

1. 纲要

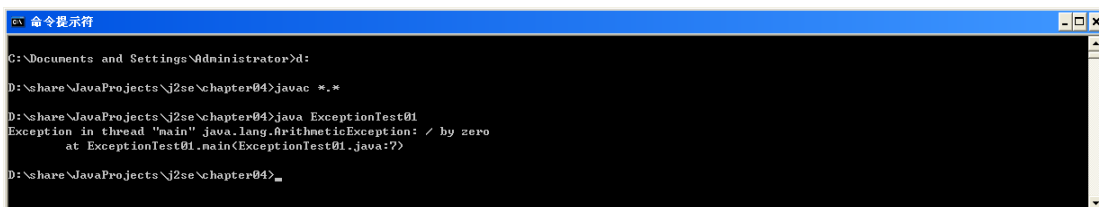
1. 异常的基本概念
2. 异常的分类
3. 异常的捕获和处理
4. 自定义异常
5. 方法覆盖与异常

2、 内容

2.1、异常的基本概念

什么是异常，在程序运行过程中出现的错误，称为异常

```
public class ExceptionTest01 {  
  
    public static void main(String[] args) {  
        int i1 = 100;  
        int i2 = 0;  
  
        int i3 = i1/i2;  
  
        System.out.println(i3);  
    }  
}
```



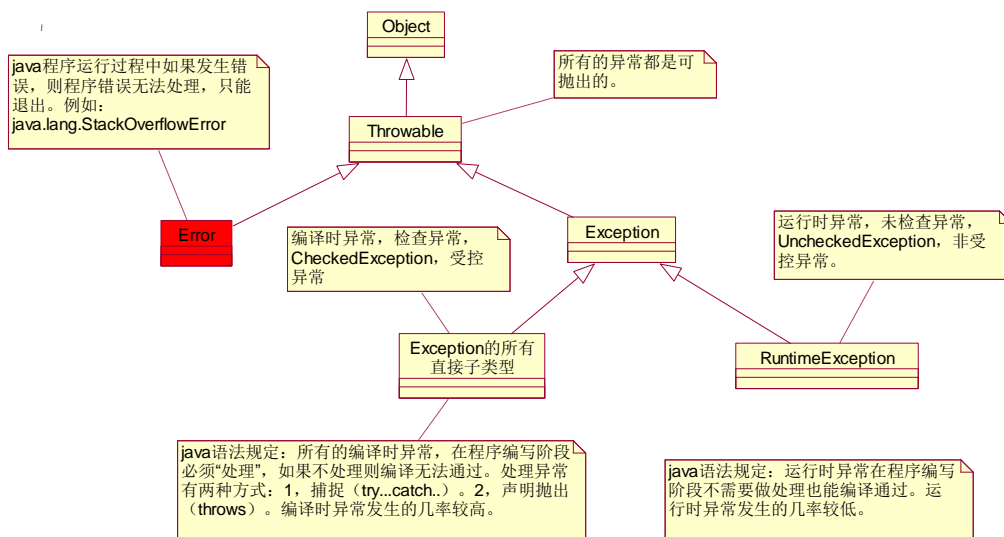
```
C:\Documents and Settings\Administrator>d:  
D:\share\JavaProjects\j2se\chapter04>javac *.java  
D:\share\JavaProjects\j2se\chapter04>java ExceptionTest01  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at ExceptionTest01.main(ExceptionTest01.java:?)  
D:\share\JavaProjects\j2se\chapter04>
```

没有正确输出，抛出了被0除异常

通过以上示例，我们看到 java 给我们提供了这样一个体系结构，当出现问题的时候，它会告诉我们，并且把错误的详细信息也告诉了我们了，这就是异常的体系结构，这样我们的程序更健壮，我们可以把这个信息，再进行处理以下告诉用户。从上面大家还可以看到，java 异常都是类，在异常类中会携带一些信息给我们，我们可以通过这个类把信息取出来

2.2、异常的分类

2.2.1、异常的层次结构



2.2.2、异常的分类

异常主要分为：错误、一般性异常（受控异常）、运行期异常（非受控异常）

- **错误**：如果应用程序出现了 **Error**，那么将无法恢复，只能重新启动应用程序，最典型的 **Error** 的异常是：OutOfMemoryError
- **受控异常**：出现了这种异常必须显示的处理，不显示处理 java 程序将无法编译通过
- **非受控异常**：此种异常可以不用显示的处理，例如被 0 除异常，java 没有要求我们一定要处理

2.3.1、try、catch 和 finally

异常的捕获和处理需要采用 try 和 catch 来处理，具体格式如下：

```
try {  
  
  
}catch(OneException e) {  
  
  
}catch(TwoException e) {  
  
  
}finally {  
  
  
}  
}
```

- try 中包含了可能产生异常的代码
- try 后面是 catch，catch 可以有一个或多个，catch 中是需要捕获的异常
- 当 try 中的代码出现异常时，出现异常下面的代码不会执行，马上会跳转到相应的 catch 语句块中，如果没有异常不会跳转到 catch 中
- finally 表示，不管是出现异常，还是没有出现异常，finally 里的代码都执行，finally 和 catch 可以分开使用，但 finally 必须和 try 一块使用，如下格式使用 finally 也是正确的

```
try {  
  
  
}finally {  
  
  
}  
}
```

【示例代码】

```
public class ExceptionTest02 {  
  
    public static void main(String[] args) {  
        int i1 = 100;  
        int i2 = 0;  
        //try 里是出现异常的代码  
        //不出现异常的代码最好不要放到 try 作用  
        try {  
  
  
            //当出现被 0 除异常，程序流程会执行到 “catch(ArithmeticException
```

ae)” 语句

//被 0 除表达式以下的语句永远不会执行

```
int i3 = i1/i2;
```

//永远不会执行

```
System.out.println(i3);
```

//采用 catch 可以拦截异常

//ae 代表了一个 ArithmeticException 类型的局部变量

//采用 ae 主要是来接收 java 异常体系给我们 new 的 ArithmeticException 对象

//采用 ae 可以拿到更详细的异常信息

```
}catch(ArithmeticException ae) {  
    System.out.println("被 0 除了");  
}
```

```
}
```

```
}
```

2.3.2、getMessage 和 printStackTrace()

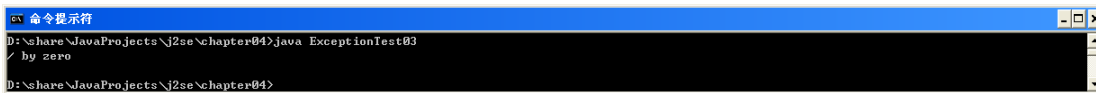
如何取得异常对象的具体信息，常用的方法主要有两种：

- 取得异常描述信息：getMessage()
- 取得异常的堆栈信息(比较适合于程序调试阶段)：printStackTrace();

【代码示例】

```
public class ExceptionTest03 {  
  
    public static void main(String[] args) {  
        int i1 = 100;  
        int i2 = 0;  
        try {  
            int i3 = i1/i2;
```

```
        System.out.println(i3);
    } catch (ArithmeticException ae) {
        //ae 是一个引用，它指向了堆中的 ArithmeticException
        //通过 getMessage 可以得到异常的描述信息
        System.out.println(ae.getMessage());
    }
}
```



【代码示例】

```
public class ExceptionTest04 {

    public static void main(String[] args) {
        method1();
    }

    private static void method1() {
        method2();
    }

    private static void method2() {
        int i1 = 100;
        int i2 = 0;
        try {
            int i3 = i1/i2;
            System.out.println(i3);
        } catch (ArithmeticException ae) {
            //ae 是一个引用，它指向了堆中的 ArithmeticException
        }
    }
}
```

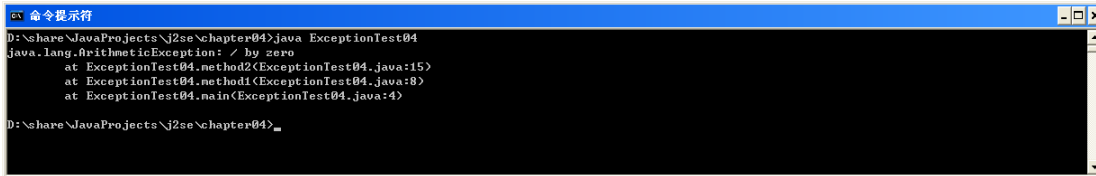
```
//通过 printStackTrace 可以打印栈结构
```

```
ae.printStackTrace();
```

```
}
```

```
}
```

```
}
```



2.3.3、受控异常

```
import java.io.FileInputStream;
```

```
public class ExceptionTest05 {
```

```
    public static void main(String[] args) {
```

```
        FileInputStream fis = new FileInputStream("test.txt");
```

```
    }
```

```
}
```



从上面输出可以看到，无法编译，它抛出了一个异常，这个异常叫做“受控异常”
FileNotFoundException，也就是说在调用的时候必须处理文件不能找到

处理 FileNotFoundException

```
/*
```

```
import java.io.FileInputStream;
```

```
import java.io.FileNotFoundException;
```

```
*/
```

```
import java.io.*;

public class ExceptionTest06 {

    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("test.txt");

        } catch (FileNotFoundException ffe) { //此异常为受控异常，必须处理
            ffe.printStackTrace();
        }
    }
}
```

2.3.4、finally 关键字

finally 在任何情况下都会执行，通常在 finally 里关闭资源

【示例代码】

```
import java.io.*;

public class ExceptionTest07 {

    public static void main(String[] args) {
        try {
            FileInputStream fis = new FileInputStream("test.txt");

            System.out.println("-----before fis.close-----");
            //close 是需要拦截 IOException 异常
            //在此位置关闭存在问题，当出现异常
        }
    }
}
```

```
//那么会执行到 catch 语句，以下 fis.close 永远不会执行
//这样个对象永远不会得到释放，所以必须提供一种机制
//当出现任何问题，都会释放相应的资源（恢复到最初状态）
//那么就要使用 finally 语句块
```

```
    fis.close();
    System.out.println("-----after fis.close-----");
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}
```

采用 finally 来释放资源

```
import java.io.*;

public class ExceptionTest08 {

    public static void main(String[] args) {

        //因为 fis 的作用域问题，必须放到 try 语句块外,局部变量必须给初始值
        //因为是对象赋值为 null
        FileInputStream fis = null;
        try {
            //FileInputStream fis = new FileInputStream("test.txt");
            fis = new FileInputStream("test.txt");

            /*
            System.out.println("-----before fis.close-----");
            fis.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



```
        System.out.println("-----after fis.close-----");
        */
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } finally {
        try {
            System.out.println("-----before fis.close-----");
            //放到 finally 中的语句，程序出现任何问题都会被执行
            //所以 finally 中一般放置一些需要及时释放的资源
            fis.close();
            System.out.println("-----after fis.close-----");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

深入 finally

【代码示例】

```
public class ExceptionTest09 {

    public static void main(String[] args) {
        int i1 = 100;
        int i2 = 10;
        try {
            int i3 = i1/i2;
            System.out.println(i3);
            return;
        } catch (ArithmeticException ae) {
            ae.printStackTrace();
        }
    }
}
```

```
    }finally {  
        //会执行 finally  
        System.out.println("-----finally-----");  
    }  
}  
}
```

深入 finally

```
public class ExceptionTest10 {  
  
    public static void main(String[] args) {  
        int i1 = 100;  
        int i2 = 10;  
        try {  
            int i3 = i1/i2;  
            System.out.println(i3);  
            //return;  
            System.exit(-1); //java 虚拟机退出  
        } catch (ArithmeticException ae) {  
            ae.printStackTrace();  
        } finally {  
  
            //只有 java 虚拟机退出不会执行 finally  
            //其他任何情况下都会执行 finally  
            System.out.println("-----finally-----");  
        }  
    }  
}
```

深入 finally

```
public class ExceptionTest11 {
```

```
public static void main(String[] args) {  
    int r = method1();  
    //输出为: 10  
    System.out.println(r);  
}  
  
/*  
private static int method1()  
    {  
        byte byte0 = 10;  
        byte byte3 = byte0; //将原始值进行了保存  
        byte byte1 = 100;  
        return byte3;  
        Exception exception;  
        exception;  
        byte byte2 = 100;  
        throw exception;  
    }  
*/  
  
private static int method1() {  
    int a = 10;  
    try {  
        return a;  
    } finally {  
        a = 100;  
    }  
}
```

```
}
```

深入 finally

```
public class ExceptionTest12 {

    public static void main(String[] args) {
        int r = method1();
        //输出为: 100
        System.out.println(r);
    }

    /*
    private static int method1()
    {
        byte byte0 = 10;
        byte0 = 50;
        byte0 = 100;
        break MISSING_BLOCK_LABEL_18;
        Exception exception;
        exception;
        byte0 = 100;
        throw exception;
        return byte0;
    }
    */
    private static int method1() {
        int a = 10;
        try {
            a = 50;
        } finally {
            a = 100;
        }
    }
}
```

```
    }  
    return a;  
}  
}
```

2.3.5、final、finalize 和 finally?

2.3.6、如何声明异常

在方法定义处采用 throws 声明异常，如果声明的异常为受控异常，那么调用方法必须处理此异常

【示例代码】，声明受控异常

```
import java.io.*;  
  
public class ExceptionTest13 {  
  
    public static void main(String[] args)  
        //throws FileNotFoundException, IOException { //可以在此声明异常,这样就交给 java 虚拟机处理了,不建议这样使用  
        throws Exception { //可以采用此种方式声明异常,因为 Exception 是两个异常的父类  
  
        /*  
        //分别处理各个异常  
        try {  
            readFile();  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
*/  
//可以采用如下方式处理异常  
//因为 Exception 是 FileNotFoundException 和 IOException 的父类  
//但是一般不建议采用如下方案处理异常，粒度太粗了，异常信息  
//不明确  
/*  
try {  
    readFile();  
} catch (Exception e) {  
    e.printStackTrace();  
}  
*/  
readFile();  
}  
  
private static void readFile()  
throws FileNotFoundException, IOException { //声明异常，声明后调用者必须  
处理  
    FileInputStream fis = null;  
    try {  
        fis = new FileInputStream("test.txt");  
    } catch (FileNotFoundException e) {  
        // e.printStackTrace();  
    } finally {  
        //try {  
            fis.close();  
        } catch (IOException e) {  
            // e.printStackTrace();  
        }  
    }  
}
```

```
}  
}
```

【示例代码】，声明非受控异常

```
public class ExceptionTest14 {  
  
    public static void main(String[] args) {  
        //不需要使用 try...catch.., 因为声明的是非受控异常  
        //method1();  
  
        //也可以拦截非受控异常  
        try {  
            method1();  
        } catch (ArithmeticException e) {  
            e.printStackTrace();  
        }  
    }  
  
    //可以声明非受控异常  
    private static void method1() throws ArithmeticException {  
        int i1 = 100;  
        int i2 = 0;  
        // try {  
            int i3 = i1/i2;  
            System.out.println(i3);  
        /*  
        } catch (ArithmeticException ae) {  
            ae.printStackTrace();  
        }  
        */  
    }  
}
```

```
}
```

2.3.7、如何手动抛出异常

```
public class ExceptionTest15 {

    public static void main(String[] args) {
        int ret = method1(1000, 10);
        if (ret == -1) {
            System.out.println("除数为 0");
        }
        if (ret == -2) {
            System.out.println("被除数必须为 1~100 之间的数据");
        }
        if (ret == 1) {
            System.out.println("正确");
        }
        //此种方式的异常处理，完全依赖于程序的返回
        //另外异常处理和程序逻辑混在一起,不好管理
        //异常是非常，程序语句应该具有一套完成的异常处理体系
    }

    private static int method1(int value1, int value2){
        if (value2 == 0) {
            return -1;
        }
        if (!(value1 >0 && value1<=100)) {
            return -2;
        }
        int value3 = value1/value2;
    }
}
```



```
        System.out.println("value3=" + value3);
        return 1;
    }
}
```

采用异常来处理参数非法

```
public class ExceptionTest16 {

    public static void main(String[] args) {
        //int ret = method1(10, 2);
        //System.out.println(ret);
        /*
        try {
            int ret = method1(1000, 10);
            System.out.println(ret);
        } catch (IllegalArgumentException iae) {
            //ide 为指向堆中的 IllegalArgumentException 对象的地址
            System.out.println(iae.getMessage());
        }
        */

        try {
            int ret = method1(1000, 10);
            System.out.println(ret);
        } catch (Exception iae) { //可以采用 Exception 拦截所有的异常
            System.out.println(iae.getMessage());
        }

    }

    private static int method1(int value1, int value2){
```

```
        if (value2 == 0) {  
            ////手动抛出异常  
            throw new IllegalArgumentException("除数为 0");  
        }  
        if (!(value1 > 0 && value1 <= 100)) {  
            //手动抛出异常  
            throw new IllegalArgumentException("被除数必须为 1~100 之间的数据");  
        }  
        int value3 = value1 / value2;  
        return value3;  
    }  
}
```

throws 和 throw 的区别? throws 是声明异常, throw 是抛出异常
进一步了解 throw

```
public class ExceptionTest17 {  
  
    public static void main(String[] args) {  
        try {  
            int ret = method1(1000, 10);  
            System.out.println(ret);  
        } catch (Exception iae) {  
            System.out.println(iae.getMessage());  
        }  
    }  
  
    private static int method1(int value1, int value2) {  
        try {  
            if (value2 == 0) {
```

```
    ///手动抛出异常
    throw new IllegalArgumentException("除数为 0");

    //加入如下语句编译出错，throw 相当于 return 语句
    //System.out.println("-----test11-----");
}
if (!(value1 > 0 && value1 <= 100)) {
    //手动抛出异常
    throw new IllegalArgumentException("被除数必须为 1~100 之间的数据");
}
int value3 = value1/value2;
return value3;
}finally {
    //throw 虽然类似 return 语句，但 finally 会执行的
    System.out.println("-----finally-----");
}
}
```



```
命令提示符
D:\share\JavaProjects\j2se\chapter04>java ExceptionTest17
finally
被除数必须为1~100之间的数据
```

2.3.8、异常的捕获顺序

异常的捕获顺序应该是：从小到大

```
import java.io.*;

public class ExceptionTest18 {

    public static void main(String[] args) {
```

```
try {
    FileInputStream fis = new FileInputStream("test.txt");
    fis.close();
} catch (IOException e) {
    e.printStackTrace();
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
//将 IOException 放到前面，会出现编译问题
//因为 IOException 是 FileNotFoundException 的父类，
//所以截获了 IOException 异常后，IOException 的子异常
//都不会执行到，所以再次截获 FileNotFoundException 没有任何意义
//异常的截获一般按照由小到大的顺序，也就是先截获子异常，再截获父
异常
}
```

2.4、如何自定义异常

自定义异常通常继承于 `Exception` 或 `RuntimeException`，到底继承那个应该看具体情况来定，关于异常再以后的项目中再继续讨论

【示例代码】，自定义受控异常

```
import java.io.*;

public class ExceptionTest19 {

    public static void main(String[] args) {
        try {
            method1(10, 0);
        } catch (MyException e) {
            //必须拦截,拦截后必须给出处理，如果不给出处理，就属于吃掉了该
```

异常

//系统将不给出任何提示，使程序的调试非常困难

```
System.out.println(e.getMessage());
```

```
}
```

```
}
```

```
private static void method1(int value1, int value2)
```

```
throws MyException { //如果是受控异常必须声明
```

```
    if (value2 == 0) {
```

```
        throw new MyException("除数为 0");
```

```
    }
```

```
    int value3 = value1 / value2;
```

```
    System.out.println(value3);
```

```
}
```

```
}
```

//自定义受控异常

```
class MyException extends Exception {
```

```
    public MyException() {
```

```
        //调用父类的默认构造函数
```

```
        super();
```

```
    }
```

```
    public MyException(String message) {
```

```
        //手动调用父类的构造方法
```

```
        super(message);
```

```
    }
```

```
}
```

【示例代码】，自定义非受控异常

```
import java.io.*;

public class ExceptionTest20 {

    public static void main(String[] args) {
        method1(10, 0);
    }

    private static void method1(int value1, int value2)
        //throws MyException {
        if (value2 == 0) {
            //抛出非受控异常，方法可以不适用 throws 进行声明
            //但也也可以显示的声明
            throw new MyException("除数为 0");
        }
        int value3 = value1 / value2;
        System.out.println(value3);
    }
}

//自定义非受控异常
class MyException extends RuntimeException {

    public MyException() {
        //调用父类的默认构造函数
        super();
    }

    public MyException(String message) {
```

```
//手动调用父类的构造方法
super(message);
}
}
```

2.5、方法覆盖与异常

方法覆盖的条件:

- 子类方法不能抛出比父类方法更多的异常，但可以抛出父类方法异常的子异常

```
import java.io.*;

public class ExceptionTest21 {

    public static void main(String[] args) {
    }
}

interface UserManager {

    public void login(String username, String password) throws
    UserNotFoundException;

}

class UserNotFoundException extends Exception {

}

class UserManagerImpl1 implements UserManager {
```

```
//正确
public void login(String username, String password) throws
UserNotFoundException {

}

}

//class UserManagerImpl2 implements UserManager {

//不正确，因为 UserManager 接口没有要求抛出 PasswordFailureException 异常
//子类异常不能超出父类的异常范围
//public void login(String username, String password) throws
UserNotFoundException, PasswordFailureException{

//}
//}

class UserManagerImpl3 implements UserManager {

//正确，因为 MyException 是 UserNotFoundException 子类
//MyException 异常没有超出接口的要求
public void login(String username, String password) throws
UserNotFoundException, MyException {

}

}

class PasswordFailureException extends Exception {
```



```
}  
  
class MyException extends UserNotFoundException {  
  
}
```

3、总结

- a) 异常的分类
- b) 受控异常和非受控异常的区别
- c) 异常的 5 个关键字 `try`、`catch`、`finally`、`throws`、`throw`
- d) 异常的捕获顺序，先捕获小的，再捕获大的
- e) 方法覆盖和异常的关系