

第七章第二节 错误与异常处理

本节关注：Java中错误和异常处理的典型技术——把原理落实到代码上！

Outline：

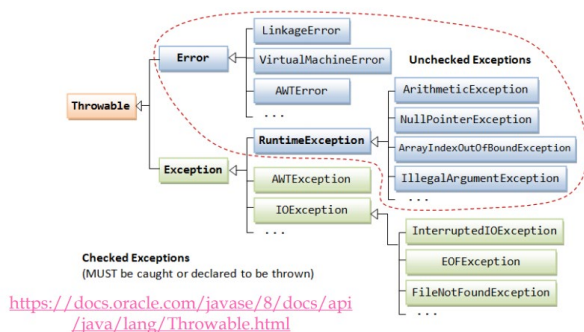
- Java中的错误和异常（`java.lang.throwable`）
- 异常
 - Runtime异常与其他异常（`Exception`）
 - Checked异常和unchecked异常
- checked异常的处理机制
- 自定义异常

Notes：

Java中的错误和异常

【Throwable】

- [Java.lang.throwable](#)
 - Throwable 类是 Java 语言中所有错误或异常的超类。
 - 继承的类：extends Object。
 - 实现的接口：implements Serializable。
 - 直接已知子类：Error, Exception（直接已知子类：IOException、RuntimeException）。



【Error】

- Error类描述很少发生的Java运行时系统内部的系统错误和资源耗尽情况（例如，`VirtualMachineError`，`LinkageError`）。
- 对于内部错误：程序员通常无能为力，一旦发生，想办法让程序优雅地结束
- Error的类型：
 - 用户输入错误

- 例如：用户要求连接到语法错误的URL，网络层会投诉。
- 设备错误
 - 硬件并不总是做你想做的。
 - 输出器被关闭
- 物理限制
 - 磁盘可以填满
 - 可能耗尽了可用内存
- 典型错误：
 - **VirtualMachineError**: thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.
 - **OutOfMemoryError**: thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.
 - **StackOverflowError**: thrown when a stack overflow occurs because an application recurses too deeply.
 - **InternalError**: Thrown to indicate some unexpected internal error has occurred in the JVM.
 - **LinkageError**: a class has some dependency on another class; however, the latter class has incompatibly changed after the compilation of the former class.
 - **NoClassDefFoundError**: Thrown if JVM or a **ClassLoader** instance tries to load in the definition of a class but no definition could be found.

异常 (Exception)

- 异常：程序执行中的非正常事件，程序无法再按预想的流程执行。
- 异常处理：
 - 将错误信息传递给上层调用者，并报告“案发现场”的信息。
 - **return**之外的第二种退出途径：若找不到异常处理程序，整个系统完全退出

【异常按结构层次的分类】

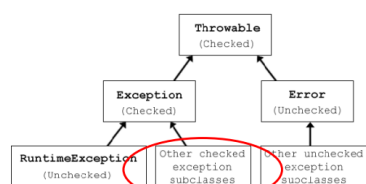
- 运行时异常：由程序员处理不当造成，如空指针、数组越界、类型转换
- 其他异常：程序员无法完全控制的外在问题所导致的，通常为IOE异常，即找不到文件路径等

【异常按处理机制角度的分类】

- 为什么区分checked 和 unchecked：原因其实很简单，编译器将检查你是否为所有的已检查异常提供了异常处理机制，比如说我们使用Class.forName()来查找给定的字符串的class对象的时候，如果没有为这个方法提供异常处理，编译是无法通过的。
- Checked exception：
 - 编译器可帮助检查你的程序是否已抛出或处理了可能的异常
 - 异常的向上抛出机制进行处理，如果子类可能产生A异常，那么在父类中也必须throws A异常。可能导致的问题：代码效率低，耦合度过高。
 - checked exception是需要强制catch的异常，你在调用这个方法的时候，你如果不catch这个异常，那么编译器就会报错，比如说我们读写文件的时候会catch IOException，执行数据库操作会有SQLException等。
 - 对checked Exception处理机制
 - 抛出：声明是throws，抛出时throw
 - 捕获 (try/catch)：try出现异常，忽略后面代码直接进入catch；无异常不进入catch；若catch中没有匹配的异常处理，程序退出；若子类重写了父类方法，父类方法没有抛出异常，子类应自己处理全部异常而不再传播；子类从父类继承的方法不能增加或更改异常
 - 处理：不能代替简单的测试，尽量苛刻、不过分细化、将正常处理与异常处理分开、利用好层次结构、早抛出晚

捕获、避免不必要的检查

- 清理现场、释放资源（finally）：finally中语句不论有无异常都执行



必须捕获并指定错误处理器handler，否则编译无法通过

类似于编程语言中的static type checking

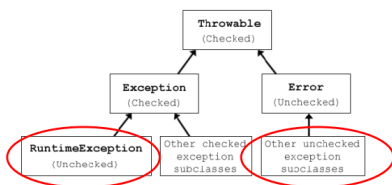
```

import java.io.*;

public class className
{
    public void deposit(double amount) throws RemoteException
    {
        // Method implementation
        throw new RemoteException();
    }
    //Remainder of class definition
}
  
```

• unchecked exception:

- 程序猿对此不做任何事情，不得不重写你的代码（不需要在编译时使用try-catch等机制处理）
- 这类异常都是RuntimeException的子类，它们不能通过client code来试图解决
- 这种异常不是必须需要catch的，你是无法预料的，比如说你在调用一个list.size()的时候，如果这个list为null，那么就会报NullPointerException，而这个异常就是 RuntimeException，也就是UnChecked Exception
- 常见的unchecked exception：JVM抛出，如空指针、数组越界、数据格式、不合法的参数、不合法的状态、找不到类等



可以不处理，编译没问题，但执行时出现就导致程序失败

类似于编程语言中的dynamic type checking

```

public class NullPointerExceptionExample {
    public static void main(String args[]){
        String str=null;
        System.out.println(str.trim());
    }
}
  
```

Exception in thread "main" java.lang.NullPointerException

【checked和unchecked总结】

- 当要决定是采用checked exception还是Unchecked exception的时候，问一个问题：“如果这种异常一旦抛出，client会做 怎 样的补救？”
 - 如果客户端可以通过其他的方法恢复异常，那么采用checked exception；
 - 如果客户端对出现的这种异常无能为力，那么采用unchecked exception；
 - 异常出现的时候，要做一些试图恢复它的动作而不要仅仅的打印它的信息。
- 尽量使用unchecked exception来处理编程错误：因为uncheckedexception不用使客户端代码显式的处理它们，它们自己会在出现的地方挂起程序并打印出异常信息。
- 如果client端对某种异常无能为力，可以把它转变为一个unchecked exception，程序被挂起并返回客户端异常信息

– Checked exception应该让客户端从中得到丰富的信息。

– 要想让代码更加易读，倾向于用unchecked exception来处理程序中的错误

	Checked exception	Unchecked exception
Basic	The compiler checks the checked exception. If we do not handle the checked exception, then the compiler objects. 必须被显式地捕获或者传递 (try-catch-finally-throw)，否则编译器无法通过	The compiler does not check the Unchecked exception. Even if we do not handle the unchecked exception, the compiler doesn't object. 异常可以不必捕获或抛出，编译器不去检查
Class of Exception	Except RuntimeException class, all the child classes of the class “ Exception ”, and the “ Error ” class and its child classes are Checked Exception. 继承自Exception类	RuntimeException class and its child classes, are Unchecked Exceptions. 继承自RuntimeException类
Handling	从异常发生的现场获取详细的信息，利用异常返回的信息来明确操作失败原因，并加以合理的恢复处理	简单打印异常信息，无法再继续处理
Appearance	代码看起来复杂，正常逻辑代码和异常处理代码混在一起	清晰，简单

checked异常的处理机制

【异常中的LSP原则】

- 如果子类型中override了父类型中的函数，那么子类型中方法抛出的异常不能比父类型抛出的异常类型更广泛
- 子类型方法可以抛出更具体的异常，也可以不抛出任何异常
- 如果父类型的方法未抛出异常，那么子类型的方法也不能抛出异常。
- 其他的参考第五章第二节的LSP

【利用throws进行声明】

- 使用throws声明异常：此时需要告知你的client需要处理这些异常，如果client没有handler来处理被抛出的checked exception，程序就终止执行。
- 程序员必须在方法的spec中明确写明本方法会抛出的所有checked exception，以便于调用该方法的client加以处理
- 在使用throws时，方法要在定义和spec中明确声明所抛出的全部checked exception，没有抛出checked异常，编译出错，Unchecked异常和Error可以不用处理。

【利用throw抛出一个异常】

- 步骤：
 - 找到一个能表达错误的Exception类/或者构造一个新的Exception类
 - 构造Exception类的实例，将错误信息写入
 - 抛出它
- 一旦抛出异常，方法不会再将控制权返回给调用它的client，因此也无需考虑返回错误代码


【try-catch语句】

- 使用 try 和 catch 关键字可以捕获异常。try/catch 代码块放在异常可能发生的地方。


- `try/catch`代码块中的代码称为保护代码，
- `Catch` 语句包含要捕获异常类型的声明。当保护代码块中发生一个异常时，`try` 后面的 `catch` 块就会被检查。
- 如果发生的异常包含在 `catch` 块中，异常会被传递到该 `catch` 块，这和传递一个参数到方法是一样。

【finally语句】

- 场景：当异常抛出时，方法中正常执行的代码被终止；但如果异常发生前曾申请过某些资源，那么异常发生后这些资源要被恰当的清理，所以需要`finally`语句。
- `finally` 关键字用来创建在 `try` 代码块后面执行的代码块。
- 无论是否发生异常，`finally` 代码块中的代码总会被执行。
- 在 `finally` 代码块中，可以运行清理类型等收尾善后性质的语句。
- `finally` 代码块出现在 `catch` 代码块最后：
- 注意下面事项：
 - `catch` 不能独立于 `try` 存在。
 - 在 `try/catch` 后面添加 `finally` 块并非强制性要求的。
 - `try` 代码后不能既没 `catch` 块也没 `finally` 块。
 - `try`, `catch`, `finally` 块之间不能添加任何代码。



```
1 public class ExcepTest{
2     public static void main(String args[]){
3         int a[] = new int[2];
4         try{
5             System.out.println("Access element three :" + a[3]);
6         }catch(ArrayIndexOutOfBoundsException e){
7             System.out.println("Exception thrown  :" + e);
8         }
9         finally{
10            a[0] = 6;
11            System.out.println("First element value: " +a[0]);
12            System.out.println("The finally statement is executed");
13        }
14    }
15 }
```



自定义异常

- 如果JDK提供的`exception`类无法充分描述你的程序发生的错误，可以创建自己的异常类。
 - 如果希望写一个检查性异常类，则需要继承 `Exception` 类。
 - 如果你想写一个运行时异常类，那么需要继承 `RuntimeException` 类。

抛出检查型异常：

```
public class FooException extends Exception {
    public FooException() { super(); }
    public FooException(String message) { super(message); }
    public FooException(String message, Throwable cause) {
        super(message, cause);
    }
    public FooException(Throwable cause) { super(cause); }
}

try {
    ...
} catch(FooException ex) {
    ex.printStackTrace();
    System.exit(1);
} catch(IOException ex) {
    throw new FooException(ex);
}
```

抛出**unchecked exception**:

```
class FileFormatException extends IOException {  
    public FileFormatException() {}  
    public FileFormatException(String gripe){  
        super(gripe);  
    }  
}  
  
String readData(BufferedReader in) throws FileFormatException {  
    ...  
    while (. . .){  
        if (ch == -1) { // EOF encountered  
            if (n < len) {  
                String errorInfo = ...;  
                throw new FileFormatException(efforInfo);  
            }  
        }  
        ...  
    }  
    return s;  
}
```

