

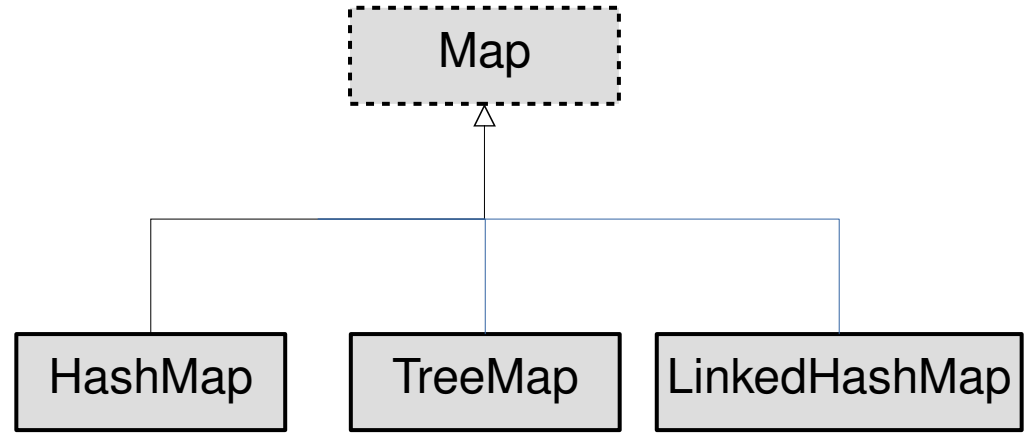
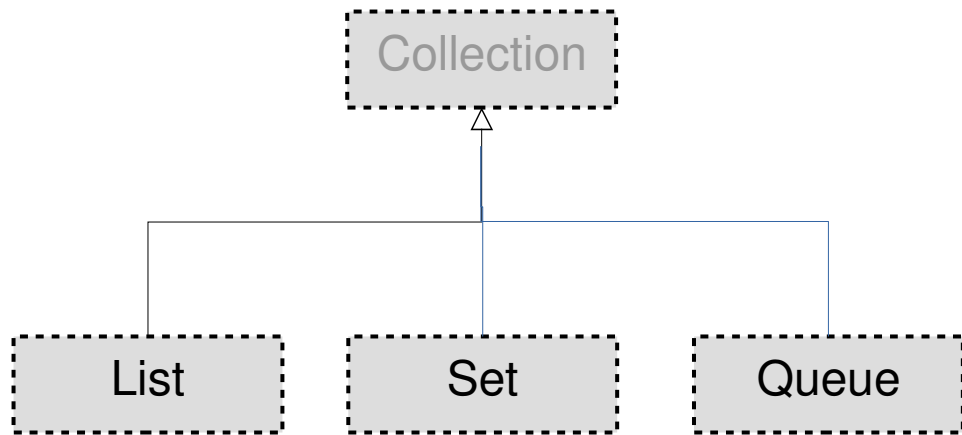
OOP with Java

Yuanbin Wu
cs@ecnu

OOP with Java

- 通知
 - Project 6: 6 月 6 日晚 9 点

- 复习
 - 容器



- 复习

- 类型安全容器

```
public class ApplesAndOrangesWithGenerics {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
        for(int i = 0; i < 3; i++)
            apples.add(new Apple());

        // Compile error!
        // apples.add(new Orange());
        for(int i = 0; i < apples.size(); i++)
            apples.get(i).id();
    }
    for(Apple c: apples)
        System.out.println(c.id());
}
```

OOP with Java

- 异常处理

异常处理

- 程序中的错误
 - 编译错误
 - 语法错误，类型错误 ...
 - 运行时错误

```
public class ErrorTypes {  
    public static void main(String[] args) {  
        If (2) { System.out.println("hello"); }  
        int a = new String("hello");  
        int [ ] array = new int[10];  
        for (int i = 0; i < 11; i++)  
            array[i] = i;  
        Integer t = null;  
        System.out.println(t.toString());  
    }  
}
```

ArrayIndexOutOfBoundsException
NullPointerException
ArithmeticException
....

Exceptions

异常处理

- C 语言如何处理程序中的错误？

如果 malloc() 失败？

```
void func()
{
    int *ptr = (int*)malloc(sizeof(int));
    If (!ptr) {
        printf("malloc fail");
        return;
    }
    ...
    *ptr = 2;
    ....
}
```

% man malloc

...
Return Value: The malloc() returns a pointer to the allocated memory. ... **On error, the function return NULL.** ...

异常处理

- C 语言如何处理程序中的错误？
 - 依靠程序员 "自觉"
 - 错误处理代码 "散落" 在各处
 - 程序易读性变差

异常处理

- Java 异常处理机制
 - 语法支持：语言本身包含异常处理的语法
 - 将正常代码与错误处理代码分开
 - 编译器支持：通过编译器强制错误检查

异常处理

■ Normal code
■ Handling error



C code



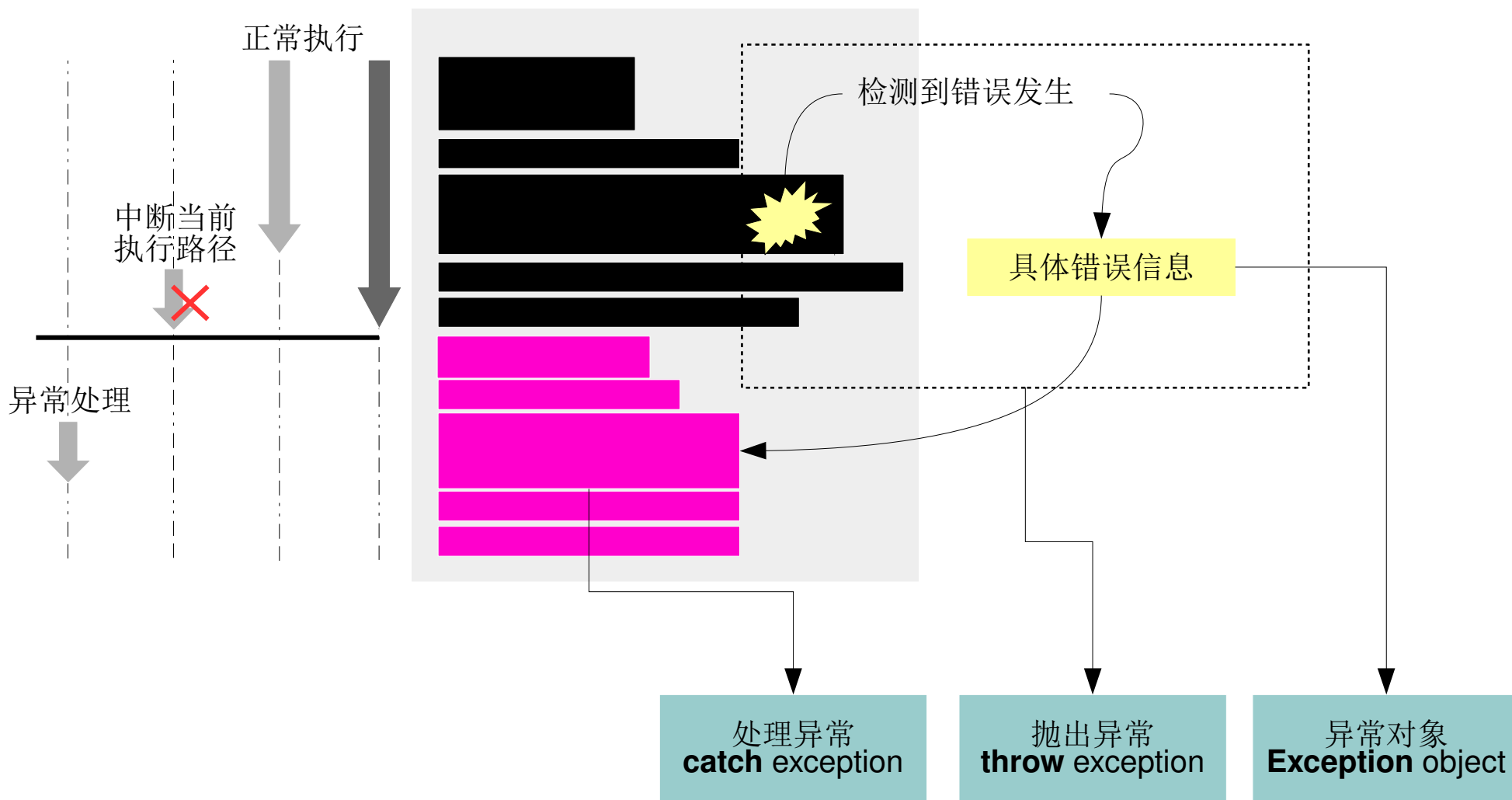
Java code

异常处理

- Java 错误处理场景
 - 某方法中发现错误
 - 中断当前执行路径
 - 创建 / 捕捉 Exception 类对象
 - 跳转到相应的异常处理代码段
 - 在代码段中处理该异常

异常处理

■ Normal code
■ Handling error



异常处理

- 抛出异常
- 处理异常
- 异常对象

异常处理

- 抛出异常
 - 检查错误条件
 - **throw** 关键字
 - 例子

```
if (t == null)  
    throw new NullPointerException();
```

throw 语句的含义为：

1. 发生了一个异常，请找到**合适的**异常处理模块处理
2. 该异常的具体信息存储在一个 **Exception 对象**中。

异常处理

- 处理异常
 - try 关键字
 - catch 关键字

```
try {  
    // 可能会抛出异常的代码  
}  
catch (Type1Exception e){  
    // 处理类型为 "Type1Exception" 的异常  
}  
catch (Type2Exception e){  
    // 处理类型为 "Type2Exception" 的异常  
}  
catch (Type3Exception e){  
    // 处理类型为 "Type3Exception" 的异常  
}  
...
```

1. catch 语句能处理对应的异常
2. 一旦发生异常立即跳转，不是等到所有的异常都发生
3. “On error goto”

异常处理

- 异常对象
 - Exception 类的子类
 - 通常情况
 - 不需要重写 Exception 类中的任何方法
 - Exception 类的方法
 - toString()
 - printStackTrace()


```
class SimpleException extends Exception { }

public class InheritingExceptions {

    public static void main(String[] args) {
        try {
            System.out.println("Throw SimpleException from f()");
            throw new SimpleException();
        } catch(SimpleException e) {
            System.out.println("Caught it!");
            System.out.println(e);
            System.out.println(e.printStackTrace(System.out));
        }
    }
}
```

异常处理

- 异常对象
 - catch 任意类型的对象

```
try{  
    ...  
}  
catch (Exception e){  
    ...  
}
```

```
class SimpleException extends Exception { }

public class InheritingExceptions {

    public static void main(String[] args) {
        try {
            System.out.println("Throw SimpleException from f()");
            throw new SimpleException();
        } catch (Exception e) {
            System.out.println("Caught it!");
            System.out.println(e);
            System.out.println(e.printStackTrace(System.out));
        }
    }
}
```

异常处理

- 抛出异常
 - throw
- 处理异常
 - try, catch
- 异常对象
 - Exception 类的子类

异常处理



- 基本异常处理
 - 在同一方法中完成 `throw`, `try`, `catch`
 - 在同一方法中完成正常代码与错误处理代码的隔离
- 在不同方法中完成正常代码与错误代码的隔离
 - 方法只抛出异常，而将异常交由别人处理
 - 方法中仅有 `throw`
 - 将抛出的异常交给该方法的调用者

异常处理

```
bar() {  
    ...  
    throw new Type1Exception ();  
    ...  
    throw new Type2Exception ();  
}
```

1. bar() 仅抛出异常，没有 try-catch
2. foo() 作为 bar() 的调用者处理 bar() 抛出的异常

问题：
foo() 如何知道 bar() 会抛出何种类型的异常？

```
foo() {  
    try{  
        ...  
        bar();  
    }  
    catch (Type1Exception e){  
        ...  
    }  
    catch (Type2Exception e){  
        ...  
    }  
}
```

异常处理

- 类方法的异常说明 (**Exception Specification**)
 - 标识该方法可能会抛出何种类型的异常
 - **throws** 关键字

```
bar() throws Type1Exception, Type2Exception{  
    ...  
    throw new Type1Exception ();  
    ...  
    throw new Type2Exception ();  
}
```

```
foo() {  
    try{  
        ...  
        bar();  
    }  
    catch (Type1Exception e){  
        ...  
    }  
    catch (Type2Exception e){  
        ...  
    }  
}
```



```
class SimpleException extends Exception { }

public class InheritingExceptions {
    // compile error if no throws
    public void f() throws SimpleException {
        System.out.println("Throw SimpleException from f()");
        throw new SimpleException();
    }

    public static void main(String[] args) {
        InheritingExceptions sed = new InheritingExceptions();
        try {
            sed.f();
        } catch(SimpleException e) {
            System.out.println("Caught it!");
        }
    }
}
```

异常处理

- 类方法的异常说明 (**Exception specification**)
 - 标识该方法可能会抛出何种类型的异常
 - **throws** 关键字
- **编译器保证**，如果方法使用了 **throws** 关键字，则在该方法的调用处必须要处理相应的异常

```
bar() throws Type1Exception, Type2Exception{  
    ...  
    throw new Type1Exception ();  
    ...  
    throw new Type2Exception ();  
}
```

```
foo() {  
    // 编译错误：必须 catch Type1Exception 和 Type2Exception  
    bar();  
}
```

异常处理

- 类方法的异常说明 (**Exception specification**)
 - 标识该方法可能会抛出何种类型的异常
 - **throws** 关键字
- 包含 **throws** 关键字，但函数本身并不抛出相应异常
 - 用于 **interface, abstract method**
 - 保证重写的方法必须考虑所列出的异常

```
bar() throws Type1Exception, Type2Exception{  
    ...  
    // throw new Type1Exception ();  
    ...  
    // throw new Type2Exception ();  
}
```

```
foo() {  
    try{  
        ...  
        bar();  
    }  
    catch (Type1Exception e){  
        ...  
    }  
    catch (Type2Exception e){  
        ...  
    }  
}
```

异常处理

- 类方法的异常说明 (**Exception specification**)
 - 标识该方法可能会抛出何种类型的异常
 - **throws** 关键字
- 如果不包含 **throws** 关键字
 - 默认会抛出 **RuntimeException** 类型的异常

```
bar() {  
    ...  
}
```

```
foo() {  
    try{  
        ...  
        bar();  
    }  
    catch (RuntimeException e){  
        ...  
    }  
}
```

异常处理

- 重新抛出异常
 - 调用函数可以将 `catch` 到的异常重新抛出
 - 交给调用者的调用者来处理

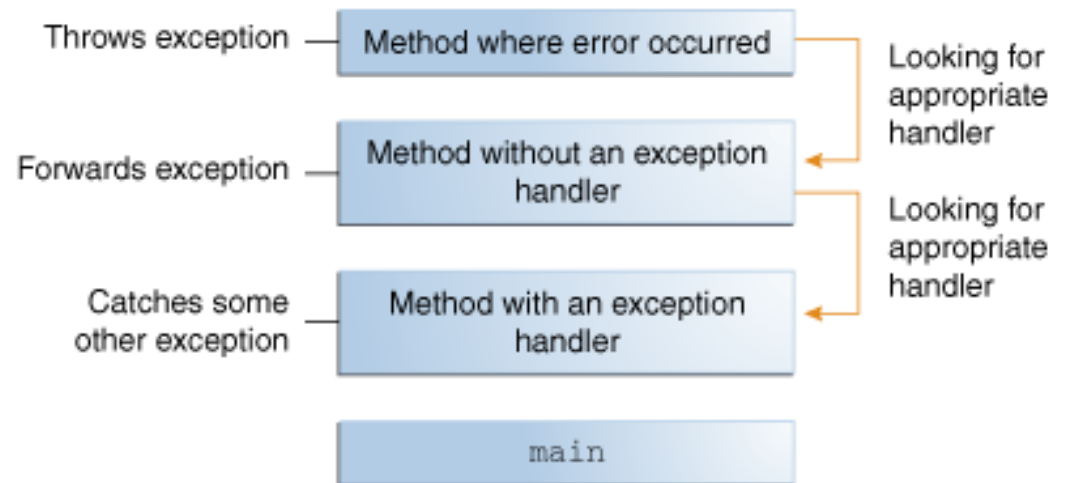
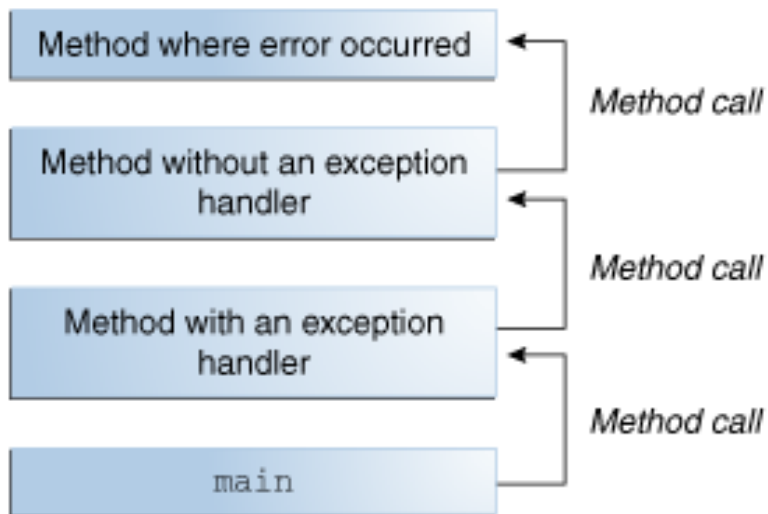

```
bar() throws Type1Exception, Type2Exception{
    ...
    throw new Type1Exception ();
    ...
    throw new Type2Exception ();
}
```

```
foo() throws Type1Exception{
    try{
        ...
        bar();
    }
    catch new Type1Exception (){
        throw e;
    }
    catch new Type2Exception (){
        ...
    }
}
```

```
test() {
    try{
        foo();
    }
    catch (Type1Exception e){
        ...
    }
}
```

异常处理

- 总结：让其他方法处理异常
 - 在运行到 **throw** 处终止当前方法
 - “**return**” 相应的异常
 - 逐层寻找 **catch** 语句，直到最后该异常得到处理



异常处理

- Java 标准异常
 - 都为 `Exception` 类的子类
 - 通过名字能知道含义
 - `IOException`
 - `RuntimeException`
 - `ArrayOutOfBoundsException`
 - `NullPointerException`
 - `SQLException`
 - ...

异常处理

- **RuntimeException**

- **Java** 提供了许多现成的异常，帮助用户更方便的使用异常处理机制
- 例如
 - 当数组越界时，自动抛出 **ArrayOutOfBoundsException**
 - 当访问 **null** 的成员时，自动抛出 **NullPointerException**
 - 当除以 **0** 时，自动抛出 **ArithmeticException**

```
if (t == null)  
    throw new NullPointerException();
```

不必要！

异常处理

- **RuntimeException**
 - 表示程序有 bug
 - 通常情况下不会主动抛出 `RuntimeException`
 - 通常情况下不需要 `catch RuntimeException`
 - 由 `main` 函数自动 `catch`
 - 调用 `printStackTrace()`

异常处理

- **finally** 关键字
 - 无论 **try** 语句中是否有异常抛出，都会执行

```
try {  
    // 可能会抛出异常的代码  
}  
catch (Type1Exception e){  
    // 处理类型为 "Type1Exception" 的异常  
}  
finally{  
    // 无论是否有异常被抛出，总会被执行  
}
```

```
class ThreeException extends Exception { }

public class FinallyWorks {
    static int count = 0;
    public static void main(String[] args) {
        while(true) {
            try {
                if(count++ == 0)
                    throw new ThreeException();
                System.out.println("No exception");
            }
            catch(ThreeException e) {
                System.out.println("ThreeException");
            }
            finally {
                System.out.println("In finally clause");
                if(count == 2) break; // out of "while"
            }
        }
    }
}
```

异常处理

- **finally** 关键字
 - 无论 **try** 语句中是否有异常抛出，都会执行
 - **return**


```
public class MultipleReturns {
    public static void f(int i) {
        print("Initialization that requires cleanup");
        try {
            System.out.println("Point 1");
            if(i == 1)
                return;
            System.out.println("Point 2");
            if(i == 2)
                return;
            System.out.println("Point 3");
            if(i == 3)
                return;
            System.out.println("End");
            return;
        } finally {
            System.out.println("Performing cleanup");
        }
    }
    public static void main(String[] args) {
        for(int i = 1; i <= 4; i++)
            f(i);
    }
}
```

异常处理

- **finally** 关键字
 - 无论 **try** 语句中是否有异常抛出，都会执行
 - **return**
 - 无论是否有对应的 **catch** 语句，都会执行

```
class FourException extends Exception { }

public class AlwaysFinally {
    public static void main(String[] args) {
        System.out.println("Entering first try block");
        try {
            System.out.println("Entering second try block");
            try {
                throw new FourException();
            } finally {
                System.out.println("finally in 2nd try block");
            }
        } catch (FourException e) {
            System.out.println("Caught FourException in 1st try block");
        } finally {
            System.out.println("finally in 1st try block");
        }
    }
}
```

异常处理

- **finally** 的作用
 - 帮助保证一致性
 - 简化代码
- 例子
 - 电灯开关 : **on/off**
 - 保证在程序运行结束，开关最终处于 **off** 状态

```
public class Switch {  
    private boolean state = false;  
    public boolean read() { return state; }  
    public void on() { state = true; }  
    public void off() { state = false; }  
    public String toString() { return state ? "on" : "off"; }  
}
```

```

public class OnOffException1 extends Exception { }
public class OnOffException2 extends Exception { }

public class OnOffSwitch {
    private static Switch sw = new Switch();
    public static void f() throws OnOffException1, OnOffException2 {}
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
            f();
            sw.off();
        } catch (OnOffException1 e) {
            System.out.println("OnOffException1");
            sw.off();
        } catch (OnOffException2 e) {
            System.out.println("OnOffException2");
            sw.off();
        }
    }
}

```

无 finally 语句：

1. 每处异常都需手动 `sw.off()`
2. 如果异常没有被 `catch`，则最终开关不处于 `off` 状态 !!

异常处理

- 继承，接口与异常
 - 父类 / 接口的方法有异常说明 (**throws** 关键字)
 - 子类 / 实现重写该方法时
 - 有同样的异常说明
 - 有 "更少" 的异常说明
 - 抛出子异常
 - 原因？
 - **upcasting**

```
class SimpleException extends Exception { }
class MoreSimpleException extends SimpleException { }
class MyException extends Exception { }
```

```
interface Inter{
    void f1() throws SimpleException;
    void f2();
    void f3() throws SimpleException;
    void f4() throws MyException;
    void f5() throws SimpleException;
}
```

```
class Impl implements Inter{
    // OK, 异常说明相同
    public void f1() throws SimpleException {System.out.println("In Impl");}

    // compile error, 子类可能抛出异常, 父类却没有说明该异常
    // public void f2() throws SimpleException {System.out.println("In Impl");}

    // OK, 子类异常说明比父类少
    public void f3() {System.out.println("In Impl");}

    // compile error, 异常说明不同, 不能重写
    // public void f4() throws SimpleException {System.out.println("In Impl");}

    // OK, 子类可以抛出子异常
    public void f5() throws MoreSimpleException {System.out.println("In Impl");}
}
```

1. **Upcasting:** 父类出现的地方可以用子类代替
2. 对父类操作的代码会 **catch** 父类方法抛出的异常
3. 因此, 子类在重写方法时不能抛出其他类型的异常, 否则无法 **catch**

总结

- 语法
 - 抛出异常 : `throw`
 - 处理异常 : `try, catch`
 - 异常对象 : `Exception` 类的子类
- 从方法中抛出异常
 - 方法的异常说明 : `throws`
 - 中断当前方法的执行，返回抛出的异常对象，在该方法的调用路径上寻找合适的 `catch`.