Java进阶1 第3天

**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

1. **【应用】异常概述与分类**
2. 【理解】能够阐述异常的由来
3. 【应用】能够独立写出异常的继承体系
4. **【应用】异常的处理方式**
5. 【应用】能够使用try…catch处理单个异常和多个异常
6. 【应用】能够使用 throws 处理单个异常和多个异常
7. **【应用】异常的注意事项与自定义异常**
8. 【理解】能够阐述异常在继承中的注意事项
9. 【理解】能够阐述当有多个catch,catch中参数为父类的异常需要放在最后的原因
10. 【理解】能够阐述写在finally中代码的作用
11. 【理解】能够独立定义一个自定义异常,并使用该异常
12. **【应用】API帮助文档生成与日期概述**
13. 【应用】能够独立查询API帮助文档
14. 【应用】能够独立使用Eclipse导出API文档
15. **【应用】API常用类:Date,DateFormat,Calendar,Math,System**
16. 【应用】能够独立写出用Date对象设置毫秒值和获取毫秒值
17. 【应用】能够独立写出用DateFormat完成日期对象和日期字符串互相转换
18. 【应用】能够独立写出用DateFormat完成日期对象和日期字符串互相转换
19. 【应用】能够独立写出用Calendar获取日期和设置日期
20. 【理解】能够独立练习Math类和System类的常用方法
21. **【应用】包装类与正则表达式.**
22. 【理解】能够阐述基本类型对应哪些包装类
23. 【理解】能够阐述包装类的装箱和拆箱机制
24. 【应用】能够独立利用正则完成matches方法和split方法相关案例

# 异常概述与分类

## 异常概念

A:异常:就是程序运行中出现的不正常的情况.

B:异常的由来:程序中的问题也是现实生活中的一个具体的事物,也可以通过java的类的形式进行描述,并封装成对象.

其实异常就是java对不正常情况描述后的对象体现.

## 异常的分类

A:异常的继承体系

Throwable

Error (错误:一出现就是致命的)

服务器宕机,数据库崩溃等

Exception

RuntimeException(运行时异常,一般都是程序员犯的错误,需要修改源码的.)

编译时异常:在编译时必须进行处理,不处理无法通过编译.

### 案例代码一:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: ExceptionDemo  \* **@Description**: 异常  \* **@date** 2017年11月16日 上午9:17:26  \* Company www.igeekhome.com  \*  \* 概念:代码运行当中可能出现的不正常情况  \* java当中将这些已经预见到的不正常情况,封装成了类.即异常类.  \*  \* 产生异常:  \* 当满足了异常产生条件时,会执行以下两个动作  \* 创建异常对象  \* 抛出异常对象  \* 执行完该两个动作后,叫做产生了异常,只是这个过程是java自动执行的,我们看不到.  \*  \* 异常分类:  \* Throwable  \* Error(程序无法处理)  \* Exception(可以通过程序处理的)  \* 编译期异常(编译过程报错,必须处理)  \* 运行期异常RuntimeException(编译期可以不处理,但是运行时可能会报错)  \*  \* 处理:  \* 捕获处理  \* 声明抛出处理  \*  \*/  **public** **class** ExceptionDemo {  **public** **static** **void** main(String[] args) **throws** IOException {    //编译时异常  //"d:/test.txt"  //指定文件  Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("请输入文件路径：");  String path = sc.next();    FileWriter fw = **new** FileWriter(path);  //写出内容  //fw.write("i love java");  //关闭资源  //fw.close();  System.***out***.println("后边的代码");    //运行时异常  System.***out***.println(10 / 0);  }  } |

# 异常的处理方式

## try…catch处理单个异常

### try…catch格式:

|  |
| --- |
| try{  需要检测的异常；  } catch(异常类名异常变量) {  异常处理代码  可以调用异常的方法  通常我们只使用一个方法：printStackTrace打印异常信息  } |

### 案例代码二:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: TryCatchDemo  \* **@Description**: 异常捕获处理  \* **@date** 2017年11月16日 上午9:28:05  \* Company www.igeekhome.com  \*  \* 捕获处理，将异常获取，使用try/catch做分支处理  \*  \* try{  \* 需要检测的异常；  \* } catch(异常对象) {  \* 异常处理代码  \* 可以调用异常的方法  \* 通常我们只使用一个方法：printStackTrace打印异常信息  \* }  \*/  **public** **class** TryCatchDemo {  **public** **static** **void** main(String[] args) {  //编译时异常  //"d:/test.txt"  //指定文件  Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("请输入文件路径：");  String path = sc.next();    **try** {  FileWriter fw = **new** FileWriter(path);  System.***out***.println("输出成功了");  } **catch**(IOException e) {  System.***out***.println("您输入的路径有误!");  //用于展示异常出现的位置和基本信息  e.printStackTrace();  }    System.***out***.println("后边的代码");  }  } |

## 使用throws关键字处理异常

throws的方式处理异常

定义功能方法时，需要把出现的问题暴露出来让调用者去处理。

那么就通过throws在方法上标识

### throws格式

修饰符 返回值类型 方法名(参数类型参数名1,参数类型参数名2...) **throws 异常类名** {

}

### 案例代码三:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: ThrowsDemo  \* **@Description**: 异常声明抛出处理  \* **@date** 2017年11月16日 上午9:35:25  \* Company www.igeekhome.com  \*  \* 在某个方法的声明上,让它(这个方法)抛出这个产生的异常,自己不处理,谁调用这个方法,谁处理  \* 方法声明上加 throws 异常类名  \*  \* 如:  \* 方法a中产生了异常,声明抛出  \* 方法b调用方法a,则该异常由a转向了b  \*  \* 一般情况下,main方法是不会声明抛出异常的,因为该方法抛出,直接抛出给了JVM,JVM的默认处理机制是终止程序运行,则这样处理,相当于没有处理!  \*  \*/  **public** **class** ThrowsDemo {  **public** **static** **void** main(String[] args) {  **try** {  *method*();  } **catch** (IOException e) {  System.***out***.println("文件有误");  //调用方法,打印异常信息  e.printStackTrace();  }    System.***out***.println("后边的代码");  }  /\*\*  \* **@Title**: method  \* **@Description**: 抛出异常的方法  \* **@throws** IOException  \*/  **public** **static** **void** method() **throws** IOException {  *method2*();  }    /\*  \* 该方法的代码中,可能会产生IOException异常,该方法处理的方式是声明如果出现该异常,就抛出该异常,自身并不处理,哪个方法调用该方法,哪个方法处理  \*/  /\*\*  \* **@Title**: method2  \* **@Description**: 抛出异常的方法2  \* **@throws** IOException  \*/  **public** **static** **void** method2() **throws** IOException {    //编译时异常  //"d:/test.txt"  //指定文件  Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("请输入文件路径：");  String path = sc.next();    FileWriter fw = **new** FileWriter(path);  System.***out***.println("输出成功了");  }  } |

## 单个运行时异常处理

### 案例代码四:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: RuntimeExceptionDemo  \* **@Description**: 运行时异常处理  \* **@date** 2017年11月16日 上午9:42:39  \* Company www.igeekhome.com  \*  \* 该异常在编译期不检查语法错误的.可以处理,也可以不处理.  \*  \* 一般情况下,遇到java自身提供的运行时异常,都可以通过逻辑解决,避免异常产生.  \* 常见运行时异常:空指针异常,索引越界异常,类型转换异常  \*  \* 声明抛出处理  \* 捕获处理  \*/  **public** **class** RuntimeExceptionDemo {  **public** **static** **void** main(String[] args) {    **try** {  System.***out***.println(10/0);  } **catch** (ArithmeticException ae) {  //可以打印异常信息,也可以不打印  //ae.printStackTrace();    //如果出现异常的话  System.***out***.println("除数不能为0!");  }  System.***out***.println("后边的代码");    **try** {  *method*();  } **catch** (ArithmeticException ae) {  System.***out***.println("除数不能为0!");  }  }    /\*\*  \* **@Title**: method  \* **@Description**: 运算方法  \* **@throws** ArithmeticException  \*/  **public** **static** **void** method() **throws** ArithmeticException{  System.***out***.println(10/1);  }  } |

## 多个异常处理

### try…catch处理多个异常格式

|  |
| --- |
| try{  需要检测的异常；  } catch(异常类名1 异常变量1) {  异常处理代码  可以调用异常的方法  通常我们只使用一个方法：printStackTrace打印异常信息  } catch(异常类名2 异常变量2) {  异常处理代码  可以调用异常的方法  通常我们只使用一个方法：printStackTrace打印异常信息  }......//可以有无数个catch |

### throws 处理多个异常格式

修饰符 返回值类型 方法名(参数类型参数名1,参数类型参数名2...)

**throws 异常类名1,异常类名2,......**{

}

### 案例代码五:

|  |
| --- |
| **package** com.igeek\_04;  **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: MultiExceptionDemo  \* **@Description**: 多异常处理  \* **@date** 2017年11月16日 上午9:49:52  \* Company www.igeekhome.com  \*  \* 多异常处理一:  \* 声明抛出处理,直接抛出多个异常即可  \*  \* 多异常处理二:  \* 一个try多个catch的方式  \*  \* java是中断异常的处理方式  \*/  **public** **class** MultiExceptionDemo {  **public** **static** **void** main(String[] args) {    //一个try多个catch的方式  **try**{  *method*();  }**catch**(ArithmeticException ae) {  System.***out***.println("您的除数不能为0");  }**catch**(IOException e) {  System.***out***.println("您要访问的文件不存在");  }    System.***out***.println("后边的代码");  }  //一个方法声明抛出多个异常  /\*\*  \* **@Title**: method  \* **@Description**: 抛出异常的方法  \* **@throws** ArithmeticException  \* **@throws** IOException  \*/  **public** **static** **void** method() **throws** ArithmeticException,IOException{    System.***out***.println(10/1);  System.***out***.println("method逻辑1完成");    *method2*();  System.***out***.println("method逻辑2完成");  }    /\*\*  \* **@Title**: method2  \* **@Description**: 抛出异常的方法2  \* **@throws** IOException  \*/  **public** **static** **void** method2() **throws** IOException {  //指定文件  Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("请输入文件路径：");  String path = sc.next();    FileWriter fw = **new** FileWriter(path);  }  } |

### 多个异常用一个catch处理格式

try{

需要检测的异常；

} catch(**异常类名1|异常类名2……**异常变量) {

异常处理代码

可以调用异常的方法

通常我们只使用一个方法：printStackTrace打印异常信息

}

### 案例代码六:

|  |
| --- |
| **package** com.igeek\_04;  **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: MultiExceptionDemo  \* **@Description**: 多个异常用一个catch处理格式  \* **@date** 2017年11月16日 上午9:49:52  \* Company www.igeekhome.com  \*  \*  \* 1、多个异常分别每个try对应一个catch处理，不建议  \* 2、多个异常只有一个try一个catch  \* 3、一个try多个catch的方式  \*/  **public** **class** MultiExceptionDemo2 {  **public** **static** **void** main(String[] args) {    /\*多个异常分别处理  Scanner sc = new Scanner(System.in);  String path = sc.next();    try {  FileWriter fw = new FileWriter(path);  } catch (IOException e) {  System.out.println("您访问的文件不存在!");  }    try {  System.out.println(10/0);  } catch (ArithmeticException e) {  System.out.println("除数不能为0");  }  \*/    /\*多个异常一次try一次catch  Scanner sc = new Scanner(System.in);  String path = sc.next();    try {  FileWriter fw = new FileWriter(path);  System.out.println(10/0);  }catch(ArithmeticException|IOException ae) {  System.out.println("您访问的文件不存在!或者除数不能为0");  }    System.out.println("后边的代码");  \*/    //一个try多个catch的方式  Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("请输入文件路径：");  String path = sc.next();  **try**{  FileWriter fw = **new** FileWriter(path);  System.***out***.println(10/0);  }**catch**(ArithmeticException ae) {  System.***out***.println("您的除数不能为0");  }**catch**(IOException e) {  System.***out***.println("您要访问的文件不存在");  }    System.***out***.println("后边的代码");  }  } |

# 异常的注意事项与自定义异常

## 异常在继承中的注意事项

 A:异常注意事项

a:子类重写父类方法时，子类的方法必须抛出相同的异常或父类异常的子类。(父亲坏了,儿子不能比父亲更坏)

b:如果父类抛出了多个异常,子类重写父类时,只能抛出相同的异常或者是他的子集,子类不能抛出父类没有的异常

c:如果被重写的方法没有异常抛出,那么子类的方法绝对不可以抛出异常,如果子类方法内有异常发生,那么子类只能try,不能throws

### 案例代码七:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.io.IOException;  **import** java.text.ParseException;  **import** java.util.zip.DataFormatException;  /\*\*  \* **@ClassName**: MultiExceptionNotesDemo  \* **@Description**: 异常在继承中的注意事项  \* **@date** 2017年11月16日 上午10:21:59  \* Company www.igeekhome.com  \*  \* 如果父类抛出了多个异常,子类覆盖父类方法时,只能抛出相同的异常或者是他的子集,只能少不能多  \* 父类方法没有抛出异常，子类覆盖父类该方法时也不可抛出异常。此时子类产生该异常，只能捕获处理，不能声明抛出  \*  \* 现实开发过程中,一般子类重写的方法抛出的异常与父类方法抛出的异常是一样的,不存在不一样的情况.  \*/  **public** **class** MultiExceptionNotesDemo {  // 为了授课方便,则在一个文件中定义多个类,其他类不能使用public修饰    // 如果父类抛出了多个异常,子类覆盖父类方法时,只能抛出相同的异常或者是他的子集  /\*\*  \* **@ClassName**: Fu  \* **@Description**: 父类  \* **@date** 2017年11月16日 上午10:40:05  \* Company www.igeekhome.com  \*  \*/  **class** Fu {  /\*\*  \* **@Title**: method  \* **@Description**: 父类中抛出异常的方法  \* **@throws** IOException  \* **@throws** ParseException  \* **@throws** DataFormatException  \*/  **public** **void** method() **throws** IOException,ParseException,DataFormatException{  //产生异常的方法体  }  }    /\*\*  \* **@ClassName**: Zi  \* **@Description**: 子类  \* **@date** 2017年11月16日 上午10:40:15  \* Company www.igeekhome.com  \*  \*/  **class** Zi **extends** Fu {    /\*\*  \* **@Title**: method  \* **@Description**: 子类中重写父类的方法  \* **@throws** IOException  \* **@throws** ParseException  \* **@see** javaadvanced1\_d03\_c03.MultiExceptionNotesDemo.Fu#method()  \*/  @Override  **public** **void** method() **throws** IOException, ParseException {  //子类的方法体  }  }  } |

|  |
| --- |
| **package** com.igeek\_01;  **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: MultiExceptionNotesDemo  \* **@Description**: 异常在继承中的注意事项2  \* **@date** 2017年11月16日 上午10:21:59  \* Company www.igeekhome.com  \*  \* 如果父类抛出了多个异常,子类覆盖父类方法时,只能抛出相同的异常或者是他的子集,只能少不能多  \* 父类方法没有抛出异常，子类覆盖父类该方法时也不可抛出异常。此时子类产生该异常，只能捕获处理，不能声明抛出  \*  \* 现实开发过程中,一般子类重写的方法抛出的异常与父类方法抛出的异常是一样的,不存在不一样的情况.  \*/  **public** **class** MultiExceptionNotesDemo2 {  // 为了授课方便,则在一个文件中定义多个类,其他类不能使用public修饰    // 父类方法没有抛出异常，子类覆盖父类该方法时也不可抛出异常。此时子类产生该异常，只能捕获处理，不能声明抛出  /\*\*  \* **@ClassName**: Fu  \* **@Description**: 父类  \* **@date** 2017年11月16日 上午10:41:38  \* Company www.igeekhome.com  \*  \*/  **class** Fu {  /\*\*  \* **@Title**: method  \* **@Description**: 父类中的方法  \*/  **public** **void** method() {  // 不产生异常的方法体  }  }  /\*\*  \* **@ClassName**: Zi  \* **@Description**: 子类  \* **@date** 2017年11月16日 上午10:41:50  \* Company www.igeekhome.com  \*  \*/  **class** Zi **extends** Fu {  /\*\*  \* **@Title**: method  \* **@Description**: 子类重写父类的方法  \* **@see** javaadvanced1\_d03\_c03.MultiExceptionNotesDemo2.Fu#method()  \*/  @Override  **public** **void** method() {  // 子类的方法体  // 编译时异常  //"d:/test.txt"  //指定文件  Scanner sc = **new** Scanner(System.***in***);  String path = sc.next();  **try** {  FileWriter fw = **new** FileWriter(path);  } **catch** (IOException e) {  System.***out***.println("文件不存在");  }  }  }  } |

## 处理多个构成子父类关系的异常

### 案例代码八:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: MultiExceptionNotesDemo  \* **@Description**: catch多个异常时的注意事项  \* **@date** 2017年11月16日 上午10:21:59  \* Company www.igeekhome.com  \*  \* 在捕获异常时,可以使用父类异常的引用指向子类异常的对象.是多态的应用.  \* 当多异常处理时，捕获处理，前边的类不能是后边类的父类  \*/  **public** **class** MultiExceptionNotesDemo {  **public** **static** **void** main(String[] args) {  //一个try多个catch的方式  Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("请输入文件路径：");  String path = sc.next();    **try**{  System.***out***.println(10/0);  FileWriter fw = **new** FileWriter(path);  }**catch**(IOException e) {  System.***out***.println("您要访问的文件不存在");  }**catch**(Exception ae) {  System.***out***.println("您的除数不能为0");  }    System.***out***.println("后边的代码");    /\*多个异常一次try一次catch,异常之间不能存在子父类关系    Scanner sc = new Scanner(System.in);  String path = sc.next();    try {  FileWriter fw = new FileWriter(path);  System.out.println(10/0);  }catch(IOException|Exception ae) {  System.out.println("您访问的文件不存在!或者除数不能为0");  }    System.out.println("后边的代码");  \*/  }  } |

## finally关键字

|  |
| --- |
| try{  需要检测的异常；  } catch(异常类名异常变量) {  异常处理代码  可以调用异常的方法  通常我们只使用一个方法：printStackTrace打印异常信息  } finally{  此处存放一定要执行的代码  一般把释放资源的代码放到finally中  } |

### 案例代码九:

|  |
| --- |
| **package** com.igeek\_03;  **import** java.io.FileWriter;  **import** java.io.IOException;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: FinallyDemo  \* **@Description**: finally代码块  \* **@date** 2017年11月16日 上午10:55:49  \* Company www.igeekhome.com  \*  \* 在try/catch后可以追加finally代码块，其中的代码一定会被执行，通常用于资源回收。  \*/  **public** **class** FinallyDemo {  **public** **static** **void** main(String[] args) {    //编译时异常  //"d:/test.txt"  //指定文件  Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("请输入文件路径：");  String path = sc.next();    **try** {  FileWriter fw = **new** FileWriter(path);  fw.write("abc");  } **catch** (IOException e) {  System.***out***.println("产生了IO问题");  } **finally** {  System.***out***.println("我要关闭资源");  }    System.***out***.println("后边的代码");  }  } |

## 自定义异常

在上述代码中，发现这些异常都是JDK内部定义好的，并且这些异常不好找。书写时也很不方便，那么能不能自己定义异常呢？

之前的几个异常都是java通过类进行的描述，并将问题封装成对象，异常就是将问题封装成了对象。

这些异常不好认，书写也很不方便，能不能定义一个符合我的程序要求的异常名称。

既然JDK中是使用类在描述异常信息，那么我们也可以模拟Java的这种机制，我们自己定义异常的信息，异常的名字，让异常更符合自己程序的阅读。

准确对自己所需要的异常进行类的描述

### 自定义异常格式

Class 异常名 extends Exception{ **//或继承RuntimeException**

public 异常名(){

}

public 异常名(String s){

super(s);

}

}

### 案例代码十:

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: U18Exception  \* **@Description**: 小于18岁异常  \* **@date** 2017年11月16日 上午11:01:20  \* Company www.igeekhome.com  \*  \*/  **public** **class** U18Exception **extends** Exception {    /\*\*  \* **@Title**: U18Exception  \* **@param** message  \*/  **public** U18Exception(String message) {  **super**(message);  }  } |

|  |
| --- |
| **package** com.igeek\_04;  **import** java.util.Scanner;  /\*\*  \* **@ClassName**: CustomExceptionDemo  \* **@Description**: 自定义异常的测试类  \* **@date** 2017年11月16日 上午11:02:10  \* Company www.igeekhome.com  \*  \* 自定义异常:  \* 编译时异常继承Exception，运行时异常继承RuntimeException。  \* 关键字 throw 抛出异常  \* 回忆产生异常:  \* 创建异常对象  \* 抛出异常  \*/  **public** **class** CustomExceptionDemo {  **public** **static** **void** main(String[] args) {  //年龄小于18岁,产生异常  Scanner sc = **new** Scanner(System.***in***);  System.***out***.println("请输入年龄：");  **int** age = sc.nextInt();    **if**(age < 18) {  //产生异常  //创建异常对象  U18Exception u18 = **new** U18Exception("年龄小于18岁");  //抛出异常  **try** {  **throw** u18;  } **catch** (U18Exception e) {  System.***out***.println("年龄不满足婚恋网站年龄");  e.printStackTrace();  }  }    //可以直接手动创建异常对象  **try** {  **throw** **new** RuntimeException("我产生了一个运行时异常");  } **catch** (Exception e) {  System.***out***.println("直接创建了一个RuntimeException对象");  }    System.***out***.println("后边的代码");  }  } |

# API帮助文档生成与日期概述

## API帮助文档

API（Application Programming Interface,应用程序编程接口）是一些预先定义的函数，目的是基于某软件或硬件提供应用程序给予开发人员访问，无需访问源码，或理解内部工作机制的细节。

API帮助文档就是对这些预先定义的函数写的文档，帮助开发人员了解函数的使用方法和功能,达到快速上手的目的

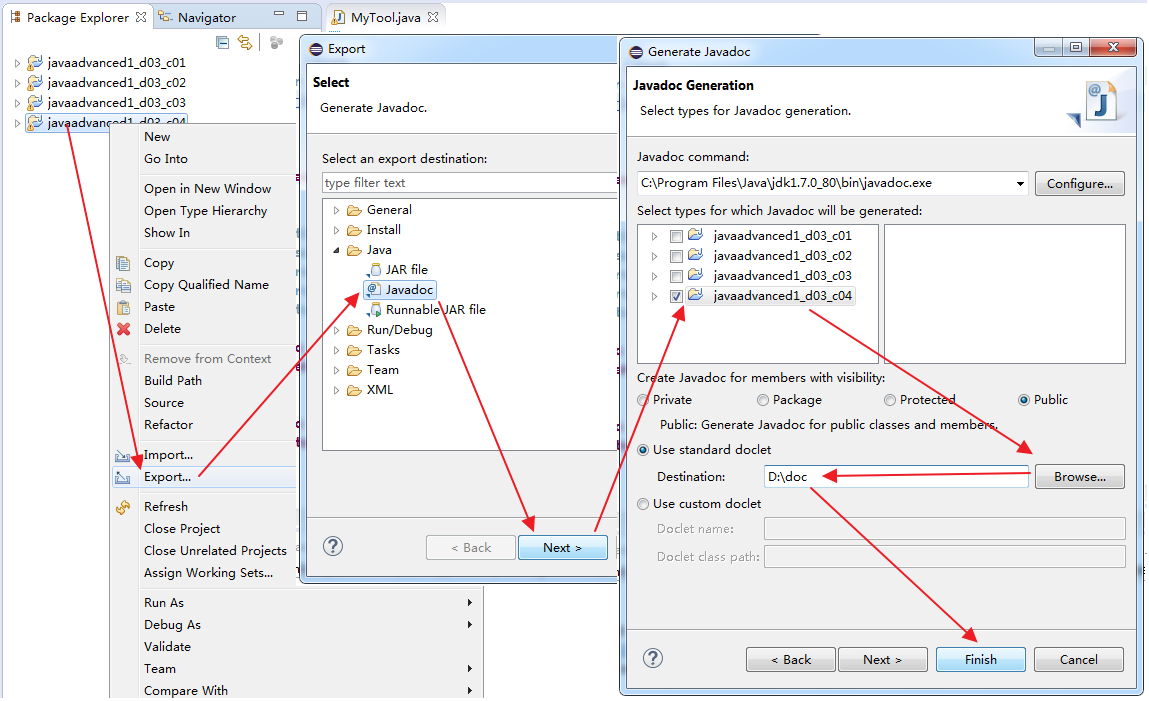
## Eclipse导出API帮助文档

在eclipse使用时，可以配合文档注释，导出对类的说明文档，从而供其他人阅读学习与使用。

通过使用文档注释，将类或者方法进行注释用@简单标注基本信息。如@author 作者、@version代码版本、@param方法参数、@return方法返回值等。

### 案例代码十:

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: MyTool  \* **@Description**: 文档注释的作用  \* **@date** 2017年11月16日 下午1:39:48  \* Company www.igeekhome.com  \*  \* 使用文档注释完成帮助文档的生成,其过程就是通过文档注释注解的方式完成类或者方法的说明,使用eclipse完成文档生成  \*  \* **@author** igeek  \* **@version** V0.1  \* **@since** java还没有出现的时候  \*  \*/  **public** **class** MyTool {  /\*\*  \* **@Title**: add2Number  \* **@Description**: 求两个数的和  \* **@param** a 参与计算的第一个数  \* **@param** b 参与计算的第二个数  \* **@return** 两个参数的和  \*/  **public** **int** add2Number(**int** a,**int** b) {  **return** a+b;  }    **public** **void** method(){}    **private** **void** method2(){}  } |



## 日期概述:

日期：Date类是操作最简单的日期类，代表一个瞬间。

以毫秒作为标记，即距离1970年1月1日00:00:00所经过的毫秒值，再通过毫秒值计算出对应的年、月、日、时、分、秒、星期等时间信息。

# API常用类:Date,DateFormat,Calendar,Math,System

## Date类方法

A：构造方法：

public Date() //返回当前时间

public Date(long date) //返回指定毫秒值的日期对象

B：普通方法：

public long getTime() //获取当前时间对象的毫秒值

public void setTime(long time) //设置时间毫秒值

### 案例代码十一:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.Date;  /\*\*  \* **@ClassName**: DateDemo  \* **@Description**: 日期类的测试类  \* **@date** 2017年11月16日 下午2:02:32  \* Company www.igeekhome.com  \*  \* API:application programming interface  \* 应用程序变成时用到的接口  \* 应用程序编程接口  \* 白话:java提供的类或者接口  \*  \* 日期:Date类是操作最简单的日期类，代表一个瞬间。  \* 以毫秒作为标记,即距离1970年1月1日00:00:00所经过的毫秒值,再通过毫秒值计算出对应的年、月、日、时、分、秒、星期等时间信息  \*  \* 构造方法：  \* public Date() //返回当前时间  \* public Date(long date) //返回指定毫秒值的日期对象  \* 普通方法:  \* public long getTime() //获取当前时间对象的毫秒值  \* public void setTime(long time) //设置时间毫秒值  \*  \*/  **public** **class** DateDemo {  **public** **static** **void** main(String[] args) {  //创建当前时间的Date对象  Date date = **new** Date();    System.***out***.println(date);    **long** time = date.getTime();  System.***out***.println(time);    //制定毫秒值创建Date对象  Date date2 = **new** Date(0L);  System.***out***.println(date2);    //修改日期毫秒值  date2.setTime(14962627904L);  System.***out***.println(date2);  }  } |

## DateFormat类方法

DateFormat 是日期/时间格式化子类的抽象类，它以与语言无关的方式格式化并解析日期或时间。

日期/时间格式化子类（如 **SimpleDateFormat类**）允许进行格式化（也就是日期 -> 文本）、解析（文本-> 日期）和标准化。

我们通过这个类可以帮我们完成日期和文本之间的转换。

### 日期对象格式化为日期字符串

#### 案例代码十二:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.text.DateFormat;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  /\*\*  \* **@ClassName**: DateFormatDate2StringDemo  \* **@Description**: Date对象转为String字符串  \* **@date** 2017年11月16日 下午2:07:01  \* Company www.igeekhome.com  \*  \* DateFormat:日期格式化类  \* SimpleDateFormat:实际使用的日期格式化子类  \*  \* 在创建SimpleDateFormat对象时,可以指定生成字符串的模板,规则见API帮助文档  \* 模板是一个字符串,代表转换规则:特殊字母代表一定的时间组成部分  \*  \* public final String format(Date date) 将日期格式化成字符串  \*/  **public** **class** DateFormatDate2StringDemo {  **public** **static** **void** main(String[] args) {    //定义日期规则字符串  String rule = "yyyy年MM月dd日HH:mm:ss";  //使用日期转换规则,创建日期格式化类对象  DateFormat format = **new** SimpleDateFormat(rule);    //准备要转换的日期对象  Date dNow = **new** Date();  //转换Date对象为String字符串  String sNow = format.format(dNow);    //打印结果  System.***out***.println(sNow);  System.***out***.println(dNow);  }  } |

### 日期字符串解析为日期对象

#### 案例代码十三:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.text.DateFormat;  **import** java.text.ParseException;  **import** java.text.SimpleDateFormat;  **import** java.util.Date;  /\*\*  \* **@ClassName**: DateFormatString2DateDemo  \* **@Description**: String字符串转为Date对象  \* **@date** 2017年11月16日 下午2:09:13  \* Company www.igeekhome.com  \*  \* DateFormat:日期格式化类  \* SimpleDateFormat:实际使用的日期格式化子类  \* 在创建SimpleDateFormat对象时,可以指定生成字符串的模板,规则见API帮助文档  \* 模板是一个字符串,代表转换规则:特殊字母代表一定的时间组成部分  \*  \* public Date parse(String source) throws ParseException  \*/  **public** **class** DateFormatString2DateDemo {  **public** **static** **void** main(String[] args) **throws** ParseException {    //定义日期规则字符串  String rule = "yyyy年MM月dd日HH:mm:ss";  //创建日期格式化类对象  DateFormat format = **new** SimpleDateFormat(rule);    //准备日期字符串  String sTime = "2017年11月06日 14:10:28";  //转换生成Date对象  Date dTime = format.parse(sTime);    //打印Date对象  System.***out***.println(dTime);  }  } |

## Calendar类方法

A:获取日历类的子类对象:

public static Calendar getInstance()

B:获取时间属性的方法:

public int get(int field)

//获取时间字段值，字段参见帮助文档 int field对应的字段

public final void set(intyear,intmonth,int date)

//设置年月日，可查看对应重载

public final Date getTime()

//获取该日历对象转成的日期对象

### 案例代码十四:

|  |
| --- |
| **package** com.igeek\_03;  **import** java.util.Calendar;  **import** java.util.Date;  /\*\*  \* **@ClassName**: CalendarDemo  \* **@Description**: Calendar:日历类的使用  \* **@date** 2017年11月16日 下午2:13:06  \* Company www.igeekhome.com  \*  \* Date类的大批方法(尤其是获取各种时间属性值得方法)都过期,被Calendar类替代了.  \*  \* Calendar:日历类,将各种时间的属性信息以字段的方式记录下来  \*  \* 创建Calendar对象,不使用构造方法,使用一下方法,支持语言敏感的问题,静态方法getInstance,获取当前时间  \* Calendar rightNow = Calendar.getInstance();  \*  \* 获取时间属性的方法:  \* public int get(int field) //获取时间字段值，字段参见帮助文档int field对应的字段  \* public final void set(int year,int month,int date) //设置年月日，可查看对应重载  \* public final Date getTime() //获取该日历对象转成的日期对象  \*/  **public** **class** CalendarDemo {  **public** **static** **void** main(String[] args) {  //创建日历类对象,当前时间  Calendar rightNow = Calendar.*getInstance*();  System.***out***.println(rightNow);    //Calendar转成Date对象  Date time = rightNow.getTime();  System.***out***.println(time);    //获取属性值  **int** year = rightNow.get(Calendar.***YEAR***);  System.***out***.println(year);    **int** month = rightNow.get(Calendar.***MONTH***)+1;  System.***out***.println(month);    **int** day = rightNow.get(Calendar.***DAY\_OF\_MONTH***);  System.***out***.println(day);    //设置年月日  rightNow.set(2008, 8-1, 8);  System.***out***.println(rightNow);  }  } |

## Math类方法

public static int abs(int a)：返回a的绝对值  
public static double ceil(double a) ：求大于等于a的最小整数,返回该整数的小数形式

public static double floor(double a) ：求小于等于a的最小整数

public static long round(double a) ：对a进行四舍五入,返回四舍五入后的值

public static double pow(double a, double b) ：求a的b次幂,返回double形式

### 案例代码十五:

|  |
| --- |
| **package** com.igeek\_04;  /\*\*  \* **@ClassName**: MathDemo  \* **@Description**: Math:数学工具类  \* **@date** 2017年11月16日 下午2:19:58  \* Company www.igeekhome.com  \*  \* 如果方法不需要访问成员变量,只关注方法功能本身,则该方法可以定义为静态方法,通常定义这样方法的类为工具类  \*  \* Math当中所有字段与方法均为静态,无需创建对象(无法创建对象).直接使用类名调用.  \*/  **public** **class** MathDemo {  **public** **static** **void** main(String[] args) {    //访问静态常量  System.***out***.println(Math.***PI***);    //访问静态方法  //求绝对值  System.***out***.println(Math.*abs*(-105));    //向上向下取整  System.***out***.println(Math.*ceil*(1.26));  System.***out***.println(Math.*floor*(1.26));  System.***out***.println(Math.*round*(1.26));    //求第一个参数的,第二个参数次方是多少  System.***out***.println(Math.*pow*(10, 3));  }  } |

## System类方法

public static void exit(int status)：让操作系统终止该java程序 0代表正常终止

public static void gc()：暗示java虚拟机回收不使用的对象(垃圾)

public static String getProperty(String key) ：查找制定的系统属性 , 系统属性参照API

### 案例代码十六:

|  |
| --- |
| **package** com.igeek\_05;  /\*\*  \* **@ClassName**: SystemDemo  \* **@Description**: System类常用方法  \* **@date** 2017年11月16日 下午2:25:32  \* Company www.igeekhome.com  \*  \*/  **public** **class** SystemDemo {  **public** **static** **void** main(String[] args) {  //标准输出与错误输出  System.***out***.println("i love java");  System.***err***.println("i love java");    //获取指定的系统属性  System.***out***.println(System.*getProperty*("os.name"));    //终止java程序  System.*exit*(0);  System.***out***.println("后边的代码");  }  } |

# 包装类与正则表达式

## 基本类型包装类简介

Java中提供了相应的对象来解决该问题，基本数据类型对象包装类。

java将基本数据类型值封装成了对象。

封装成对象有什么好处？可以提供更多的操作基本数值的功能。

8种基本类型对应的包装类如下：

其中需要注意int对应的是Integer，char对应的Character，其他6个都是基本类型首字母大写即可。

### 案例代码十七:

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: PrimaryDataTypeDemo  \* **@Description**: 基本类型对应的包装器类  \* **@date** 2017年11月16日 下午2:32:42  \* Company www.igeekhome.com  \*  \* 每一种基本数据类型都对应一种引用数据类型,如int对应Integer  \*  \* 引用类型在基本数据类型的基础上拥有更多的字段与功能  \* 如:  \* 最大值/最小值字段：MAX\_VALUE/MIN\_VALUE  \* public static String toBinaryString(int i) 将整数转为二进制的整数字符串  \*/  **public** **class** PrimaryDataTypeDemo {  **public** **static** **void** main(String[] args) {    //定义基本类型10  **int** i = 10;  //定义引用类型10  Integer i2 = **new** Integer(10);    //查看integer可以存放的最大/最小的数  System.***out***.println(Integer.***MAX\_VALUE***);  System.***out***.println(Integer.***MIN\_VALUE***);    //将整数转为二进制字符串  String binaryString = Integer.*toBinaryString*(10);  System.***out***.println(binaryString);  }  } |

## 自动拆箱与装箱机制

在需要的情况下，基本类型与包装类型可以通用。有些时候我们必须使用引用数据类型时，可以传入基本数据类型。

比如：

基本类型可以使用运算符直接进行计算，但是引用类型不可以。而基本类型包装类作为引用类型的一种却可以计算，原因在于，Java“偷偷地”自动地进行了对象向基本数据类型的转换。

相对应的，引用数据类型变量的值必须是new出来的内存空间地址值，而我们可以将一个基本类型的值赋值给一个基本类型包装类的引用。原因同样在于Java又“偷偷地”自动地进行了基本数据类型向对象的转换。

**自动拆箱：将基本类型包装类的对象转成该基本类型的数值**

**自动装箱：将基本类型的数值转成基本类型对应的包装类对象**

### 案例代码十八:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.ArrayList;  /\*\*  \* **@ClassName**: PrimaryDataTypeDemo  \* **@Description**: 自动拆箱与装箱机制  \* **@date** 2017年11月16日 下午2:32:42  \* Company www.igeekhome.com  \*  \* 自动拆箱:  \* 当希望使用基本数据类型时,可以给于基本数据类型包装类的对象.此时会自动由引用类型"拆箱"成了基本类型,再使用  \* 如:  \* 运算符只能操作基本类型,但是给予两个Integer的对象,其自动拆箱后,进行了计算.  \*  \* 自动装箱:  \* 当希望使用基本数据类型包装类的对象时,可以给于基本数据类型的值.此时会自动由基本类型"装箱"成了引用类型的对象,再使用  \* 如:  \* 集合中只能存放引用数据类型,但是可以给于基本类型的值,其自动装箱成对象后,再操作  \*/  **public** **class** PrimaryDataTypeDemo {    **public** **static** **void** main(String[] args) {    **int** i = 10;  **int** i2 = 20;  System.***out***.println(i + i2);    //自动拆箱  //运算符只能操作基本类型  //Scanner sc = new Scanner(System.in);  //Scanner sc2 = new Scanner(System.in);  //System.out.println(sc + sc2);    Integer i3 = **new** Integer(100);  Integer i4 = **new** Integer(200);    System.***out***.println(i3 + i4);    //自动装箱  Integer i5 = 300;    ArrayList<Integer> list = **new** ArrayList<Integer>();    list.add(10);  list.add(20);  list.add(30);  }  } |

## 正则表达式

### 正则表达式概述

正则表达式是专门解决字符串规则匹配的工具。

正则表达式也是一个字符串，用来定义匹配规则。

参照帮助文档，在Pattern类中有简单的规则定义，可以结合字符串类的方法使用。

### 正则表达式匹配规则

参照帮助文档，在Pattern类中有正则表达式的的规则定义，正则表达式中明确区分大小写字母。我们来学习语法规则。

正则表达式的语法规则：

字符：x

含义：代表的是字符x

例如：匹配规则为 "a"，那么需要匹配的字符串内容就是 ”a”

字符：\\

含义：代表的是反斜线字符'\'

例如：匹配规则为"\\" ，那么需要匹配的字符串内容就是 ”\”

字符类：[abc]

含义：代表的是字符a、b 或 c

例如：匹配规则为"[abc]" ，那么需要匹配的内容就是字符a，或者字符b，或字符c的一个

字符类：[^abc]

含义：代表的是除了 a、b 或 c以外的任何字符

例如：匹配规则为"[^abc]"，那么需要匹配的内容就是不是字符a，或者不是字符b，或不是字符c的任意一个字符

字符类：[a-zA-Z]

含义：代表的是a 到 z 或 A 到 Z，两头的字母包括在内

例如：匹配规则为"[a-zA-Z]"，那么需要匹配的是一个大写或者小写字母

字符类：[0-9]

含义：代表的是 0到9数字，两头的数字包括在内

例如：匹配规则为"[0-9]"，那么需要匹配的是一个数字

字符类：[a-zA-Z\_0-9]

含义：代表的字母或者数字或者下划线(即单词字符)

例如：匹配规则为" [a-zA-Z\_0-9] "，那么需要匹配的是一个字母或者是一个数字或一个下滑线

预定义字符类：.

含义：代表的是任何字符

例如：匹配规则为" . "，那么需要匹配的是一个任意字符。如果，就想使用 . 的话，使用匹配规则"\\."来实现

预定义字符类：\d [0-9]

含义：代表的是 0到9数字，两头的数字包括在内，相当于[0-9]

例如：匹配规则为"\d "，那么需要匹配的是一个数字

预定义字符类：\w [a-zA-Z\_0-9]

含义：代表的字母或者数字或者下划线(即单词字符)，相当于[a-zA-Z\_0-9]

例如：匹配规则为"\w "，，那么需要匹配的是一个字母或者是一个数字或一个下滑线

边界匹配器：\b

含义：代表的是单词边界

例如：匹配规则为"\b[abc]\b" ，那么代表的是字母a或b或c的左右两边需要的是非单词字符([a-zA-Z\_0-9])

数量词：X?

含义：代表的是X出现一次或一次也没有

例如：匹配规则为"a?"，那么需要匹配的内容是一个字符a，或者一个a都没有

数量词：X\*

含义：代表的是X出现零次或多次

例如：匹配规则为"a\*" ，那么需要匹配的内容是多个字符a，或者一个a都没有

数量词：X+

含义：代表的是X出现一次或多次

例如：匹配规则为"a+"，那么需要匹配的内容是多个字符a，或者一个a

数量词：X{n}

含义：代表的是X出现恰好 n 次

例如：匹配规则为"a{5}"，那么需要匹配的内容是5个字符a

数量词：X{n,}

含义：代表的是X出现至少 n 次

例如：匹配规则为"a{5, }"，那么需要匹配的内容是最少有5个字符a

数量词：X{n,m}

含义：代表的是X出现至少 n 次，但是不超过 m 次

例如：匹配规则为"a{5,8}"，那么需要匹配的内容是有5个字符a 到 8个字符a之间

### 案例代码十九:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: RegexDemo  \* **@Description**: 正则表达式:regex的使用  \* **@date** 2017年11月16日 下午2:48:47  \* Company www.igeekhome.com  \*  \* 正则表达式:regex  \* 匹配字符串的规则  \*  \* 正则表达式本身也是一个字符串  \*  \* 在使用正则表达式匹配字符串时,一共会至少出现两个字符串:代表规则的字符串,被规则匹配的字符串  \*  \* 正则表达式规则复杂,在日常的开发过程中,一般使用其他资源已经提供好的正则  \* 正则表达式使用相对广泛,并不属于某个具体的语言,而是各种语言都有对应的API操作正则  \*  \* 这里使用字符串中相关正则的方法  \* public boolean matches(String regex)  \* 被匹配的字符串调用方法,接收正则表达式字符串,如果匹配返回true,否则返回false  \* public String[] split(String regex)  \* 被匹配的字符串调用方法,将字符串中的字符与规则匹配,如果符合就匹配为分隔符,将一个完整的字符串通过这些分隔符切割为不同的子字符串  \*/  **public** **class** RegexDemo {  **public** **static** **void** main(String[] args) {    //验证电话号码  //准备被匹配的字符串  String phoneNumber = "13546546587";    //准备规则字符串  String regex = "1[34578][0-9]{9}";  //正则匹配,返回结果  **boolean** matches = phoneNumber.matches(regex);  System.***out***.println(phoneNumber+"是否为电话号码:"+matches);    //准备被匹配切割的字符串  String s = "jack rose obama trump";  //准备规则字符串  //String sRegex = "[,]"; //匹配,  //String sRegex = "[ ,]"; //匹配空格与,  String sRegex = " +"; //匹配一个或多个空格字符    //匹配切割  String[] sStringArray = s.split(sRegex);  //查看结果  **for**(**int** i = 0; i<sStringArray.length; i++) {  System.***out***.println(sStringArray[i]);  }  }  } |

重点和总结

1、异常的概述与分类 ：编译时异常、运行时异常

2、各种异常形式的不同处理方式

4、异常处理中的相关关键字：try、catch、throw、throws、finally

5、异常处理中的相关注意事项

6、自定义异常

7、API帮助文档，及如何把自己写的doc注释转成帮助文档

8、和时间、日期有关的常用类：Date，DateFormat，Calendar

9、其他的常用类：数学函数类Math、系统类System

10、基本类型对应的包装器类

11、字符串的正则表达式