异常处理

1、异常：指的是程序在执行过程中，出现的**非正常情况**，如果不处理最终会**导致 JVM 的非正常停止**。如：输入数据的格式问题，读取文件是否存在，网络是否始终保持通畅等。

【注】：异常指的并**不是语法错误和逻辑错误**。**语法错**了，编译不通过，不会产生字节码文件，根本不能运行。代码**逻辑错误**，只是没有得到想要的结果，例如：求 a+b，写成了 a-b。

2、异常的抛出机制

Java 中把**不同的异常**用**不同的类**表示，一旦**发生**某种异常，就**创建**该异常类型的**对象**，并且**抛出（throw）**。然后程序员可以**捕获(catch)**到这个异常对象，**并处理**；如果**没有捕获**(catch)这个异常对象，那么这个异常对象将会**导致程序终止**。

3、如何对待异常

对于程序出现的异常，一般有**两种**解决方法：一是遇到错误就**终止程序的运行**。

另一种方法是程序员在编写程序时，就充分**考虑到各种可能发生的异常和错误**，极力预防和避免。实在无法避免的，要**编写相应的代码**进行**异常的检测**、以及**异常的处理**，保证代码的健壮性。

4、java.lang.Throwable（异常体系根父类）

Throwable 中的常用方法：

1）public void **printStackTrace()**：打印异常的详细信息。包含了异常的**类型**、异常的**原因**、异常出现的**位置**、在开发和调试阶段都得使用 printStackTrace()。

2）public String **getMessage()**：获取发生**异常的原因**。

5、Throwable 可分为两类：Error 和 Exception。分别为 java.lang.Error 与 java.lang.Exception 两个类。

1）Error：Java 虚拟机**无法解决的严重问题**。如：**JVM 系统内部错误**、**资源耗尽**等严重情况。一般不编写针对性的代码进行处理。

例如：StackOverflowError（**栈内存溢出**）和 OutOfMemoryError（**堆内存溢出**，简称 OOM）。

2）Exception：其它因**编程错误**或**偶然的外在因素**导致的一般性问题，需要**使用针对性的代码进行处理**，使程序继续运行。否则一旦发生异常，程序也会挂掉。

6、Java 程序的执行分为编译时过程和运行时过程。根据异常可能出现的阶段，可以将异常分为：

1）编译时期异常（即 checked 异常、受检异常）：在代码编译阶段，**编译器**就能明确**警示**当前代码**可能发生（不是一定发生）xx 异常**，并明确督促程序员提前编写处理它的代码。如果程序员没有编写对应的异常处理代码，则编译器就会直接判定编译失败，从而不能生成字节码文件。通常，这类异常的发生不是由程序员的代码引起的，或者不是靠加简单判断就可以避免的，例如：（文件找不到异常）FileNotFoundException。

2）运行时期异常（即 runtime 异常、unchecked 异常、非受检异常）：在代码编译阶段，编译器完全不做任何检查，**无论该异常是否会发生，编译器都不给出任何提示**。只有等**代码运行起来**并确实发生了 xx 异常，它**才能被发现**。通常，这类异常是由程序员的代码编写不当引起的，只要稍加判断，或者细心检查就可以避免。java.lang.RuntimeException:类及它的子类都是运行时异常。比如：数组下标越界异常 ArrayIndexOutOfBoundsException:，类型转换异常 ClassCastException。

【注意】RuntimeException 类或其子类的异常的特点：即使没有使用 try 和 catch 捕获，Java 自己也能捕获，并且编译通过（但运行时会发生异常使得程序运行终止）。所以，对于这类异常，可以不作处理，因为这类异常很普遍，若全处理可能会对程序的可读性和运行效率产生影响。如果抛出的异常是 **IOException 等类型的非运行时异常**，则**必须捕获，否则编译错误**。即，我们**必须处理编译时异常，将异常进行捕捉，转化为运行时异常**。

7、常见的错误和异常

1）Error

最常见的就是 **VirtualMachineError**，俩经典的子类：**栈内存溢出** StackOverflowError、**堆内存溢出**(OOM)OutOfMemoryError。

2）运行时异常

```
1>//ArrayIndexOutOfBoundsException 越界
int[] arr = new int[5];
System.out.println(arr[5]);
2>//NullPointerException 空指针
int[][] arr = new int[3][];
System.out.println(arr[0].length);
3>//ClassCastException 强制类型转换
Object obj = 15;
String str = (String) obj;(向下转型)
4>//NumberFormatException 数字格式转换
String str = “123”;
Int i = Integer.parseInt(str);
System.out.println(i)
5>//InputMismatchException 输入类型不匹配
Scanner input = new Scanner(System.in);
System.out.print("输入整数：");//输入非整数
int num = input.nextInt();
input.close();//资源关闭
6> //ArithmeticException 算术
int a = 1; int b = 0;
System.out.println(a/b);
```

3）编译时异常

```
1>// InterruptedException
Thread.sleep(1000); //休眠 1 秒
2>//ClassNotFoundException
Class c = Class.forName("java.lang.String");
3>//SQLException
Connection conn = DriverManager.getConnection("....");
4>//FileNotFoundException、IOException
File file = new File("尚硅谷 Java 秘籍.txt");
//FileNotFoundException
FileInputStream fis = new FileInputStream(file);
int b = fis.read();//可能堵塞 IOException
while(b != -1){
        System.out.print((char)b);
        b = fis.read();//可能堵塞 IOException
        }
fis.close();//可能堵塞 IOException
```

8、异常的处理

在编写程序时，经常要在可能出现错误的地方加上检测的代码，如进行 x/y 运算时，要检测分母为 0，数据为空，输入的不是数据而是字符等。**过多的 if-else 分支会导致程序的代码加长、臃肿，可读性差**，程序员需要花很大的精力“堵漏洞”。**因此采用异常处理机制**。

Java 异常处理：采用的异常处理机制，是将异常处理的程序代码集中在一起，与正常的程序代码分开，使得程序简洁、优雅，并易于维护。**Java 异常处理的方式（两种）**

9、捕获异常(try-catch-finally):

1）Java 提供了**异常处理的抓抛模型**：

1>**抛出**(throw)异常过程：Java 程序的**执行过程**中如**出现异常**，会**生成一个异常类对象**，该异常对象将被提交**给 Java 运行时系统**。

2>**捕获**(catch)异常过程：如果一个方法内抛出异常，**该异常对象会被抛给调用者方法中**处理。如果异常没有在调用者方法中处理，它**继续被抛给**这个调用方法的**上层方法**。这个过程将一直继续下去，**直到异常被处理**。

3>如果一个异常**回到 main()方法**，并且 **main()也不处理**，则程序**运行终止**。

2）整体执行过程：

当某段代码可能发生异常，**不管这个异常是编译时异常（受检异常）还是运行时异常（非受检异常）**，我们都可以使用 try 块将它括起来，并在 try 块下面编写 catch 分支尝试捕获对应的异常对象。

1>如果在程序运行时，**try 块中**的代码**没有发生异常**，那么 **catch** 所有的分支**都不执行**。

2>如果在程序运行时，**try 块中**的代码**发生了异常**，根据**异常对象的类型**，将***从上到下选择第一个匹配的 catch 分支执行***。此时 **try 中发生异常的语句下面的代码将不执行**，而**整个 try...catch 之后的代码可以继续运行**。

3>如果在程序运行时，**try 块中**的代码**发生了异常**，但是***所有 catch 分支都无法匹配***（捕获）这个异常，那么 **JVM 将会终止当前方法的执行**，并把异常对象**“抛”给调用者**。如果**调用者不处理，程序就挂了**。

3）try：捕获异常的第一步是用 **try{…}语句块选定捕获异常的范围**，将**可能出现异常的业务逻辑代码**放在 try 语句块中。

4）catch (Exceptiontype e)

1>catch 分支，分为两个部分，**catch()中**编写**异常类型**和**异常参数名**，**{}中**编写如果发生了这个异常，**要做什么处理的代码**。

2>如果明确知道产生的是何种异常，可以用**该异常类**作为 catch 的参数；也可以用**其父类**作为 catch 的参数。

比如：可以用 ArithmeticException 类作为参数的地方，就可以用 RuntimeException 类作为参数，或者用所有异常的父类 Exception 类作为参数。但不能是与 ArithmeticException 类无关的异常，如 NullPointerException（catch 中的语句将不会执行）。

3>每个 try 语句块**可以伴随一个或多个 catch 语句**，处理可能产生的不同类型的异常对象。

4>如果有多个 catch 分支，并且**多个异常类型有父子类关系，必须**保证**小的子**异常类型**在上，大的父**异常类型**在下**。否则，报错。

5>catch 中常用异常处理的方式

public String **getMessage()**：获取异常的**描述信息**，返回字符串；

public void **printStackTrace()**：打印异常的**跟踪栈信息**并输出到控制台。包含了异常的类型、异常的原因、还包括异常出现的位置，在开发和调试阶段，都得使用 printStackTrace()。

5）finally(放一定要被执行的代码)

1>因为异常会引发程序跳转，从而会导致有些语句执行不到。而程序中**有一些特定的代码无论异常是否发生，都需要执行。**例如，**数据库连接、输入流输出流、Socket 连接、Lock 锁的关闭**等，这样的代码通常就会放到 finally 块中。

【注意】唯一例外，使用 **System.exit(0)**来**终止当前**正在运行的 **Java 虚拟机**时，finally 中的代码不会被执行。

2>不论在 try 代码块中是否发生了异常事件，catch 语句是否执行，catch 语句是否有异常，**catch 语句中是否有 return，finally 块中的语句都会被执行。**

【注意】

> finally 是在 **return 语句*执行之后，返回之前*执行的**（此时并没有返回运算后的值，而是先把要**返回的值保存在临时栈**，不管 finally 中的代码怎么样，返回的值都不会改变，仍然是之前保存的值），所以**如果 finally 中没有 return**，即使**对数据有操作也不会影响返回值**，即如果 finally 中没有 return，**函数返回值是在 finally 执行前就已经确定了**；

> **finally 中如果包含 return，那么程序将在这里返回，返回值就不是 try 或 catch 中保存的返回值了**。

3>finally 语句和 catch 语句是可选的，但 **finally 不能单独使用**。

10、声明抛出异常类型(throws +异常类型)

如果在编写方法体的代码时，某句代码可能发生某个编译时异常，不处理编译不通过，但是**在当前方法体中可能不适合处理或无法给出合理的处理方式**，则此方法应**显示地声明抛出异常**，表明该方法将不对这些异常进行处理，而**由该方法的调用者负责处理**。

1）具体方式：在方法声明中**用 throws 语句**可以**声明抛出异常的列表**，throws 后面的异常类型可以是方法中**产生的异常类型**，也可以是**它的父类**。

```
public class TestThrowsCheckedException {
    public static void main(String[] args) {
        System.out.println("上课.....");
        try {
            afterClass();//换到这里处理异常
        } catch (InterruptedException e) {
            e.printStackTrace();
            System.out.println("准备提前上课");
        }
        System.out.println("上课.....");
    }

    public static void afterClass() throws InterruptedException {
        for(int i=10; i>=1; i--){
            Thread.sleep(1000);//本来应该在这里处理异常
            System.out.println("距离上课还有: " + i + "分钟");
        }
    }
}
```

throws 后面也可以写运行时异常类型，只是写或不写**对于编译器和程序执行来说都没有任何区别**。**唯一的区别**就是**调用者调用该方法后**，使用 try...catch 结构时，**IDEA 可以获得更多的信息，需要添加哪种 catch 分支**。

2)方法重写中对于 throws 异常列表的要求：

1> 如果**父类被重写方法**的方法签名**后面没有 “throws 编译时异常类型”**，那么**重写方法时**，方法签名**后面也不能出现“throws 编译时异常类型”**。（父类不抛，子类也不能抛）

2> 如果**父类被重写方法**的方法签名**后面有“throws 编译时异常类型”**，那么**重写方法时，throws 的编译时异常类型必须 <= 被重写方法 throws 的编译时异常类型**，或者**不 throws 编译时异常**。（父类抛了，子类抛小于父范围）

3>【注意】方法重写，对于“throws **运行时异常类型”没有要求。**

11、两种异常处理方式的选择

前提：对于异常，使用**相应的处理方式**。此时的异常，**主要指的是编译时异常**。

1）如果程序代码中，**涉及到资源的调用（流、数据库连接、网络连接等）**，则**必须**考虑使用 **try-catch-finally 处理**，**保证不出现内存泄漏**。

2）如果**父类被重写的方法没有 throws 异常类型**，则**子类重写的方法中出现异常，只能使用 try-catch-finally 进行处理**，不能 throws。

3）开发中，方法 a 中依次调用了方法 b,c,d 等方法，方法 b,c,d 之间是递进关系。此时，**如果方法 b,c,d 中有异常，我们通常使用 throws**，而**方法 a 中通常选择使用 try-catch-finally**。

12、手动抛出异常对象(throw new)

1）在实际开发中，如果出现不满足具体场景的的代码问题（如学号不能为负数），我们就有必要手动抛出一个指定类型的异常对象。

即：在方法内部，**满足指定条件的情况下使用“throw new 异常类型(参数);”**的方式抛出。

【注】throw 后代码不能被执行，编译不通过。

2）理解“自动/手动”抛出异常对象

自动抛：程序执行过程中，一旦出现异常，就会在出现异常的代码处，**自动生成对应异常类的对象**，并将此对象**抛出**；

手动抛：程序执行过程中，不满足指定条件的情况下，我们**主动使用“throw+异常类对象”**。

3）面试题：throw 和 throws 的区别：

1>throws 出现在**方法函数头**，后面跟**异常类型**；而 throw 出现在**函数体**，后面跟**异常对象**。

2>throws 表示出现异常的一种可能性，**并不一定**会发生这些异常；throw 则是**抛出了**异常，执行 throw 则**一定抛出了某种异常对象**。

3>使用场景不同：throws 是针对**已产生的对象抛给函数的上层**调用去处理；throw 是**产生异常对象**。二者一般是**合作关系**，**产生**一个对象，然后**交给上一层**进行处理。

13、自定义异常

1）为什么需要自定义异常类

Java 中不同的异常类，**分别表示着某一种具体的异常情况**。那么在开发中**总是有些异常情况是核心类库中没有定义好的**，此时我们就有必要在实际开发场景中**不满足我们制定条件时**，指明我们自己**特有的异常类**。**通过**此**异常类的名称**就能**判断出具体出现的问题**。例如年龄负数问题，考试成绩负数问题，某员工已在团队中等。

2）如何自定义异常类

1> 要**继承一个现有的异常类型**

>自定义一个编译时异常类型：自定义类继承 java.lang.Exception。

>自定义一个运行时异常类型：自定义类继承 java.lang.RuntimeException。

2> 建议大家提供至少两个**构造器**，一个是***无参构造***，一个是***(String message)构造器***。

3> 自定义异常需要**提供一个全局常量**，声明为 **static final long serialVersionUID**;（序列版本号是实现序列化接口对象的唯一标识，用来识别该类）

3）如何使用自定义异常类

1>在具体代码中，满足指定条件的情况下，只能**手动使用“throw+自定义异常类对象”**抛出自定义异常对象；

2>如果自定义异常类是非运行时异常，则必须考虑如何处理此异常类的对象。**抛出后由 try..catch 处理，也可以 throws 给调用者处理**。

3>自定义异常最重要的是**异常类的名字**和**message 属性**。当异常出现时，可以根据名字判断异常类型。比如：TeamException("成员已满,无法添加");TeamException("该员工已是某团队成员");

```
public class ReturnExceptionDemo {
    static void methodA() {
        try {
            System.out.println("进入方法 A");
            throw new RuntimeException("制造异常");
        }finally {
            System.out.println("用 A 方法的 finally");
        }
    }
    static void methodB() {
        try {
            System.out.println("进入方法 B");
            return;
        } finally {
            System.out.println("调用 B 方法的 finally");
        }
    }
    public static void main(String[] args) {
        try {
            methodA();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
        methodB();
    }
}
```