Ngoại lệ

Bộ môn công nghệ phần mềm
Khoa công nghệ thông tin
Trường ĐHCN, ĐHQG Hà Nội

Nội dung

- Định nghĩa ngoại lệ
- Nhược điểm của cách xử lý ngoại lệ truyền thống
- Các cấu trúc xử lý ngoại lệ trong Java
- Quy trình JRE xử lý ngoại lệ
- Ném ngoại lệ
- Các thao tác khác trên ngoại lệ
- Ưu điểm của cách xử lý ngoại lệ trong Java

Tài liệu tham khảo

- Giáo trình Lập trình HĐT, chương 11
- Java documentation https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

Định nghĩa ngoại lệ

Định nghĩa ngoại lệ

- Ngoại lệ là một sự kiện xảy ra khi thực thi chương trình mà phá vỡ luồng hoạt động mong đợi
- Ngoại lệ có ba loại
  - Ngoại lệ được kiểm tra (checked exception)
  - Lỗi (error)
  - Ngoại lệ runtime (runtime exception)

Các loại ngoại lệ

- Loại 1: Ngoại lệ được kiểm tra (checked exception)
  - Là ngoại lệ xảy ra trong chương trình, chương trình có thể dự đoán được ngoại lệ và xử lý được ngoại lệ đó
  - Tất cả mọi loại ngoại lệ đều là checked exception; trừ lỗi (Error), ngoại lệ runtime (runtime exception) và các lớp con của hai loại này
  - Ví dụ: Người dùng mở một tệp có tên là DATA.txt để đọc nội dung, nhưng tệp này không tồn tại trong hệ thống
    - Luồng hoạt động mong đợi: Tệp tồn tại và dữ liệu được đọc từ tệp thành công
    - Thực tế: tệp không tồn tại => gây ra 1 ngoại lệ có kiểu FileNotFoundException
    - Ngoại lệ này thuộc phạm vi kiểm soát được của chương trình
    - Lập trình viên có thể viết mã để bắt loại ngoại lệ FileNotFoundException mỗi khi đọc tệp DATA.txt, và xử lý nó

Các loại ngoại lệ

- Đoạn chương trình sau sẽ bắn ra 1 ngoại lệ có kiểu FileNotFoundException nếu tệp DATA.txt không tồn tại
- Đoạn chương trình này chưa xử lý ngoại lệ nêu trên

```
File file = new File(”DATA.txt");
Scanner myReader = new Scanner(file);
while (myReader.hasNextLine()) {
    String data = myReader.nextLine();
    System.out.println(data);
}
myReader.close();
```

Các loại ngoại lệ

- Ngoại lệ runtime
  - Là một loại ngoại lệ không được kiểm tra (unchecked exception)
  - Là ngoại lệ xảy ra trong chương trình, chương trình **không** thể dự đoán được ngoại lệ và giải quyết được ngoại lệ đó
  - Nguyên nhân chính của loại ngoại lệ này xuất phát từ bug lập trình như dùng sai API hoặc viết mã sai logic. Các bug này không thể hiện rõ ràng cho đến khi chương trình hoạt động không như mong đợi.
  - Ví dụ: Người dùng truy cập một thuộc tính của đối tượng ref (null) sẽ gây ra ngoại lệ NullPointerException:

```
Object ref = null; // quên không khởi tạo ref
ref.toString(); // sẽ bắn ra ngoại lệ có kiểu
// NullPointerException
```

Các loại ngoại lệ

- Loại 3: Lỗi (Error)
  - Là một loại ngoại lệ không được kiểm tra (unchecked exception). Để cho đơn giản, gọi là lỗi thay vì ngoại lệ.
  - Là những lỗi xảy ra ở ngoài phạm vi kiểm soát của chương trình
    - Do đó, chương trình không thể dự đoán trước hoặc giải quyết được các lỗi này
  - Ví dụ: Chương trình mở một tệp thành công, đã sẵn sàng đọc dữ liệu từ tệp. Tuy nhiên, chương trình không thể đọc được nội dung tệp vì lỗi nào đó thuộc phần cứng hoặc thuộc hệ thống.
    - Trong trường hợp này, loại lỗi được bắn ra gọi là IOError
  - Một vài lỗi phổ biến khác
    - OutOfMemoryError (lỗi hết bộ nhớ RAM)
    - VirtualMachineError (lỗi máy ảo), v.v.

Xử lý ngoại lệ

- Để cho đơn giản, gọi mã nguồn ứng với luồng hoạt động mong đợi là thuật toán
- Hai cách xử lý ngoại lệ
  - Cách 1: Sử dụng cách truyền thống: viết trực tiếp mã nguồn xử lý ngoại lệ vào thuật toán
  - Cách 2: Phân tách mã nguồn xử lý ngoại lệ và thuật toán (sử dụng `try-catch/try-catch-finally`)
- Không nên sử dụng cách thứ 1 (làm rõ ở phần kế tiếp), thay vào đó nên sử dụng cách 2

Nhược điểm của cách xử lý ngoại lệ truyền thống

Nhược điểm của cách xử lý ngoại lệ truyền thống

Nhược điểm 1:

Phải cài đặt mã xử lý tại phương thức có ngoại lệ, từ đó làm cho chương trình trở nên khó hiểu

Ví dụ: để cho ngắn gọn, mã nguồn bên phải mô tả mã giả của một chương trình đọc nội dung file

Mã giả này chưa xử lý các kiểu ngoại lệ phổ biến như: (i) tệp không tồn tại, (ii) độ dài của tệp không thể xác định, (iii) bộ nhớ RAM không đủ để lưu nội dung tệp, v.v.

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Nhược điểm của cách xử lý ngoại lệ truyền thống

- Do đó, để cho mã giả này hoàn thiện hơn, cách xử lý ngoại lệ truyền thống sẽ dẫn đến mã giả như hình bên phải.
- Kết quả là, logic mã giả trở nên phức tạp hơn rất nhiều, làm cho mã giả sau khi chỉnh sửa khó hiểu hơn.

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Nhược điểm của cách xử lý ngoại lệ truyền thống

Nhược điểm 2:

Không hỗ trợ truyền ngoại lệ đến phương thức khác để xử lý. Nói cách khác, cách truyền thống bắt buộc phải xử lý tại phương thức xảy ra ngoại lệ.

Tuy nhiên, không phải lúc nào cũng đầy đủ thông tin để xử lý ngoại lệ tại phương thức có ngoại lệ.

Nhược điểm 3:

Dễ gây nhầm nhẫn khi xử lý ngoại lệ theo cách truyền thống do sự xuất hiện của những kiểu ngoại lệ có tính chất tương tự nhau

Nhược điểm 4:

Người lập trình thường quên không xử lý ngoại lệ, nguyên nhân bởi vì:

- bản chất con người
- thiếu kinh nghiệm, cố tình bỏ qua

Nhược điểm của cách xử lý ngoại lệ truyền thống

- Để giải quyết các nhược điểm của cách xử lý ngoại lệ truyền thống, nên sử dụng cấu trúc xử lý ngoại lệ được hỗ trợ trong Java
- Java hỗ trợ hai cấu trúc để xử lý ngoại lệ như sau
  - Cấu trúc `try-catch`
  - Cấu trúc `try-catch-finally`
- Hai cấu trúc này sẽ được diễn giải cụ thể trong phần kế tiếp

Các cấu trúc xử lý ngoại lệ trong Java

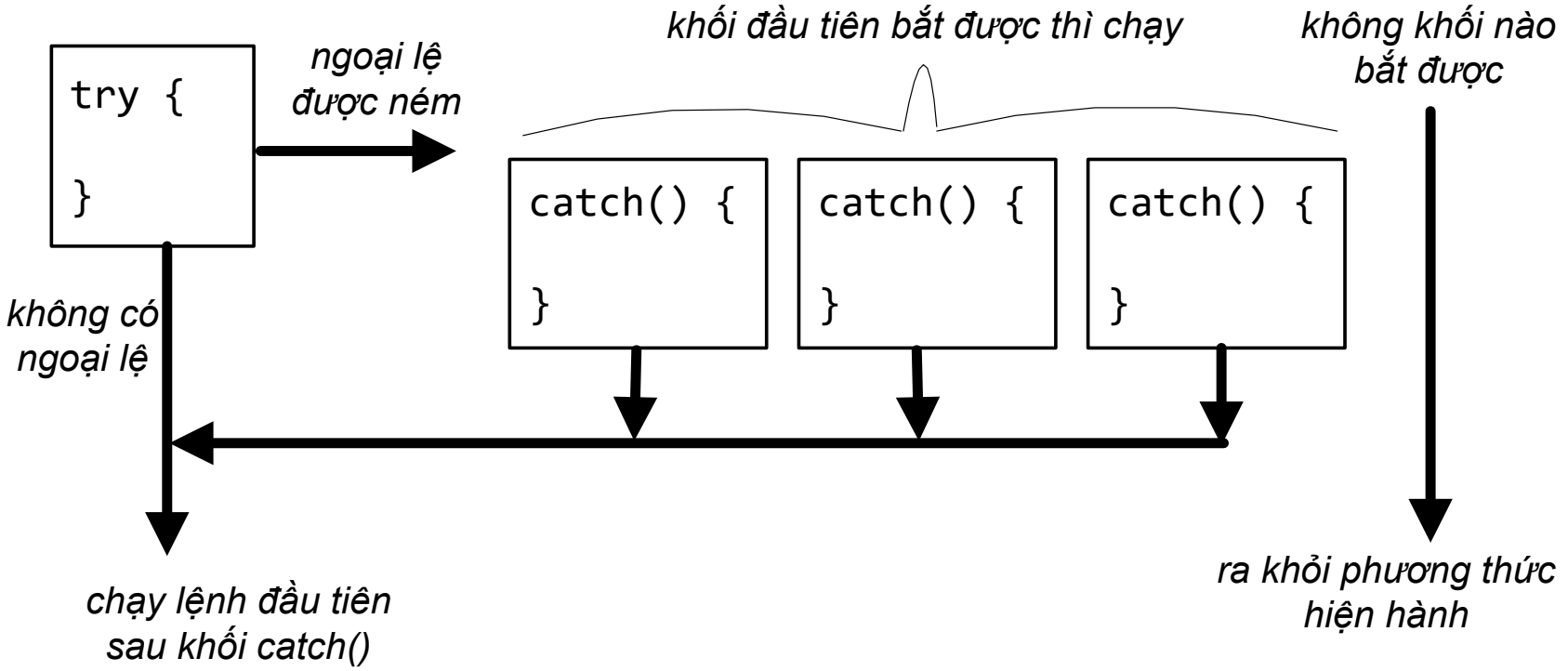
Cấu trúc try-catch

- Khối lệnh `try {...}`: khối lệnh có khả năng ném ngoại lệ
- Khối lệnh `catch() {...}`: bắt và xử lý ngoại lệ
- Có thể bắt nhiều kiểu ngoại lệ khác nhau bằng cách sử dụng nhiều khối lệnh `catch` đặt kế tiếp nhau
  - Khối lệnh `catch` sau không thể bắt ngoại lệ là lớp dẫn xuất của ngoại lệ được bắt trong khối lệnh `catch` trước

```
try {
    // …
    throw new
    Exception1();
} catch(Exception e) {
    // …(B)
} catch(Exception1 e){
    // …(A)
}
```

Cấu trúc `try-catch`

- Luồng thực thi với cấu trúc `try-catch`



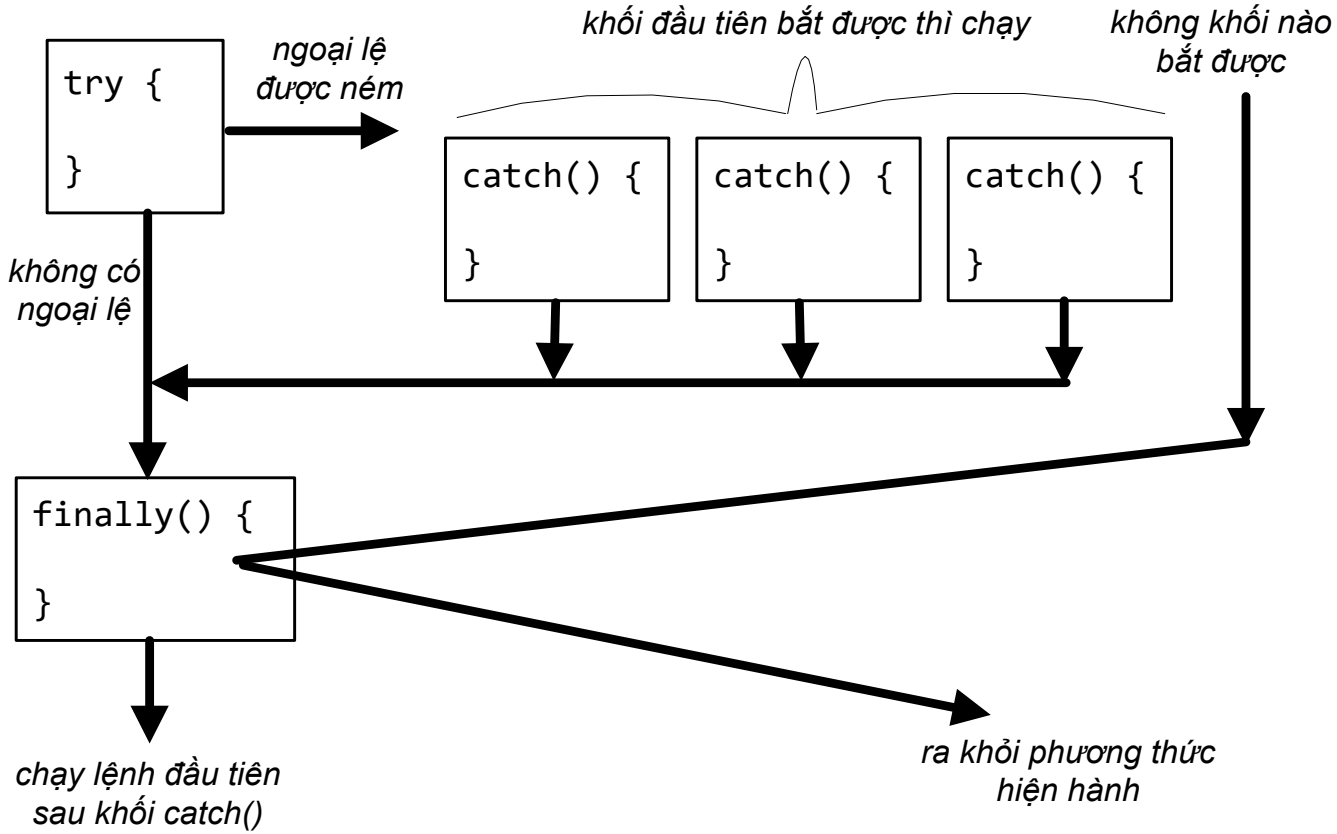
Cú pháp `try-catch-finally`

- Cú pháp `try-catch` + khối `finally`
- Khối lệnh `finally` có thể được đặt cuối cùng để thực hiện các công việc "dọn dẹp" cần thiết
  - `finally` luôn được thực hiện dù ngoại lệ có được bắt hay không
  - `finally` được thực hiện cả khi không có ngoại lệ được ném ra

```
try {
    // …
    throw new
        Exception1();
} catch(Exception e) {
    // …(B)
} catch(Exception1 e) {
    // …(A)
} finally {
    // …(C)
}
```

Cú pháp `try-catch-finally`

- Luồng thực thi với cấu trúc `try-catch-finally`



Cú pháp `try-catch-finally`

Ví dụ: Đọc một số nguyên từ bàn phím

```
InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader buf = new BufferedReader(reader);
try {
   // đọc dòng kế tiếp trong bộ nhớ đệm
   String str = buf.readLine();
   // chuyển nội dung dòng thành số nguyên và bọc trong lớp
   // Integer
   num = Integer.valueOf(str).intValue();
} catch (IOException e) { // Tệp không tồn tại
   System.err.println("IO Exception");
} finally {
   buf.close();
}
```

Quy trình JRE xử lý ngoại lệ

Quy trình JRE xử lý ngoại lệ

Gồm ba bước chính:

Bước 1: Khởi tạo đối tượng Exception

Khi một ngoại lệ xảy ra trong một phương thức, phương thức đó sẽ tạo một đối tượng có kiểu Exception, và chuyển đối tượng này cho JRE xử lý

- JRE là viết tắt của Java Runtime Environment
- Đối tượng Exception chứa thông tin về ngoại lệ, bao gồm kiểu ngoại lệ và trạng thái của chương trình khi ngoại lệ xảy ra
- Quá trình tạo một đối tượng Exception và chuyển đối tượng này cho JRE được gọi là ném ra ngoại lệ (*throw an exception*)

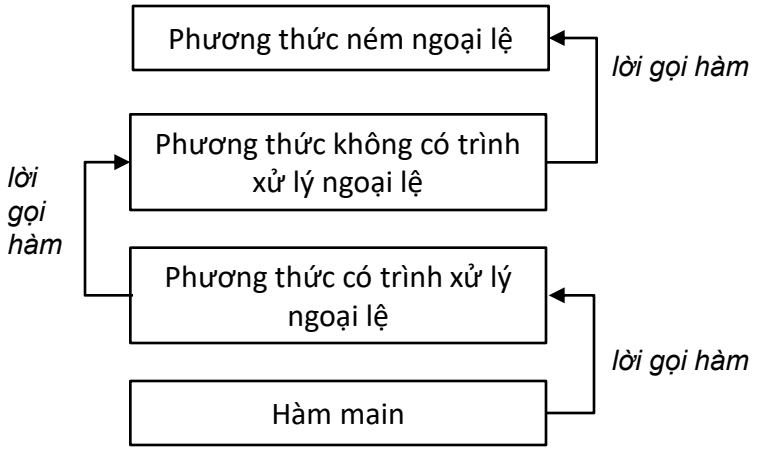
Quy trình JRE xử lý ngoại lệ

Bước 2: Tìm call stack có ngoại lệ bị ném ra

JRE sẽ cố gắng tìm một cách gì đó để xử lý ngoại lệ bị ném ra.

Tại thời điểm này, chương trình có ít nhất một ngăn xếp, gọi là *call stack*

- Một *call stack* sẽ chứa các phương thức được thực hiện trước khi ngoại lệ xảy ra
- Phương thức ném ra ngoại lệ nằm trên đỉnh *call stack*
- Hàm main nằm ở đáy *call stack*
- Có thể có nhiều *call stack* trong trường hợp dùng multi-thread

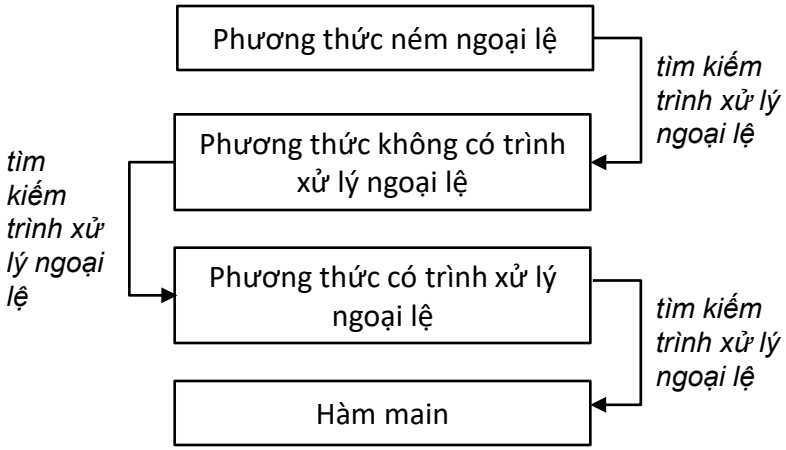


Cấu trúc một call stack

Quy trình JRE xử lý ngoại lệ

Bước 3. Xử lý ngoại lệ

- JRE bắt đầu tìm kiếm call stack có phương thức ném ra ngoại lệ
- Sau khi tìm thấy, JRE duyệt call stack tìm được để tìm phương thức có đoạn mã nguồn xử lý ngoại lệ ném ra
- Khi JRE tìm thấy phương thức, JRE chuyển ngoại lệ này cho phương thức xử lý
  - Đoạn mã nguồn xử lý ngoại lệ còn gọi là trình xử lý ngoại lệ (*exception handler*)
  - Quá trình tìm trình xử lý ngoại lệ được gọi là bắt ngoại lệ (*catch exception*)
  - Trình xử lý ngoại lệ được chọn phải xử lý ngoại lệ có kiểu khớp với kiểu ngoại lệ bị ném ra
  - Nếu JRE không tìm được trình xử lý ngoại lệ phù hợp, chương trình sẽ bị kết thúc



Quá trình tìm kiếm trình xử lý ngoại lệ trên call stack

Lần vết ngoại lệ `printStackTrace`

Có thể sử dụng phương thức `printStackTrace()` để lần vết vị trí phát sinh ngoại lệ

- Thông tin về các phương thức trong call stack bắn ra ngoại lệ sẽ được hiển thị ra

```
import java.io.*;
public class Util{
    public static float chia(int tuSo, int mauSo)
                                throws Exception{
        if (0 == mauSo)
         throw new Exception();
        else
            return tuSo / mauSo;
    }
    public static void main(String[] args){
        try{
            System.out.println(Util.chia(1, 0));
        } catch(Exception e){
            e.printStackTrace();
        }
    }
}

// Output
java.lang.Exception
   at Util.chia(Util.java:5)
   at Util.main(Util.java:11)
```

Ném ngoại lệ

Ném ngoại lệ

- Mọi phương thức đều có khả năng xảy ra một trong ba loại ngoại lệ
- Từ cách JRE xử lý ngoại lệ, khi một phương thức xảy ra ngoại lệ, phương thức này sẽ nằm ở đỉnh call stack. Hai cách xử lý gồm:
  - Ngoại lệ có thể được xử lý tại phương thức đó
  - Ngoại lệ được chuyển xuống các phương thức phía dưới call stack để xử lý (ném ngoại lệ cho phương thức khác xử lý)

Cú pháp ném ngoại lệ

- Cú pháp ném ngoại lệ ra khỏi phương thức có ngoại lệ

```
func_return func_name(args...)
    throws exception_types {
   ...
   throw throwable_object;
   ...
},
```

- , trong đó:
  - `func_return`: kiểu trả về của phương thức
  - `func_name`: tên phương thức
  - `args...`: các tham số trong hàm nếu có
  - `exception_types`: danh sách các kiểu ngoại lệ được ném ra
  - `throwable_object`: một đối tượng ngoại lệ
- Chú ý vị trí dùng throws và throw trong cú pháp ném ngoại lệ

Cú pháp ném ngoại lệ

Ví dụ: Phương thức `chia` thực hiện phép chia tử số cho mẫu số

- Nếu mẫu số bằng 0, phương thức `chia` sẽ ném ra một ngoại lệ qua câu lệnh `throw new Exception();`
- Đồng thời, phương thức `chia` phải định nghĩa kiểu ngoại lệ sẽ ném ra khi viết prototype qua câu lệnh `throws Exception`
- Trong hàm `main`, thực hiện phép chia 1 cho 0. Phép chia này sẽ gây ngoại lệ `Exception` trong phương thức `chia`.
- Bởi vì, ngoại lệ `Exception` bị ném xuống phương thức dưới trong *call stack* (tức là `main`), `main` sẽ bắt lấy ngoại lệ này
- Khối `try-catch` đảm bảo bảo rằng ngoại lệ được xử lý ngay trong `main`
- Kết quả in ra "*Ngoại lệ*"

```
public static float chia(int tuSo, int mauSo)
          throws Exception{
    if (0 == mauSo)
          throw new Exception();
    else
        return tuSo / mauSo;
}
public static void main(String[] args){
    try{
        System.out.println(Util.chia(1, 0));
    } catch(Exception e){
        System.err.print("Ngoại lệ");
    }
}

// "Ngoại lệ"
```

Ném tiếp ngoại lệ

- Sau khi một phương thức bắt ngoại lệ, nếu thấy cần thiết chúng ta có thể ném lại chính ngoại lệ vừa bắt được để cho phương thức phía dưới trong call stack tiếp tục xử lý
- Để ném ngoại lệ, dùng từ khóa throw trong khối lệnh bắt ngoại lệ

```
try {
    ///...
}
catch (Exception e) {
    System.out.println(e.getMessage());
    // ném tiếp ngoại lệ cho phương thức phía dưới trong stack
    throw e;
}
```

Ưu điểm của ném ngoại lệ

- Dễ dàng chuyển quyền xử lý ngoại lệ đến phương thức có khả năng xử lý ngoại lệ đó
- Có thể chuyển quyền xử lý nhiều loại ngoại lệ khỏi một phương thức
- Tách khối lệnh xử lý ngoại lệ khỏi luồng hoạt động mong đợi của phương thức, từ đó làm chương trình dễ đọc hơn

Ngoại lệ tại các phương thức ghi đè

- Quy tắc 1: Nếu phương thức ở lớp cha ném ngoại lệ, phương thức ghi đè (override method) tại lớp dẫn xuất có thể:
  - không ném ngoại lệ (1.1)
  - ném ngoại lệ giống hệt hoặc là ngoại lệ con (1.2)
- Quy tắc 2: Nếu phương thức ở lớp cha không ném ngoại lệ, phương thức ghi đè tại lớp dẫn xuất có thể:
  - ném ra unchecked exception (2.1)
  - không ném ngoại lệ (2.2)

→ Đảm bảo bắt được ngoại lệ khi sử dụng cơ chế đa hình

Ngoại lệ tại các phương thức ghi đè

Ví dụ:

```
class A {
   public void methodA() {
   }
}
class B extends A {
   public void methodA() throws ArithmeticException {
        // đúng(thỏa mãn 2.1 do ArithmeticException là unchecked exception)
   }
}
class C extends B {
   public void methodA()throws Exception {
        // sai do Exception không phải là con của ArithmeticException
        // (trái quy tắc 1.2)
   }
}
class D extends B {
   public void methodA(){
        // đúng do bỏ ngoại lệ (thỏa mãn quy tắc 1.1)
   }
}
```

Các thao tác khác trên ngoại lệ

Hoán đổi ngoại lệ (checked → unchecked)

- Có thể đổi checked exception thành unchecked exception
  - Khi chưa biết nên làm gì
- Ví dụ
  - Hàm `wrapException` bắn ra ngoại lệ `IOException`
  - `IOException` là checked exception
  - Tuy nhiên, do muốn chuyển sang dạng `RuntimeException`, nên trong `catch` viết `throw new RuntimeException(e);`

```
void wrapException() {
    try {
          …
        throw new IOException();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

//
try {
    wrapException();
} catch (RuntimeException e){
    try {
        throw e.getCause();
    }
    catch (IOException e1) {
        …
    }
}
```

Tự định nghĩa ngoại lệ

- Tự tạo lớp ngoại lệ để phục vụ các mục đích riêng
- Lớp ngoại lệ mới phải kế thừa từ lớp `Exception` hoặc lớp con của lớp này
- Có thể cung cấp hai constructor
  - constructor mặc định (không tham số)
  - constructor nhận một tham số `String` và truyền tham số này cho phương thức khởi tạo của lớp cơ sở

```
class SimpleException
        extends Exception {
}

class MyException
        extends Exception {
    public MyException() {}
    public MyException(String msg) {
        super(msg);
    }
}
```

Khởi tạo đối tượng và xử lý ngoại lệ

- Làm thế nào để thông báo khi hàm khởi tạo đối tượng gặp lỗi
  - không có giá trị trả lại
- Một cách là khởi tạo với một trạng thái đặc biệt và hi vọng sẽ có đoạn mã chương trình kiểm tra trạng thái này
- Cách hợp lý hơn cả là ném ngoại lệ

Khởi tạo đối tượng và xử lý ngoại lệ

```
class InputFile {
    public InputFile(String fname) throws IOException {
    ...
    }
...
}
---
try {
    InputFile fin = new InputFile(”data.txt”);
    int n = fin.readInt();
    ...
}
catch (IOException e) {
    System.err.println(e.getMessage);
}
```

Ưu điểm của cách xử lý ngoại lệ trong Java

Ưu điểm

Ưu điểm 1: Phân tách mã nguồn xử lý ngoại lệ khỏi luồng hoạt động mong đợi của chương trình

- Xử lý ngoại lệ trong khối catch
- Luồng hoạt động mong đợi được viết trong khối try

Ví dụ

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Xử lý ngoại lệ theo cách truyền thống ✗

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Xử lý ngoại lệ bằng try-catch ✓

Ưu điểm

Ưu điểm 2: Cho phép chọn phương thức xử lý ngoại lệ (là các phương thức nằm trong call stack chứa phương thức bắn ra ngoại lệ) -> cho phép xử lý ngoại lệ ở phương thức mong muốn

Tổng kết

- Nắm được định nghĩa ngoại lệ và ba loại ngoại lệ
- Hiểu được nhược điểm của cách xử lý ngoại lệ truyền thống
- Hai cấu trúc xử lý ngoại lệ trong Java gồm try-catch và try-catch-finally
- Nắm được quy trình JRE xử lý ngoại lệ
- Ném ngoại lệ khỏi phương thức xảy ra ngoại lệ
- Định nghĩa ngoại lệ mới
- v.v.