)

- ส่งต่อความผิดพลาดนั้นไปยังผู้เรียก read อีกชั้นหนึ่ง โดยประกาศ throws FileNotFoundException ไว้กับชื่อ method (ไม่ใช่การไม่รับผิดชอบ)

```
public void read(File inFile)  
    throws FileNotFoundException {
```

```
    Scanner scan = new Scanner(inFile);  
    :
```

throws FileNotFoundException  
ต่อไปยังผู้เรียก

# การจัดการกับความผิดพลาด โดย try-catch

```
try {  
    Scanner in = new Scanner(new File("sample.txt"));  
    double amount = in.nextDouble();  
    //...จะได้  
}  
catch (FileNotFoundException e) {  
    System.out.println("File not found error" + e);  
}  
catch (InputMismatchException e) {  
    System.out.println("Input was not a number");  
}
```

# การจับความผิดพลาด (Catching Exception)

- การจัดการกับความผิดพลาดโดยการจับ (Catching)
  - เริ่มต้นจะทำงานในส่วนของ *try block* ก่อน
  - ถ้าไม่มีข้อผิดพลาด ทำเมท็อดนั้นจนเสร็จ โดยไม่ทำส่วนของ catch block
  - ถ้าเกิดความผิดพลาด และมีการจับ
    - ถ้าชนิดของ exception ที่เกิดขึ้นสอดคล้องกับการจับ ย้ายไปทำงานในส่วนของ catch clause นั้น
    - ถ้าชนิดของ exception ที่เกิดขึ้นไม่สอดคล้องกับการจับ โยนความผิดพลาดกลับไปยังผู้เรียก
    - ถ้าโยนต่อจนถึง main โดยไม่มีการจับความผิดพลาดนั้น โปรแกรมหยุดการทำงาน พร้อมกับพิมพ์ stack trace ของความผิดพลาด



# สรุป Syntax สำหรับการ try-catch

## ■ การทดลองและจัดการความ

ผิดพลาด (Try-Catch): เพื่อทำงานและจัดการกับ statements ที่อาจเกิดความผิดพลาดได้

- ❑ กรณีไม่มีข้อผิดพลาด ทำเพียง statements ใน try block
- ❑ กรณีมีความผิดพลาด หยุดการทำงานใน try เพื่อทำงานตาม statement ที่อยู่ใน catch clause ที่ใกล้เคียงที่สุด

■ **หมายเหตุ** ลำดับการจับ (catch) ความผิดพลาดต้องเรียงไล่จากเฉพาะที่สุดไปหาทั่วไป

## ■ Syntax:

```
try {  
    statement;  
    :  
}  
catch (ExceptionClass1  
    exceptionObject) {  
    statement;  
    :  
}  
catch (ExceptionClass2  
    exceptionObject) {  
    statement;  
    :  
}
```

# สรุป Syntax สำหรับการ try-catch (version 7 เท่านั้น)

## การ Catch

- กรณีที่ตัวจับความผิดพลาด**ทำงานเหมือนกัน** รวบรวม catch clause โดยใช้ pipe | ได้

- ถ้า catch superclass **ไม่**อนุญาต catch subclass ใน pipe

## ■ Syntax:

```
try {  
    statement;  
:  
}  
catch (ExceptionClass1 |  
      ExceptionClass2  
      exceptionObject) {  
    statement;  
:  
}
```

Pipe

## finally Clause (1)

- **ปัญหา** เมื่อมีการโยนความผิดพลาด, เมื่้อดสิ้นการทำงานให้กับผู้เรียก อาจทำให้เมื่้อดสิ้นสุดโดยข้ามบางส่วนของการทำงานที่สำคัญไป

- ตัวอย่างเช่น: อาจไม่ได้ปิด file ให้เรียบร้อย

```
Scanner in = null;  
in = new Scanner(new File(filename));  
characterReader.read(in);  
in.close();    // ควรต้องปิด file แม้ในกรณีที่เกิดความผิดพลาดด้วย
```

## finally Clause (2)

- **ทางเลือก** ใช้ประโยค finally สำหรับส่วนของโค้ดที่ต้อง run เสมอ (ทุกกรณี ยกเว้นหยุดโปรแกรม)

```
Scanner in = null;  
try {  
    in = new Scanner(new File(filename));  
    characterReader.read(in);  
}  
catch(Exception ex) { ... }  
finally {  
    if (in !=null) in.close();  
}
```

# try พร้อมประกาศตัวแปร เฉพาะใน version 7

- **ทางแก้** ใน version 7 (กรณีทำงานกับ resource ที่จำเป็นต้องปิดให้เรียบร้อย) ประกาศตัวแปร resource เป็นส่วนหนึ่งของประโยค try
  - ❑ ถ้า resource นั้น implements อินเทอร์เฟซ AutoClosable
  - ❑ Resource นี้ จะถูกปิดโดยอัตโนมัติไม่ว่าจะในกรณีสำเร็จหรือล้มเหลว

```
try (Scanner in = new Scanner(new File(filename))) {  
    characterReader.read(in);  
}
```

**หมายเหตุ** ถ้ามีตัวแปร resource แล้ว อาจ**ไม่**ต้องมี catch หรือ finally ได้  
(ถ้ามีการจัดการกับ exception ของ resource ด้วยวิธีการที่เหมาะสม)

# สรุป Syntax สำหรับการ try-finally

## ■ การทำงานกับ Try-Finally:

เพื่อทำงาน statements ที่อาจเกิดความผิดพลาดและกรณีที่ต้อง run เสมอ

- ❑ กรณีปกติ ทำงานในส่วน try block
- ❑ **ต้อง**ทำ statements ใน finally เสมอไม่ว่าจะเกิดความผิดพลาดหรือไม่
  - ยกเว้น กรณี exit โปรแกรม

## ■ Syntax:

```
try {  
    statement;  
:  
}  
finally {  
:  
}
```

# สรุป Syntax สำหรับการ try-catch-finally

- **try-catch-finally:** เพื่อทำงานและจัดการกับ statements ที่อาจเกิดความผิดพลาดได้ และทำสิ่งจำเป็นเสมอ
  - กรณีไม่มีข้อผิดพลาด ทำเพียง statements ใน try block
  - กรณีมีความผิดพลาด หยุดการทำงานใน try เพื่อทำงานตาม statement ที่อยู่ใน catch clause ที่ใกล้เคียงที่สุด
  - เมื่อผ่านจากทั้ง 2 กรณีจะทำประโยค finally ก่อนกลับไปยังผู้เรียกเสมอ

## ■ Syntax:

```
try {  
    statement;  
    :  
}  
catch (ExceptionClass1  
    exceptionObject) {  
    statement;  
    :  
}  
finally {  
    statement;  
    :  
}
```

# การออกแบบวัตถุแสดงความผิดพลาดของเราเอง (1)

- สร้าง Class ที่สืบทอดจาก Exception หรือ Throwable
  - หากต้องการ Unchecked exception ให้สืบทอดจาก RuntimeException
    - ข้อดี เปิดทางให้ผู้พัฒนาอาจเลือกตรวจสอบ
    - ข้อด้อย อาจละเลยการตรวจสอบได้
- กำหนด Constructors 2 แบบ:
  - ไม่มี parameter ()
  - มี String เป็น parameter สำหรับแสดงเหตุผล (String reason)



## การออกแบบวัตถุแสดงความคิดพลาดของเราเอง (2)

### ■ ตัวอย่าง Checked Exception เช่น

```
public class InvalidCommandException
    extends Exception {
    public InvalidCommandException() {
        super("Using invalid command");
    }
    public InvalidCommandException(String reason) {
        super(reason);
    }
}
```

## การออกแบบวัตถุแสดงความผิดพลาดของเราเอง (3)

### ■ ตัวอย่าง Unchecked Exception เช่น

```
public class IllegalDirectionException
    extends RuntimeException {
    public IllegalDirectionException() {
        super("Move to an illegal direction");
    }
    public IllegalDirectionException(String reason)
    {
        super(reason);
    }
}
```

# ตัวอย่างการใช้งาน: CharacterFileReader.java และ MonsterCage.java

- โปรแกรมอ่านข้อมูลของสัตว์ประหลาดในเกมส์:
  - ขั้นตอนการทำงานปกติ
    - อ่าน monster จาก file
    - เพิ่ม monster ใน MonsterCage
    - พิมพ์ตำแหน่งของ monsters ที่อ่านได้
  - กรณีที่อาจผิดพลาด:
    - อะไรเป็นข้อผิดพลาดที่อาจเกิดขึ้นได้
      - อาจไม่มี File นั้นอยู่, ข้อมูลใน File อาจอยู่ในรูปแบบที่ไม่ถูกต้อง
    - Method ใดควรเป็นผู้รับผิดชอบจัดการกับความผิดพลาด
      - main method ของ MonsterCage ทำ interact กับผู้ใช้โดยตรง ควรรายงานให้ผู้ใช้รู้
      - methods อื่น ๆ ทำหน้าที่ส่งผ่านความผิดพลาดกลับมาให้ผู้เรียก

# read method ในคลาส CharacterFileReader

```
private static Monster read(Scanner scan){
    int x, y;
    boolean status = false;
    if (scan.hasNext())
        x = scan.nextInt();
    else
        return null;
    if (scan.hasNext())
        y = scan.nextInt();
    else
        throw new RuntimeException("Unexpected EOF: no y value");
    if (scan.hasNext()){
        String state = scan.nextLine();
        status = state.trim().equalsIgnoreCase("sleep");
    }
    Monster mons = new Monster();
    mons.setPositionX(x);
    mons.setPositionY(y);
    mons.setSleep(status);
    return mons;
}
```

# readMonsters method ใน CharacterFileReader class

```
public static Monster[][] readMonsters(String filename)
    throws FileNotFoundException {
    File file = new File(filename);
    Scanner scan = null;
    Monster[][] cage = null;
    try {
        scan = new Scanner(new File(filename));
        int squareSize = scan.nextInt();
        int size = (int) Math.round(Math.sqrt(squareSize));
        if(size * size != squareSize)
            throw new RuntimeException("File contains an invalid size of cage");
        cage = new Monster[size][size];
        for(int i = 0; i < cage.length; i++){
            for(int j = 0; j < cage[i].length; j++){
                Monster mon = read(scan);
                if(mon != null)
                    cage[i][j] = mon;
                else
                    throw new RuntimeException("Not enough monsters");
            }
        }
    } finally {
        if (scan != null) scan.close();
    }
    return cage;
}
```

สังเกต ใช้ **finally clause** เพื่อปิด file  
หลังการทำงานเสร็จ ไม่ว่าจะเกิดความ  
ผิดพลาดหรือไม่

# constructor และ print() method ในคลาส MonsterCage

```
private Monster[][] cage;

public MonsterCage(String filename) throws Exception{
    cage = CharacterFileReader.reasMonsters(filename);
}

public void print(){
    System.out.println("-----Cage-----");
    for(Monster[] row : cage){
        for(Monster monster : row){
            if(monster.isSleep())
                System.out.print(". ");
            else
                System.out.print("o ");
        }
        System.out.println();
    }
}
```

# Interaction กับผู้ใช้ใน main method ของคลาส MonsterCage (ต่อ)

```
public static void main(String[] args){  
    try{  
        MonsterCage cage = new MonsterCage("monsters.txt");  
        cage.print();  
    } catch(Exception ex){  
        System.out.println(ex);  
        ex.printStackTrace();  
    }  
    System.out.println("End of program");  
}
```

# ภาพลำดับการเรียกและการจัดการกับความผิดพลาด

## ■ ลำดับการเรียก methods

MonsterCage.main → CharacterFileReader.readMonsters → CharacterFileReader.read

## ■ CharacterFileReader.readMonsters throws RuntimeException ถ้าหากเกิดความผิดพลาด

1. CharacterFileReader.read ไม่จัดการแต่ส่งต่อ exception และสิ้นสุดการทำงานทันที
2. CharacterFileReader.readMonsters ไม่จัดการแต่ส่งต่อ exception และสิ้นสุดการทำงานทันที
3. MonsterCage.main มีการจัดการกับ Exception ซึ่งเป็น superclass ของ RuntimeException โดยที่ตัวจัดการจะพิมพ์ข้อผิดพลาดนั้นพร้อมพิมพ์ stack trace ของโปรแกรมให้ผู้ใช้ จากนั้นแสดงข้อความ “End of program” และสิ้นสุดการทำงาน



# สรุปการเรียนรู้ในวันนี้

- เรียนการสร้างและใช้คลาสของความผิดพลาด (Exception) ได้
- เข้าใจข้อแตกต่างระหว่าง checked และ unchecked exception
- เรียนรู้ในการส่งต่อและการจับความผิดพลาด (Exception)
- เรียนรู้เกี่ยวกับ File object
- สามารถอ่าน Text file ได้
- เรียนรู้ว่าจะควรจับความผิดพลาดเมื่อใดและที่ไหน