Lecture2-3 异常与异常处理

1. 异常的介绍

- 在 Java 中，异常通常用于控制指令执行流，以处理不同的情况
- 在 c++ 和 Lisp 中，它们用于异常和不可预测的错误

处理程序错误有两个方面：**检测 detection** 和**处理 handling**

- 例如，Scanner 构造函数可以检测从不存在的文件中读取的尝试，但是，它不能处理这个错误
- 处理错误的一种令人满意的方法可能是终止程序，或者向用户请求另一个文件名
- Scanner 类不能在这些选项中进行选择，它需要将错误报告给程序的另一部分
- 在 Java 中，异常处理提供了一种灵活的机制，可以将控制从错误检测点传递给能够处理错误的处理程序

异常是对象

- Java 中的异常是由“异常事件”生成的特殊对象 (通常是一个错误)，它通过一种不同于对象间通常消息传递的机制传递回调用方法

throw

```
throw new MyException();
```

- 抛出的是一个对象，不是一个类型
- 由于异常是一个对象，所以必须在需要 throw 时使用 new 实例化（从类定义为类型的模板创建）并抛出它

```
if(amount > balance){
    // 一个 new 出来的 exception 对象被 throw 出来
    throw new IllegalArgumentException("Amount exceeds balance");
    // 大多数异常对象的构造器里可以包含一个错误信息
}
balance -= amount; // 如果一个异常被抛出，那么这行语句不会被执行
```

throws

由于一些异常经常发生，一些方法通过在 throws 后加上可能抛出的异常的名称来发出警告，然后让 javac 编译器知道它们

```
class FileReader{
    public FileReader(String filename) throws FileNotFoundException{
        //...
    }
}
```

如果你忽略了方法种 throws 的异常

```
import java.io.File;
import java.io.FileReader;
public class IgnoredException{
    public static void main(String args[]){
        FileReader fr = new FileReader("sample.txt"); // 不可以
    }
}
```

- javac 不会让你这么做的，它的信息非常明确：它希望你要么处理问题，要么让世界知道某些东西可能会失败，而另一个对象必须处理它

错误 Error 的类型

编译时 Compile Time

- 语法错误 Syntax Errors
- 错误的类型 Wrong Type
    - 很多事情都可能出错，所以我们有不同类型的错误，当 Javac 将 .java 文件编译为 .class java 字节码文件时，它会告知错误的语法，或者在分配数据或调用方法时不兼容的类型，在考虑运行程序之前，必须解决所有这些问题

链接时 Link Time

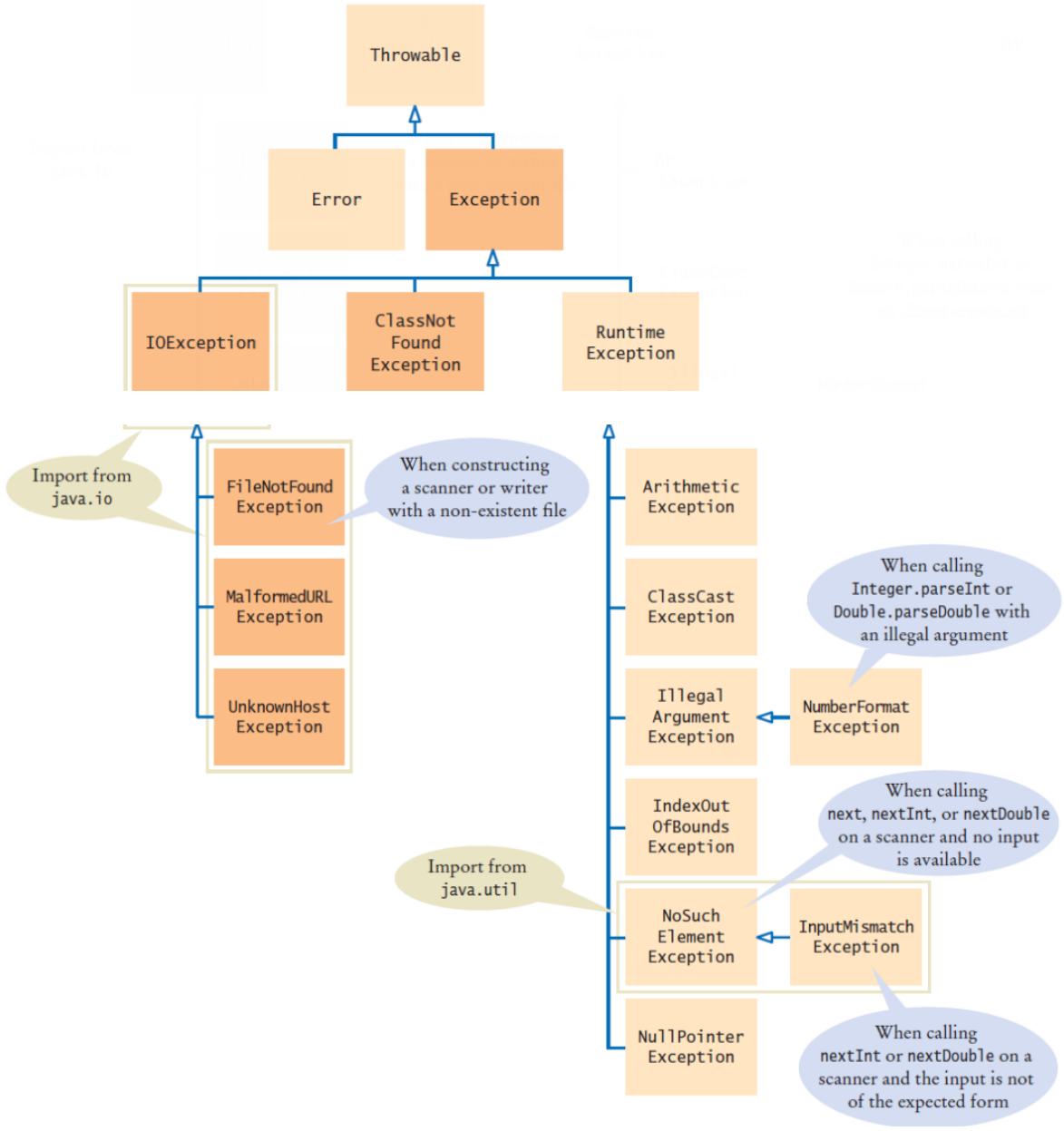
- 在你可以运行你的程序之前，类加载器必须加载它，你可能已经通过编译和失败的链接，因为类加载器不能找到 .class 对应于要使用的对象

运行时 Run Time

- 被应用检测到
- 被库检测到
- 被操作系统 / 硬件检测到
    - 当你运行你的程序时，你可能会遇到其他错误;，的应用程序方法之一可以检测到它获取的参数类型正确，但值范围错误，内置的 Java 方法可能会发现相同的情况(并抛出异常)，或者，事情可能真的出错了，操作系统可能会发现(例如，硬件故障)，逻辑也可能是错误的
    - 这些被称为**异常 Exceptions**

2. Java 异常 Exception 的类型

在 Java 中，异常分为三类



系统错误 Error

- **系统内部的错误**：Error 的子类，一个例子是 OutOfMemoryError，当所有可用的计算机内存都用完时抛出该错误，这些是很少发生的致命错误，我们在本书中将不考虑它们
- 主要是资源出现了错误
  - **file**：FileNotFoundException, SecurityException, IOException
  - **network**：SocketException
  - **memory**：OutOfMemoryError, StackOverflowError
- 在代码中或者设备中试着处理这些问题，大多数情况下如果不能处理，程序必须退出

未检查异常 Unchecked Exception

- **运行时错误**：**RuntimeException 的子类**，如 IndexOutOfBoundsException 或IllegalArgumentException 表示代码中的错误，它们被称为**未检查的异常 unchecked exceptions**
- 主要是代码的 bug
    - NullPointerException, IllegalArgumentException, IndexOutOfBoundsException
- 记录异常，修改 bug 如果问题比较严重，可以在界面上显示告警信息

检查异常 Checked Exception

- 其他异常都是**检查异常 checked exceptions**：这些异常表明由于某些超出你控制的外部原因而出现了错误
- 可能是用户的使用出现异常
- 提示用户进行正确的操作，统计和评估恶意用户请求

如果方法声明的是 Exception 类型的异常或者是 Checked Exception 异常，要求方法的调用处必须做处理

- 继续使用 throws 向上(方法的调用处)声明
- 使用 try-catch-finally 进行处理

3. 处理异常

当你面临处理**检查异常 checked exception** 的时候，javac 要求你按照规则处理

有两种处理 checked exception 的方法

捕获并处理异常

```
try{
    File inFile = new File(filename);
    Scanner in = new Scanner(inFile); // 抛出 FileNotFoundException
}catch(FileNotFoundException exception)
{
    //...
}
```

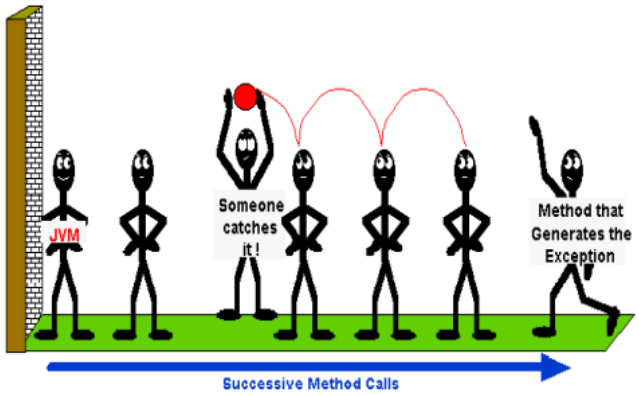
抛出异常

```
public void readData(String filename) throws FileNotFoundException{
    // 必须指明这个方法可能抛出的所有的检查异常 checked exception
    File inFile = new File(filename);
    Scanner in = new Scanner(infile);
}
```

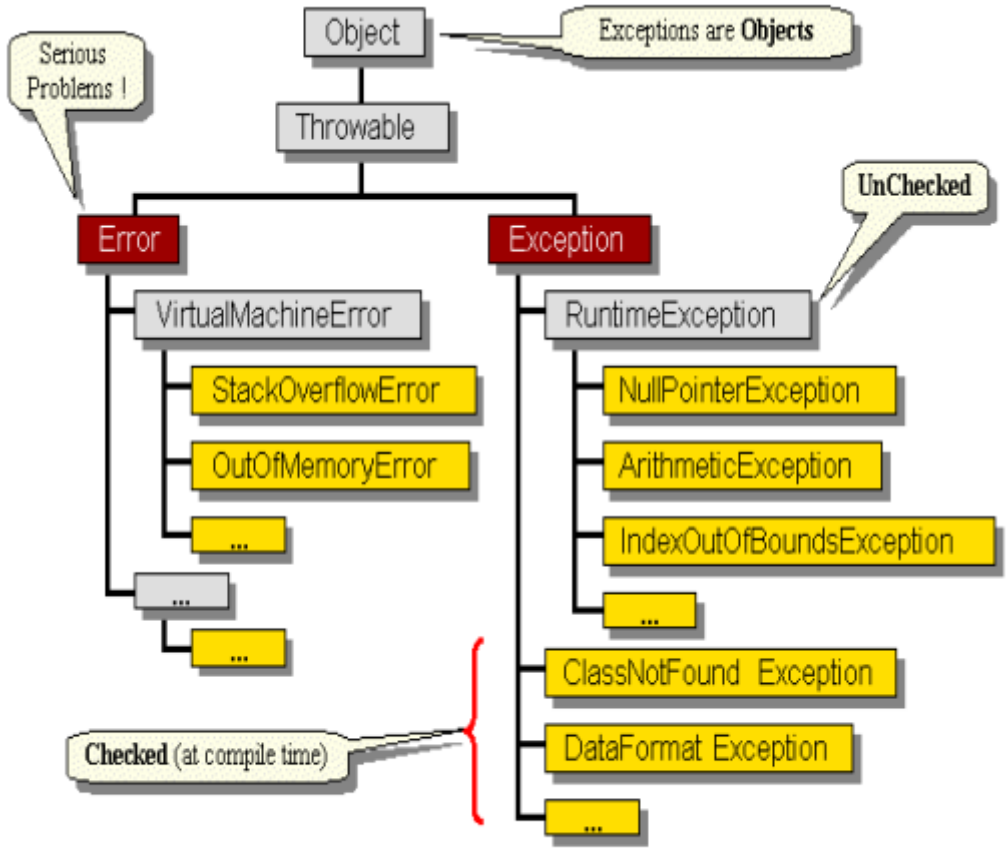
抛出的异常将在某个地方被捕获并处理

```java
public void getInformation() throws MyException(){
    //...
}

private void doStuff(){
    try{
        getInformation();
    }catch(MyException ex){
        // 在这里处理
    }
}
```



4. 异常 Exceptions vs 错误 Errors



- 异常是程序或 java 中发生的异常情况，告诉您发生了错误，当异常发生时，java 强制我们处理它
- 异常处理 exception handling 提供了一种处理错误和从错误中恢复的方法，或者一种优雅地退出程序的方法
- 错误 error 表示严重表示严重的不可恢复问题，例如 VirtualMachineError、OutOfMemoryError

5. 自定义异常

- 不需要从头创建自己的异常，而是扩展Java Throwable，因此必须决定要扩展什么类
  - 继承 **Exception** 类，那么它将是一个 **checked exception**，javac 将确保它是根据规则使用的，必须被抛出或者被捕获处理
  - 继承 **RuntimeException** 类，然后 javac 将不会强制您的异常的用户遵循在方法名中声明它并使用 throw 或 try/catch 的要求
    - **unchecked Exception 是 RuntimeException 的子类**

```
class MyException extends RuntimeException{
    //...
}
```

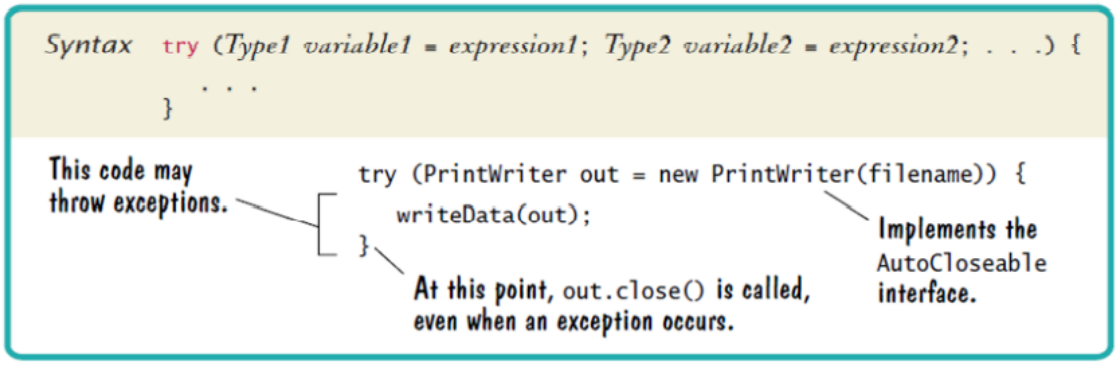
6. 关闭资源

当使用一个必须关闭的资源时，例如 PrintWriter，需要小心出现异常

```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close(); // 可能不会执行这一行语句
```

Try-with-resources 语句

在 try 语句中声明 Printwriter 变量，如下所示




```
try(PrinterWriter out = new PrinterWriter(filename)){ // 实现 AutoCloseable 接口
    writeData(out);
}// 当 try 语句完成后 out.close() 方法一直会被调用
```

```
try(Scanner in = new Scanner(inFile); PrintWriter out = new PrinterOut(outFile)){
    while(in.hasNextLine()){
        String input = in.nextLine();
        String result = process(input);
        out.println(result);
    }
}// in.close() 和 out.close() 都会在这里被调用
```

- 更一般地，您可以在 try-with-resources 语句中声明实现 **AutoCloseable** 接口的任何类的变量
- 在**关闭资源时**，应该始终使用 **try-with-resources** 语句

Try / finally 语句

除了调用 close 方法之外，还可能需要进行一些清理，在这种情况下，使用 try-finally 语句

```
public double deposit(double amount){
    try{
        //...
    }
    finally{
        // Cleanup 不论是否有异常发生，这段代码都会执行
    }
}
```

- 很少需要 try-finally 语句，因为大多数需要清理的 Java 库类都实现了 AutoCloseable 接口