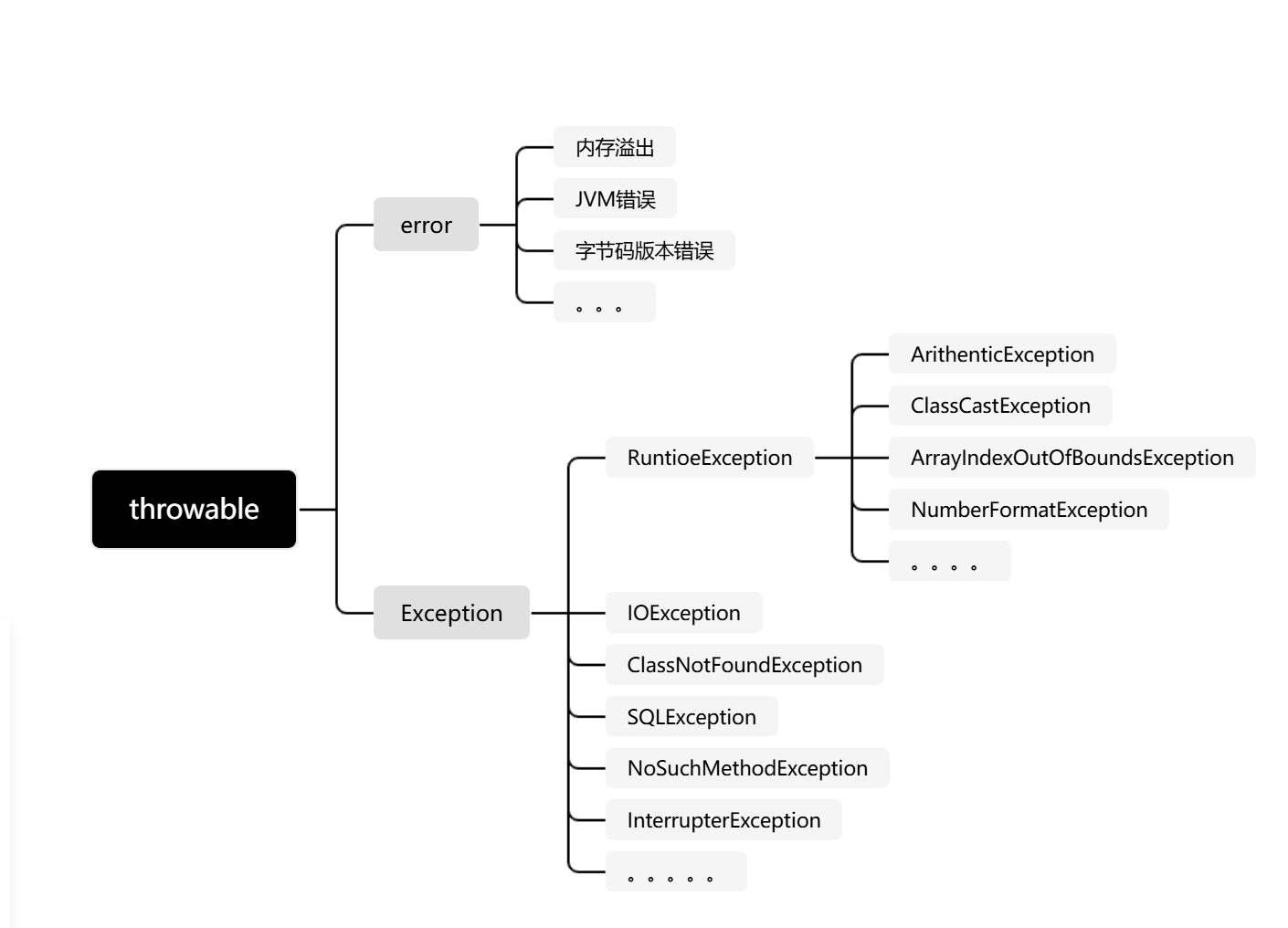
学号：2208300002、姓名：张煜

异常

# 异常体系问答

1. java中Throwable下的主要⼦类有（包括间接继承的

⼦类），请利⽤思维导图的⽅式画出异常体系



1. 简述Error和Exception区别

答：

\* `Error`，表 Error，表示错误情况，一般是程序中出现了比较严重的问题，并且程序自身并无法进行处理。

\* `Exception`，表示异常情况，程序中出了这种异常，大多是可以通过特定的方式进行处理和纠正的，并且处理完了之后，程序还可以继续往下正常运行。

3）简述运⾏时异常和编译时异常的别

答：

编译时异常

\* 继承自 `Exception` 类，也称为checked exception

\* 编译器在编译期间，会主动检查这种异常，如果发现异常则必须显示处理，否则程序就会发生错误，无法通过编译,无法生成字节码文件

运行时异常

\* `RuntimeException` 类及其子类，也称为unchecked exception

\* 编译器在编译期间，不会检查这种异常，也不要求我们去处理，但是在运行期间，如果出现这种异常则自动抛出

4）写出5个常⻅的运⾏时异常（中⽂+英⽂）

答：

1. **NullPointerException (空指针异常)**
2. **ArrayIndexOutOfBoundsException (数组索引越界异常)**
3. **ArithmeticException (算术异常)**
4. **ClassCastException (类转换异常)**
5. **IllegalArgumentException (非法参数异常)**

# 异常处理机制问答

1. 简述使⽤catch(Exception e)的好处是

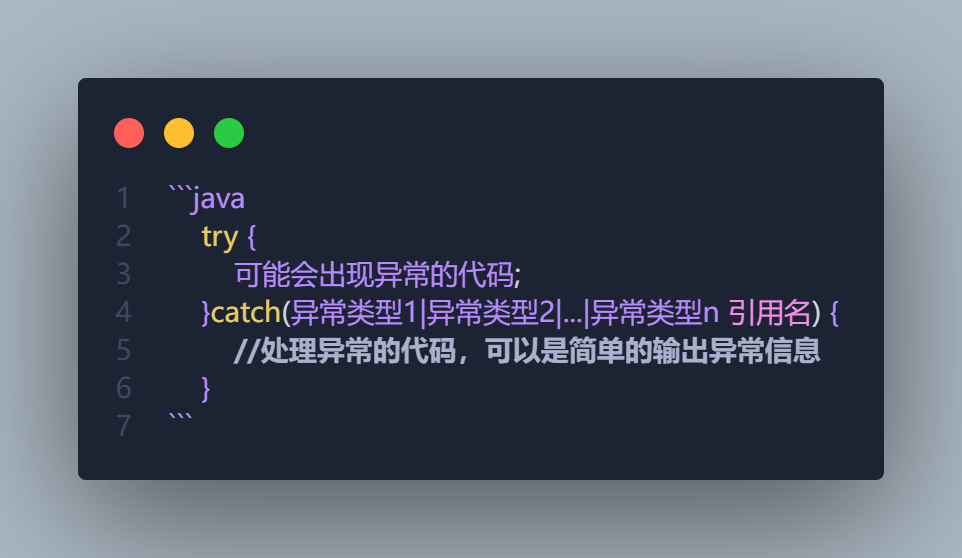
答：

1. **错误处理**：通过捕获异常，可以对程序中的错误进行处理，而不是让程序直接崩溃。这可以提高应用程序的健壮性和用户体验。
2. **控制流程**：异常捕获允许您控制程序的执行流程。当异常被捕获后，可以决定是继续执行后续代码还是采取补救措施。
3. **用户反馈**：通过捕获异常并提供用户友好的错误信息，可以让最终用户了解发生了什么问题以及他们应该如何应对，打印或者显示口语化的提示而不是看到晦涩的技术错误堆栈。
4. **调试与日志记录**：捕获异常可以提供调试的机会，并且可以在捕获异常的地方记录详细的错误日志，这对于追踪问题根源非常有用。
5. 简述当出现多个异常时，try-catch中catch⼦句的排列

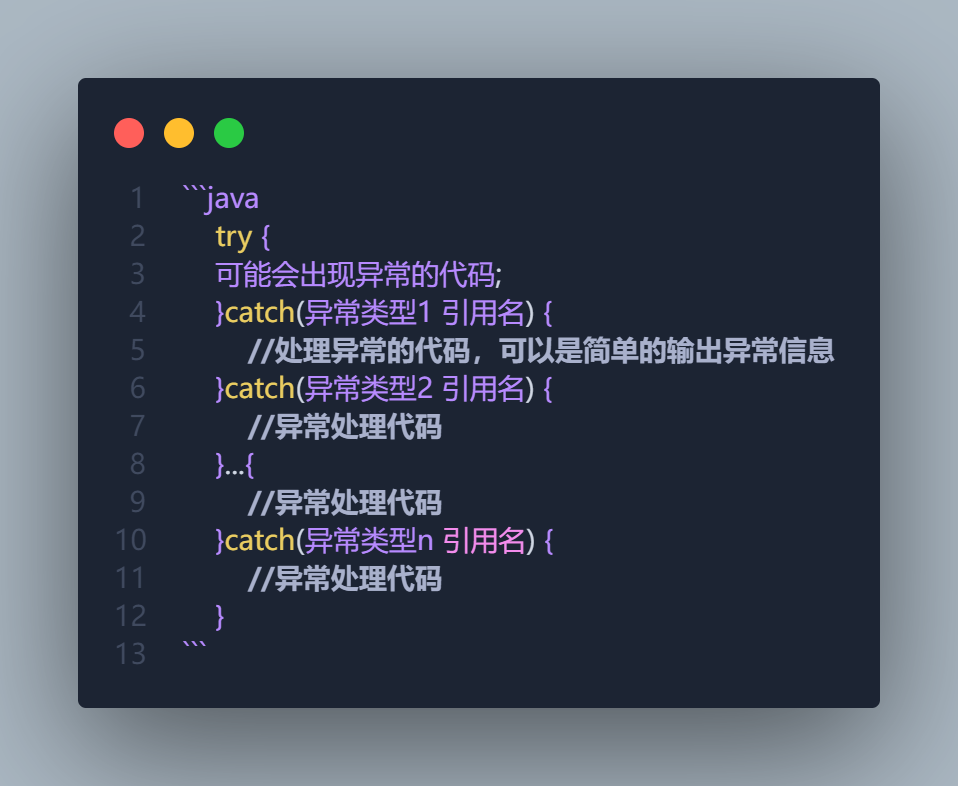
⽅式

答：

方法1：



方法2：



1. 简述throw 和 throws 的区别

【提示】

可从以下⼏个⽅⾯思考： 声明位置

抛出异常的个数

关键字作⽤

答：

**声明位置**

**throw** 关键字通常出现在方法体内部，即在方法的具体实现代码块中使用。它用于显式地抛出一个异常对象。

**throws** 关键字则通常出现在方法签名的后面，用来声明一个方法可能会抛出的异常类型。它告诉调用者该方法可能会抛出指定的异常，需要调用者进行处理。

**抛出异常的个数**

**throw** 一次只能抛出一个异常对象。如果您在一个方法体内遇到无法处理的异常情况，您会创建一个新的异常对象并通过 throw 语句将其抛出。

**throws** 可以声明一个方法可能会抛出多个异常类型。这些异常类型用逗号分隔，表明调用这个方法时需要处理这些异常之一或全部。

**关键字作用**

**throw** 的作用是抛出一个具体的异常实例。这意味着您已经检测到了某种错误条件，并决定将这种错误作为异常传递给上层处理。throw 后面跟着的是一个异常对象，通常是 new 出来的某个异常类的实例。

**throws** 的作用则是声明方法可能会抛出哪些异常类型。这并不意味着一定会抛出异常，而是说如果在方法内部发生了某些异常情况，这些异常可能会被抛出，而不是在当前方法中处理。使用 throws 可以帮助进行异常的向上委托，即将异常责任交给调用者处理。

# 异常编程

编写Test3\_DivisionByZero类

【要求】

1. 编写⽅法：division()：要求执⾏10/0操作，并使⽤异 常处理机制处理可能会产⽣的异常
2. 编写main()：调⽤division()
3. 修改division()：执⾏10/0不变，但不在⽅法中处理产

⽣的异常，改将异常抛出

1. 修改main()：调⽤division()并处理其抛出的异常

答：

public class Test\_DivisionByZero {

    public static void main(String[] args) {

        try {

            division();

        } catch (ArithmeticException e) {

            System.out.println("除数不能为0");

        }

    }

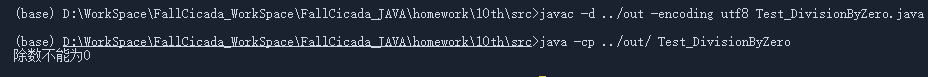
    public static void division() throws ArithmeticException {

        int c = 10 / 0;

    }

}

运行结果：



# finally问答

finally代码块是否⼀定会执⾏

答：

finally 代的块通常用于在异常处埋机制中执行一些清理工作，无论是否发生异常，finally 代码块中的代码都会执行。以下是一些关键点。

1.正常执行路径:如果没有发生异常，finally 代码块会在 try 代码块执行完之后执行。

2.异常执行路径:如果发生了异常，finally 代码块会在 catch 代码块执行完之后执行。

3.即使有 return 语句:即使在 try 或 catch 代码块中有 return 语句，finally 代码块仍然会执行。

特殊情况

有一些特殊情况可能会导致 fina11y 代码块不执行:

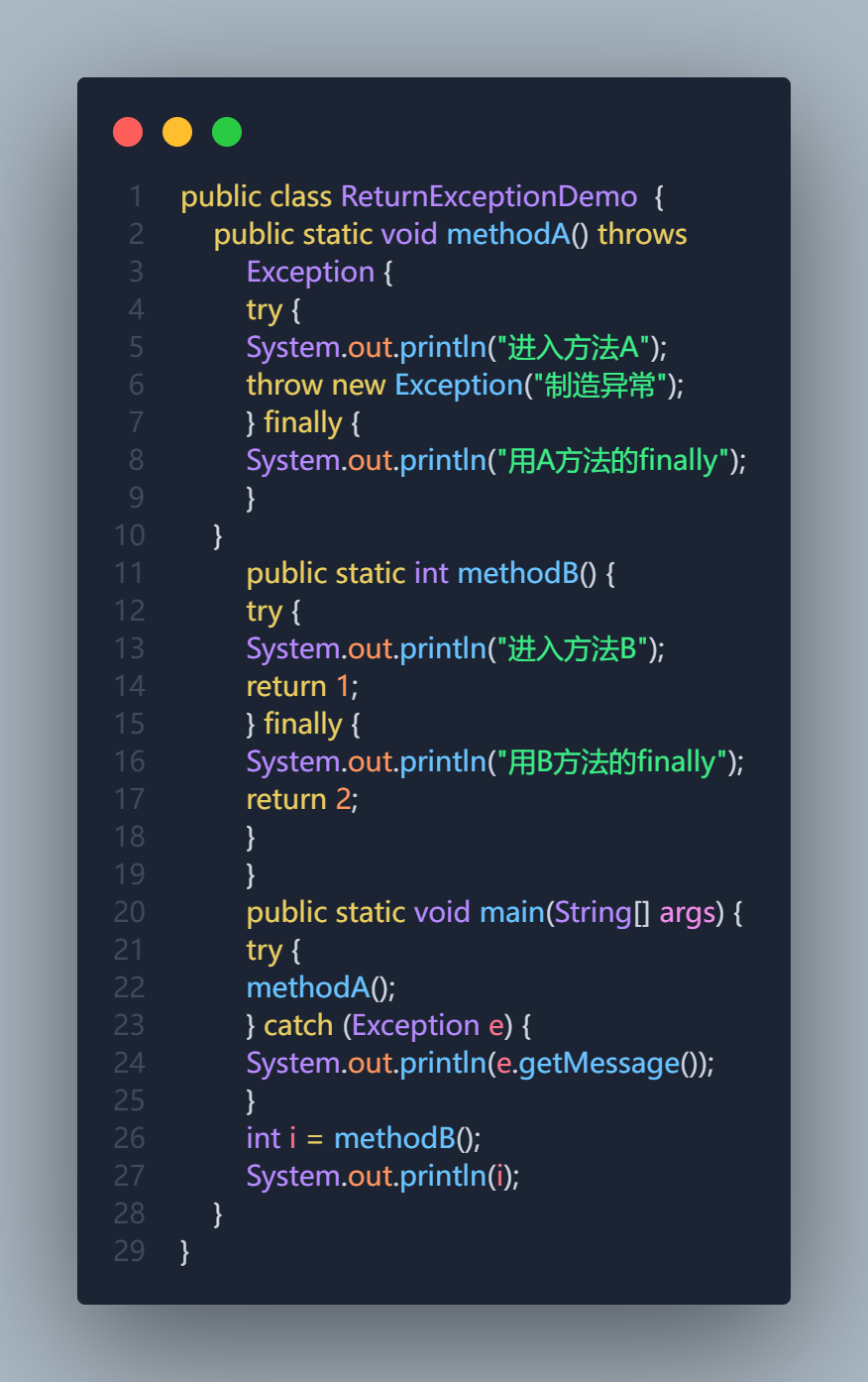
JVM 退出:如果在 try 或 catch 代码块中调用了 5ystem.exit()方法，JM 会立即终止，finally 代码块不会执行。

线程中断:如果线程在执行 try 或 catch 代码块时被中断，finally 代码块可能不会执行。

# finally代码分析

观察并分析以下代码，说出其输出结果

1）



答:

`methodA()`方法：

1. 打印 "进入方法A"。

2. 抛出一个新的 `Exception` 对象，并附带消息 "制造异常"。

3. 因为有 `throw` 语句，所以不会执行到后面的代码。

4. `finally` 块总是会被执行，因此会打印 "用A方法的finally"。

`methodB()`方法：

1. 打印 "进入方法B"。

2. 返回值 1。

3. 即使已经有了返回值，在 `finally` 块中的 `return 2;` 会导致 `methodB()` 方法最终返回 2，并且还会执行 `finally` 块中的其他代码，因此会打印 "用B方法的finally"。

最后看 `main` 方法：

1. 调用 `methodA()` 并捕获它可能抛出的 `Exception`。

2. 捕获到异常后，打印异常的消息，即 "制造异常"。

3. 调用 `methodB()` 并获取返回值，然后打印这个值，应该是 2。

综合以上分析，预期的输出结果如下：

进入方法A

用A方法的finally

制造异常

进入方法B

用B方法的finally

2

2）



答：

`testReturn1()` 方法：

1. 定义局部变量 `i` 并初始化为 `1`。

2. 在 `try` 块中，`i` 自增变为 `2`，然后打印 "try:2"。

3. 接着在 `try` 块中返回 `i`，此时 `i` 的值为 `2`。

4. 在 `finally` 块中，`i` 再次自增变为 `3`，然后打印 "finally:3"。

重要的一点是，在 Java 中，即使在 `finally` 块中有返回值，也不会影响 `try` 块中 `return` 语句的效果。`finally` 块中的 `return` 表达式会覆盖 `try` 块中的返回值，但是这里的 `finally` 块并没有返回语句。

因此，`testReturn1()` 方法返回的值仍然是 `2`，因为在 `try` 块中的 `return` 已经确定了返回值。

结合 `main` 方法中的调用：

1. 调用 `testReturn1()` 方法，打印 "try:2"，返回值为 `2`。

2. `finally` 块被执行，打印 "finally:3"。

3. `main` 方法中打印变量 `i` 的值，即 `2`。

因此，预期的输出结果如下：

try:2

finally:3

2

3）



答：

`testTest03()`方法：

1. 创建一个新的`ArrayList<Integer>`实例`list`。

2. 在`try`块中，向`list`添加整数`1`，得到`[1]`，然后打印"try:[1]"。

3. 在 `try` 块中返回 `list`。

4. 在 `finally` 块中，向 `list` 添加整数 `3`，得到 `[1, 3]`，然后打印 "finally:[1, 3]"。

由于 `finally` 块中的操作不会改变 `testTest03()` 方法返回的引用，所以在 `main` 方法中调用 `testTest03()` 后，打印的将是 `finally` 块中修改后的列表 `[1, 3]`。

结合 `main` 方法中的调用：

1. 调用 `testTest03()` 方法，打印 "try:[1]"。

2. `finally` 块被执行，打印 "finally:[1, 3]"。

3. `main` 方法中打印 `testTest03()` 返回的列表，即 `[1, 3]`。

因此，预期的输出结果如下：

try:[1]

finally:[1, 3]

[1, 3]

3）



答：

根据提供的代码，我们可以逐步分析每一部分的行为，并预测其输出结果。

`testTest04()` 方法的行为如下：

1. 初始化字符串变量 `a` 为 `"hello"`。

2. 在 `try` 块中，将 `a` 修改为 `"hello!"`，然后打印 "try:hello!"。

3. 在 `try` 块中返回 `a`，此时 `a` 的值为 `"hello!"`。

4. 在 `finally` 块中，将 `a` 修改为 `"hello!world"`，然后打印 "finally:hello!world"。

需要注意的是，虽然 `finally` 块中的代码会修改 `a` 的值，但是 `try` 块中已经决定了返回值。在 Java 中，`return` 语句实际上会保存返回值到一个临时变量中，而 `finally` 块中的代码修改不会影响这个临时变量。因此，`testTest04()` 方法返回的值仍然是 `"hello!"`。

结合 `main` 方法中的调用：

1. 调用 `testTest04()` 方法，打印 "try:hello!"。

2. `finally` 块被执行，打印 "finally:hello!world"。

3. `main` 方法中打印 `testTest04()` 方法返回的字符串，即 `"hello!"`。

因此，预期的输出结果如下：

try:hello!

finally:hello!world

hello!

这就是代码执行时按顺序输出的结果。