# Stream、异常体系

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

## 今天同学们需要学会什么

| 不可变集合 | Stream流 | 认识异常体系 |
| --- | --- | --- |
| 有些业务场景下需要有不可变集合对象，Java如何得到不可变集合对象 | 集合自己提供的API非常繁琐，JDK 8开始,得益于Lambda，提供了操作集合、数组更好用的技术：Stream流 | 程序一旦出现了bug则会终止，如何尽力避免程序出现异常，出现异常如何进行处理让程序更稳健 |

# 目录

Contents

## 什么是不可变集合？

- 不可变集合，就是不可被修改的集合。
- 集合的数据项在创建的时候提供，并且在整个生命周期中都不可改变。否则报错。

| | | | | |
|---|---|---|---|---|
| “迪丽热巴" | “古力娜扎" | “卡尔扎巴" | “马儿扎哈" | ... |

老王

## 为什么要创建不可变集合？

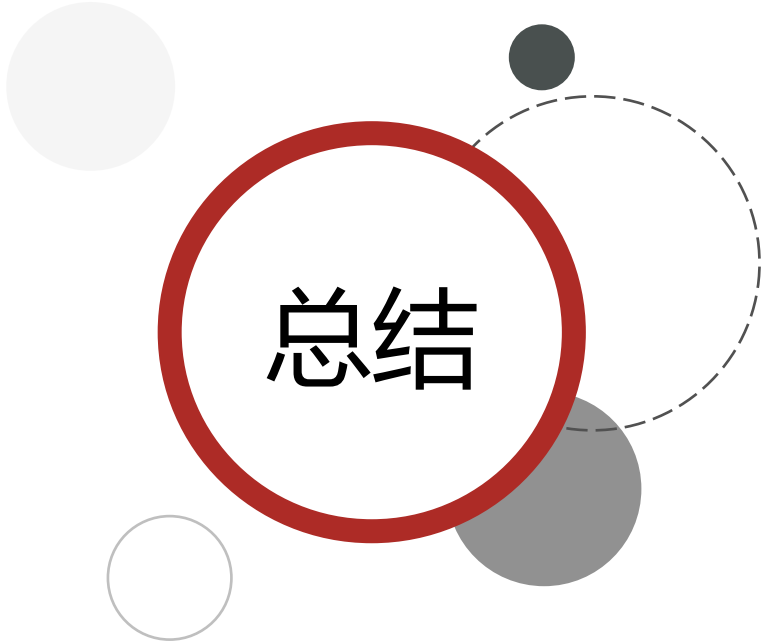
- 如果某个数据不能被修改，把它防御性地拷贝到不可变集合中是个很好的实践。
- 或者当集合对象被不可信的库调用时，不可变形式是安全的。

## 如何创建不可变集合？

- 在List、Set、Map接口中，都存在of方法，可以创建一个不可变的集合。

| 方法名称 | 说明 |
| --- | --- |
| static <E> List<E> of(E...elements) | 创建一个具有指定元素的List集合对象 |
| static <E> Set<E> of(E...elements) | 创建一个具有指定元素的Set集合对象 |
| static <K , V> Map<K , V> of(E...elements) | 创建一个具有指定元素的Map集合对象 |

- 这个集合不能添加，不能删除，不能修改。




1. 不可变集合的特点？
    - **定义完成后不可以修改，或者添加、删除**
2. 如何创建不可变集合？
    - **List、Set、Map接口中，都存在of方法可以创建不可变集合。**

# 目录

Contents

## 什么是Stream流？

- 在Java 8中，得益于Lambda所带来的函数式编程， 引入了一个全新的Stream流概念。
- **目的：用于简化集合和数组操作的API。**

## Stream流思想

**Stream流式思想的核心：**

1. 先得到集合或者数组的Stream流（就是一根传送带）
2. 把元素放上去
3. 然后就用这个Stream流简化的API来方便的操作元素。

## 案例 体验Stream流的作用

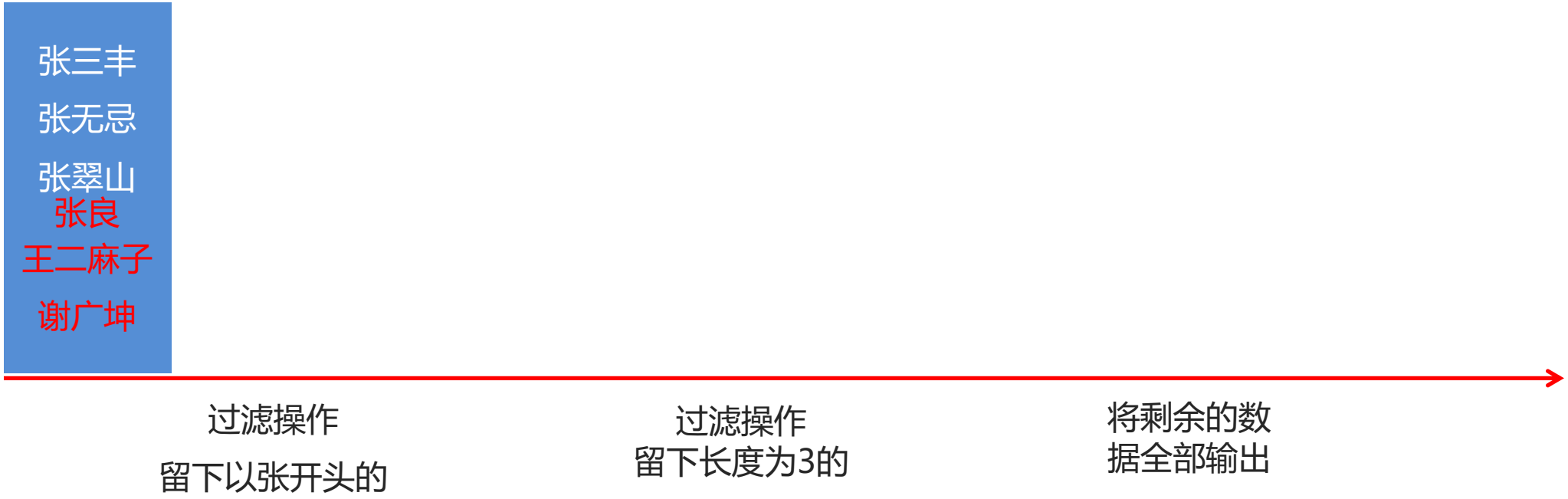
需求：按照下面的要求完成集合的创建和遍历

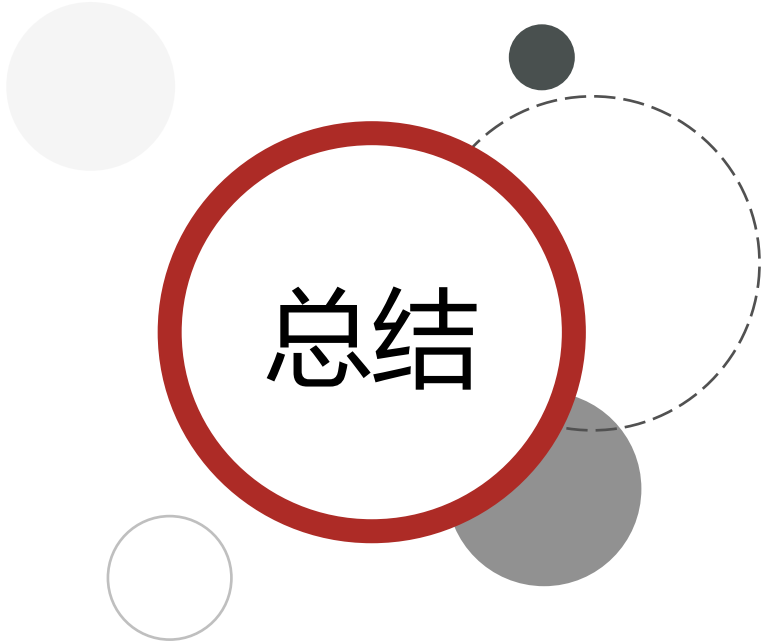
- 创建一个集合，存储多个字符串元素

```
List<String> list = new ArrayList<>();
list.add("张无忌");
list.add("周芷若");
list.add("赵敏");
list.add("张强");
list.add("张三丰");
```

- 把集合中所有以"张"开头的元素存储到一个新的集合
- 把"张"开头的集合中的长度为3的元素存储到一个新的集合
- 遍历上一步得到的集合中的元素输出。

## Stream流的思想


张三丰
张无忌
张翠山
张良
王二麻子
谢广坤
过滤操作
留下以张开头的
过滤操作
留下长度为3的
将剩余的数
据全部输出




1、Stream流的作用是什么，结合了什么技术？

- **简化集合、数组操作的API。结合了Lambda表达式。**

2、说说Stream流的思想和使用步骤。

- **先得到集合或者数组的Stream流（就是一根传送带）。**
- **把元素放上去。**
- **然后就用这个Stream流简化的API来方便的操作元素。**

# 目录

Contents

**Stream操作集合或者数组的第一步是先得到Stream流，然后才能使用流的功能。**

## 集合获取Stream流的方式

- 可以使用Collection接口中的默认方法stream()生成流

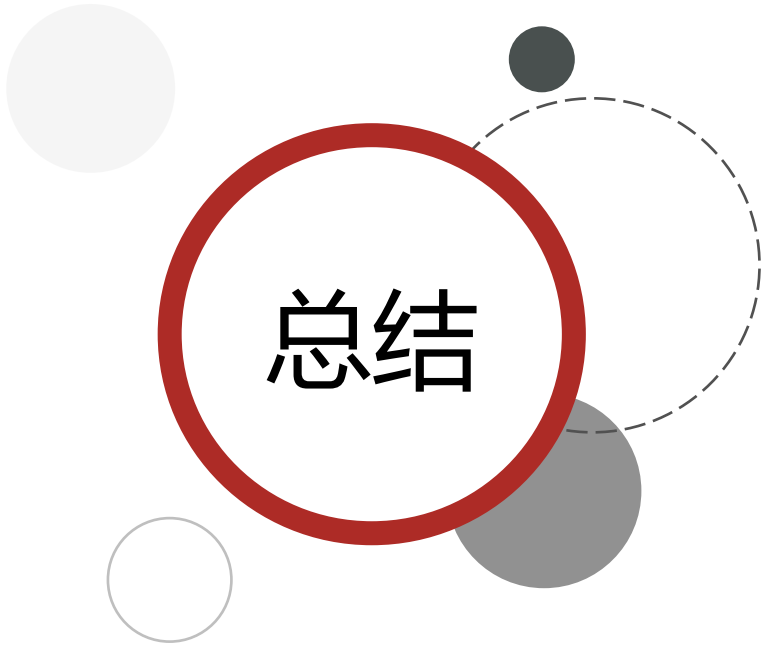
| 名称 | 说明 |
| --- | --- |
| default Stream<E> stream() | 获取当前集合对象的Stream流 |

## 数组获取Stream流的方式

| 名称 | 说明 |
| --- | --- |
| public static <T> Stream<T> stream(T[] array) | 获取当前数组的Stream流 |
| public static<T> Stream<T> of(T... values) | 获取当前数组/可变数据的Stream流 |

## Stream流的三类方法

- **获取Stream流**
  - 创建一条流水线，并把数据放到流水线上准备进行操作
- **中间方法**
  - 流水线上的操作。一次操作完毕之后，还可以继续进行其他操作。
- **终结方法**
  - 一个Stream流只能有一个终结方法，是流水线上的最后一个操作




1、集合获取Stream流的方式？

- **集合获取Stream的方式是通过调用stream()方法实现的。**

2、数组获取Stream流的方式？

| 名称 | 说明 |
| --- | --- |
| `public static <T> Stream<T> stream(T[] array)` | 获取当前数组的Stream流 |
| `public static<T> Stream<T> of(T... values)` | 获取当前数组/可变数据的Stream流 |

# 目录

Contents

## Stream流的常用API(中间操作方法)

| 名称 | 说明 |
| --- | --- |
| `Stream<T> filter(Predicate<? super T> predicate)` | 用于对流中的数据进行**过滤。** |
| `Stream<T> limit (long maxSize)` | 获取前几个元素 |
| `Stream<T> skip (long n)` | 跳过前几个元素 |
| `Stream<T> distinct ()` | 去除流中重复的元素。依赖(hashCode和equals方法) |
| `static <T> Stream<T> concat (Stream a, Stream b)` | **合并**a和b两个流为一个流 |

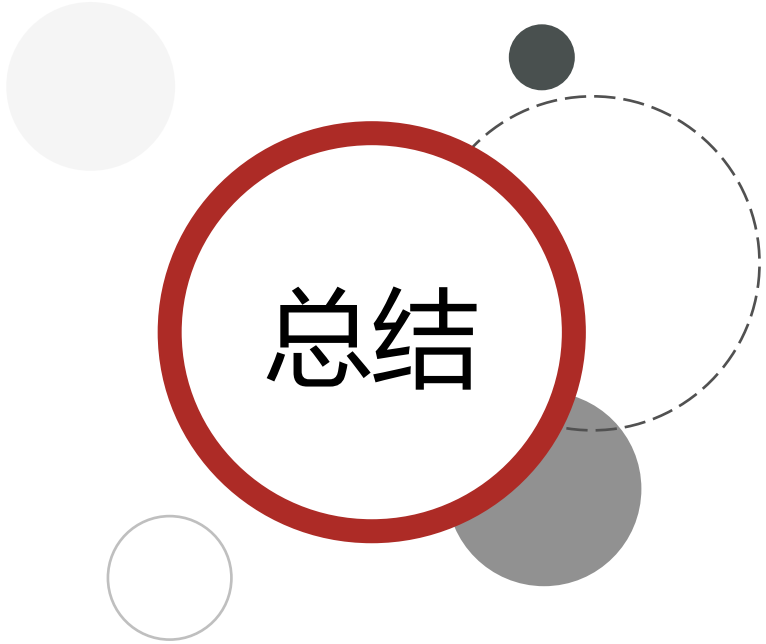
**注意：**

- 中间方法也称为非终结方法，调用完成后返回新的Stream流可以继续使用，支持链式编程。
- **在Stream流中无法直接修改集合、数组中的数据。**

## Stream流的常见终结操作方法

| 名称 | 说明 |
| --- | --- |
| `void forEach (Consumer action)` | 对此流的每个元素执行遍历操作 |
| `long count ()` | 返回此流中的元素数 |

**注意：终结操作方法，调用完成后流就无法继续使用了，原因是不会返回Stream了。**




1、终结和非终结方法的含义是什么？

- **终结方法后流不可以继续使用，非终结方法会返回新的流，支持链式编程。**

# 目录

Contents

# 案例 案例标题

需求：某个公司的开发部门，分为开发一部和二部，现在需要进行年中数据结算。

分析：

① ：员工信息至少包含了(名称、性别、工资、奖金、处罚记录)

② ：开发一部有4个员工、开发二部有5名员工

③ ：分别筛选出2个部门的最高工资的员工信息，封装成优秀员工对象Topperformer

④ ：分别统计出2个部门的平均月收入，要求去掉最高和最低工资。

⑤ ：统计2个开发部门整体的平均工资，去掉最低和最高工资的平均值。

# 目录

Contents

## Stream流的收集操作

- **收集Stream流的含义**：就是把Stream流操作后的结果数据转回到集合或者数组中去。

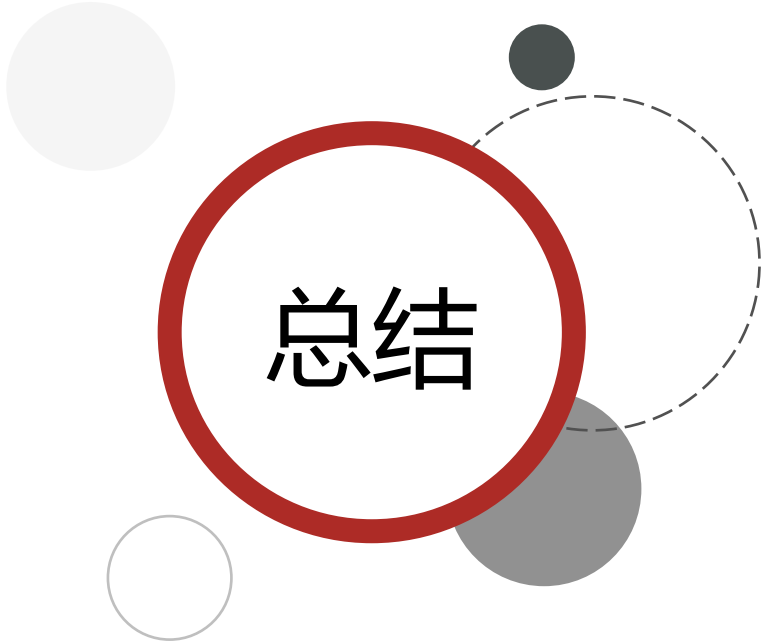
- Stream流：方便操作集合/数组的**手段。**
- 集合/数组：才是开发中的**目的。**

## Stream流的收集方法

| 名称 | 说明 |
| --- | --- |
| R collect (Collector collector) | 开始收集Stream流，指定收集器 |

## Collectors工具类提供了具体的收集方式

| 名称 | 说明 |
| --- | --- |
| public static <T> Collector toList () | 把元素收集到List集合中 |
| public static <T> Collector toSet () | 把元素收集到Set集合中 |
| public static  Collector toMap (Function keyMapper , Function valueMapper) | 把元素收集到Map集合中 |




1、收集Stream流的作用 ？

- **Stream流是操作集合/数组的手段**
- **操作的结果数据最终要恢复到集合或者数组中去。**

# 目录

Contents

- 创建不可变集合
- Stream流
- 异常处理
  - 异常概述、体系
  - 常见运行时异常
  - 常见编译时异常
  - 异常的默认处理流程
  - 编译时异常的处理机制
  - 运行时异常的处理机制
  - 异常处理使代码更稳健的案例
  - 自定义异常

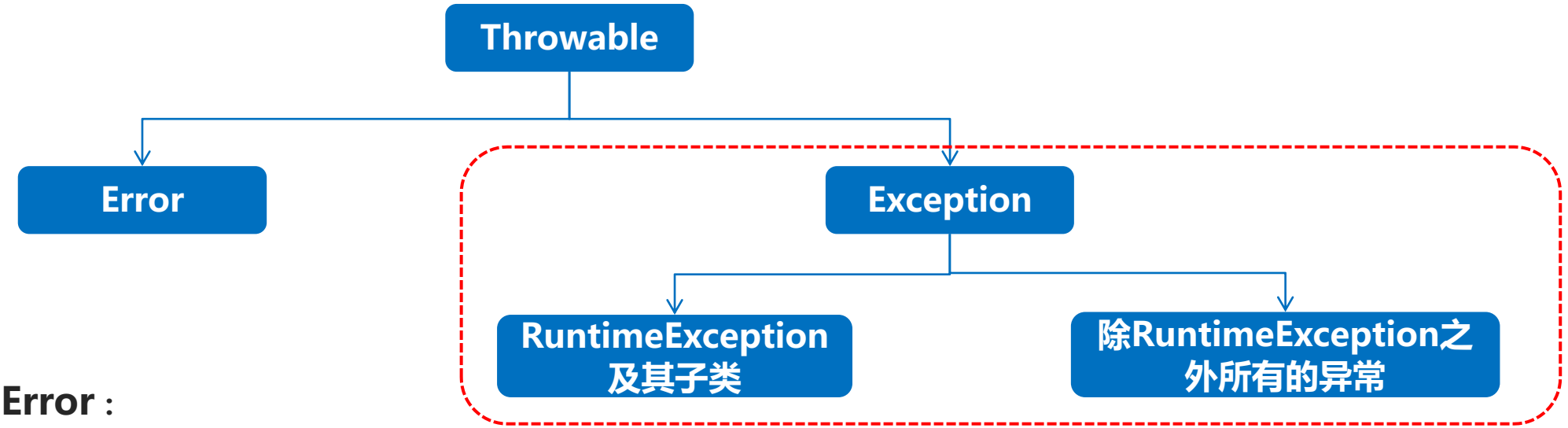
## 什么是异常？

- 异常是程序在“编译”或者“执行”的过程中可能出现的问题，**注意：**语法错误不算在异常体系中。
- 比如:数组索引越界、空指针异常、 日期格式化异常，等…

## 为什么要学习异常?

- 异常一旦出现了，如果没有提前处理，程序就会退出JVM虚拟机而终止.
- 研究异常并且避免异常，然后提前处理异常，体现的是程序的安全，健壮性。

## 异常体系



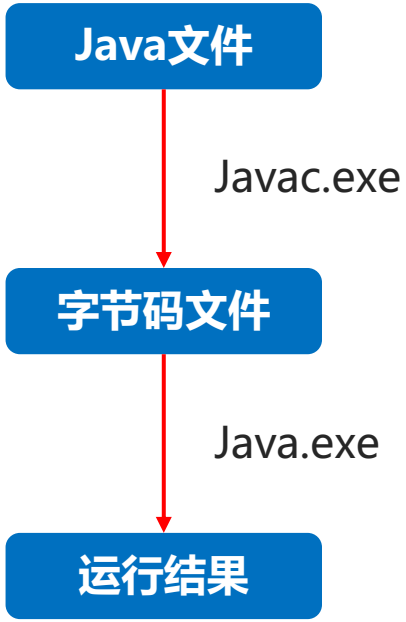

**Error**：

- 系统级别问题、JVM退出等，代码无法控制。

**Exception**：java.lang包下，称为异常类，它表示程序本身可以处理的问题

- **RuntimeException及其子类**：运行时异常，编译阶段不会报错。(空指针异常，数组索引越界异常)
- **除RuntimeException之外所有的异常**：编译时异常，编译期必须处理的，否则程序不能通过编译。(日期格式化异常)。

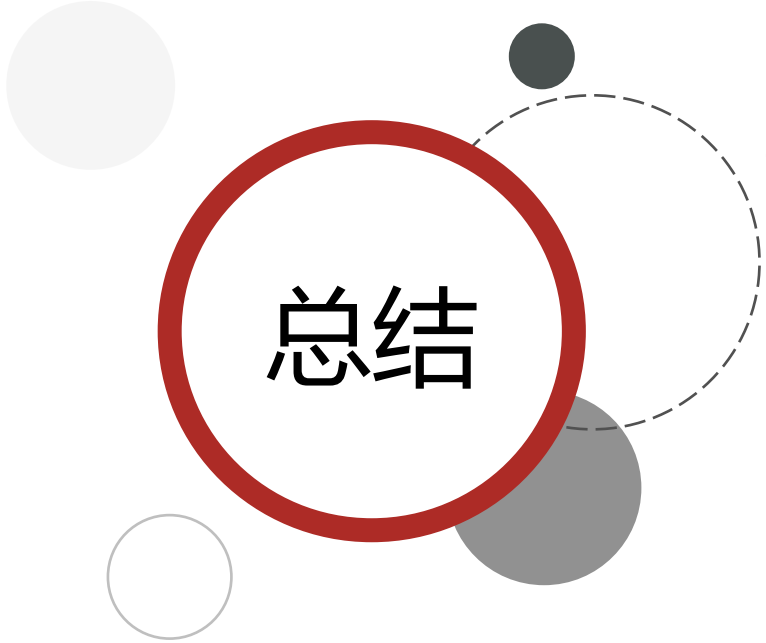
## 编译时异常和运行时异常




- 编译时异常，是在编译成class文件时必须要处理的异常，也称之为受检异常
- 运行时异常，在编译成class文件不需要处理，在运行字节码文件时可能出现的异常。

简单来说：

- 编译时异常就是在编译的时候出现的异常，
- 运行时异常就是在运行时出现的异常。




1. 异常是什么？
   - **异常是代码在编译或者执行的过程中可能出现的错误。**
2. 异常分为几类？
   - **编译时异常、运行时异常。**
   - **编译时异常：没有继承RuntimeExcpetion的异常，编译阶段就会出错。**
   - **运行时异常：继承自RuntimeException的异常或其子类，编译阶段不报错，运行可能报错。**
3. 学习异常的目的？
   - **避免异常的出现，同时处理可能出现的异常，让代码更稳健。**

# 目录

Contents

- **创建不可变集合**
- **Stream流**
- **异常处理**
  - 异常概述、体系
  - 常见运行时异常
  - 常见编译时异常
  - 异常的默认处理流程
  - 编译时异常的处理机制
  - 运行时异常的处理机制
  - 异常处理使代码更稳健的案例
  - 自定义异常

## 运行时异常

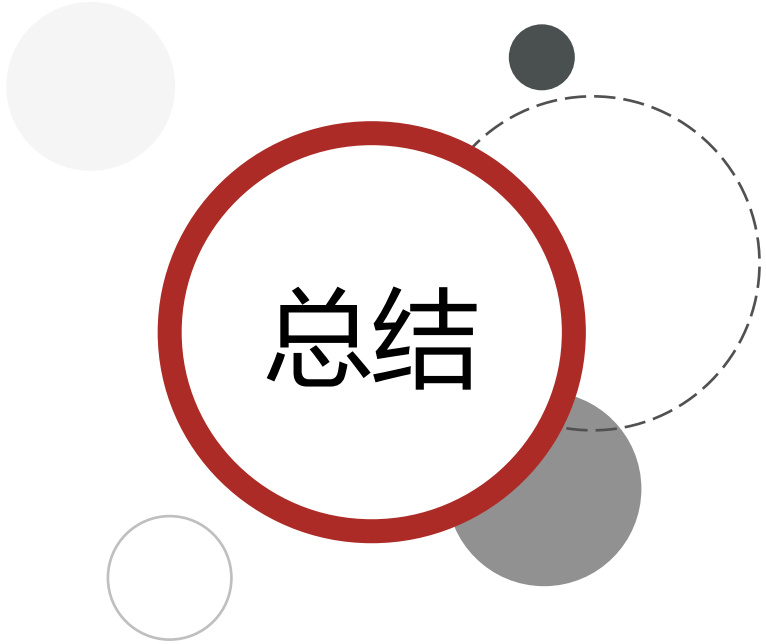
- **直接继承自RuntimeException或者其子类，**编译阶段不会报错，运行时可能出现的错误。

## 运行时异常示例

- 数组索引越界异常: ArrayIndexOutOfBoundsException
- 空指针异常：NullPointerException，直接输出没有问题，但是调用空指针的变量的功能就会报错。
- 数学操作异常：ArithmeticException
- 类型转换异常：ClassCastException
- 数字转换异常： NumberFormatException


**运行时异常：一般是程序员业务没有考虑好或者是编程逻辑不严谨引起的程序错误，**

**自己的水平有问题！**




1. 运行时异常的特点

- **运行时异常：继承自RuntimeException的异常或者其子类，**
- **编译阶段不报错,运行可能报错。**

# 目录

Contents

- 创建不可变集合
- Stream流
- 异常处理
  - 异常概述、体系
  - 常见运行时异常
  - 常见编译时异常
  - 异常的默认处理流程
  - 编译时异常的处理机制
  - 运行时异常的处理机制
  - 异常处理使代码更稳健的案例
  - 自定义异常

## 编译时异常

- **不是RuntimeException或者其子类的异常，**编译阶就报错，必须处理，否则代码不通过。

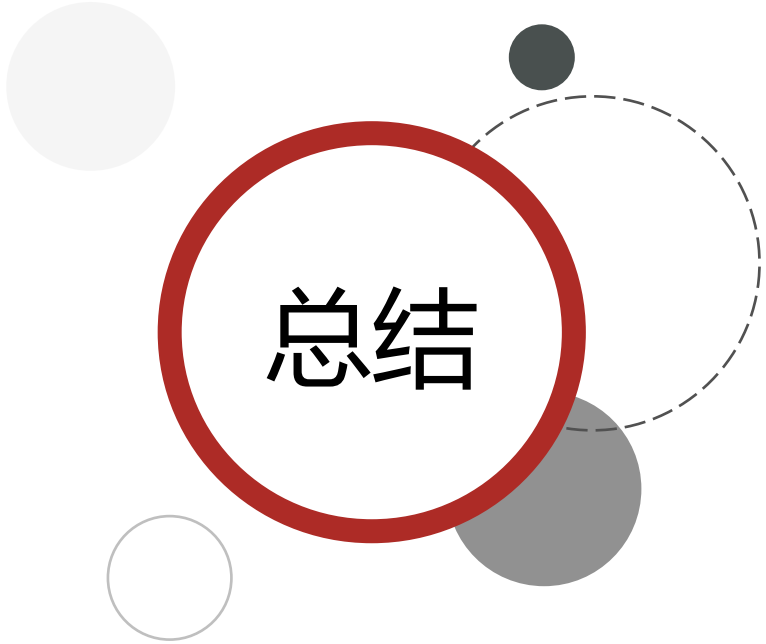
## 编译时异常示例

```
String date = "2015-01-12 10:23:21";
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
Date d = sdf.parse(date);
System.out.println(d);
```

**日期解析异常：ParseException**

## 编译时异常的作用是什么：

- 是担心程序员的技术不行，在编译阶段就爆出一个错误, 目的在于提醒不要出错!
- 编译时异常是可遇不可求。遇到了就遇到了呗。




1. 编译时异常的特点

- **编译时异常：继承自Exception的异常或者其子类**
- **编译阶段报错，必须处理，否则代码不通过。**

# 目录

Contents

- **创建不可变集合**
- **Stream流**
- **异常处理**
  - 异常概述、体系
  - 常见运行时异常
  - 常见编译时异常
  - 异常的默认处理流程
  - 编译时异常的处理机制
  - 运行时异常的处理机制
  - 异常处理使代码更稳健的案例
  - 自定义异常

① 默认会在出现异常的代码那里自动的创建一个异常对象：ArithmeticException。

② 异常会从方法中出现的点这里抛出给调用者，调用者最终抛出给JVM虚拟机。

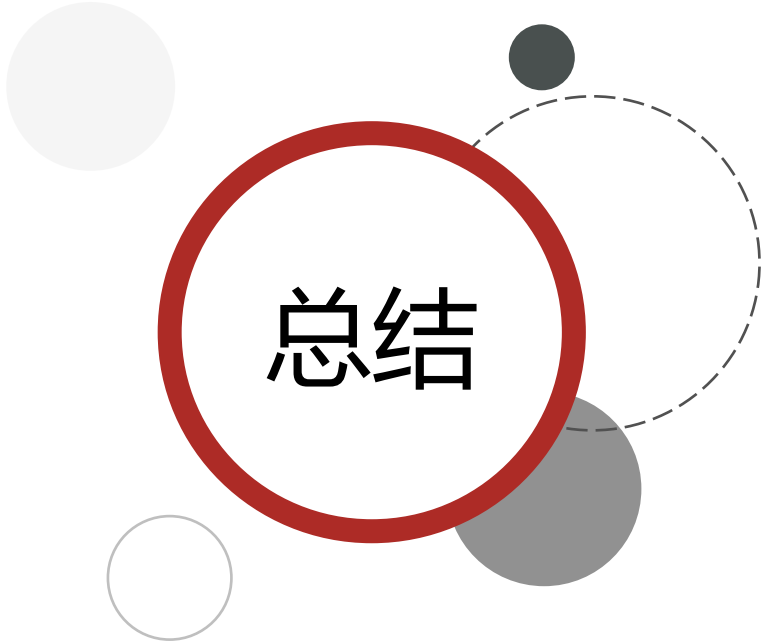
③ 虚拟机接收到异常对象后，先在控制台直接输出异常栈信息数据。

④ 直接从当前执行的异常点干掉当前程序。

⑤ 后续代码没有机会执行了，因为程序已经死亡。

```
public class ExceptionDemo {
    public static void main(String[] args) {
        System.out.println("程序开始。。。。。。。。。");
        chu( a: 10 , b: 0);
        System.out.println("程序结束。。。。。。。。。");
    }

    public static void chu(int a , int b){
        System.out.println(a);
        System.out.println(b);
        int c = a / b ;// 出现了运行时异常,自动创建异常对象：ArithmeticException
        System.out.println("结果是："+c);
    }
}
```




1. 默认异常处理机制。
   - **默认的异常处理机制并不好，一旦真的出现异常，程序立即死亡！**

# 目录

Contents

- 创建不可变集合
- Stream流
- 异常处理
  - 异常概述、体系
  - 常见运行时异常
  - 常见编译时异常
  - 异常的默认处理流程
  - 编译时异常的处理机制
  - 运行时异常的处理机制
  - 异常处理使代码更稳健的案例
  - 自定义异常

编译时异常是编译阶段就出错的，所以必须处理，否则代码根本无法通过

**编译时异常的处理形式有三种：**

- 出现异常直接抛出去给调用者，调用者也继续抛出去。
- 出现异常自己捕获处理，不麻烦别人。
- 前两者结合，出现异常直接抛出去给调用者，调用者捕获处理。

## 异常处理方式1 —— throws

- throws：用在方法上，可以将方法内部出现的异常抛出去给本方法的调用者处理。
- 这种方式并不好，发生异常的方法自己不处理异常，如果异常最终抛出去给虚拟机将引起程序死亡。

**抛出异常格式：**

```
方法 throws 异常1 ，异常2 ，异常3 ..{
}
```

**规范做法**：

```
方法 throws Exception{
}
```

- 代表可以抛出一切异常，

## 异常处理方式2 —— try...catch...

- 监视捕获异常，用在方法内部，可以将方法内部出现的异常直接捕获处理。
- 这种方式还可以，发生异常的方法自己独立完成异常的处理，程序可以继续往下执行。

**格式：**

```
try{
    // 监视可能出现异常的代码！
}catch(异常类型1 变量){
    // 处理异常
}catch(异常类型2 变量){
    // 处理异常
}...
```

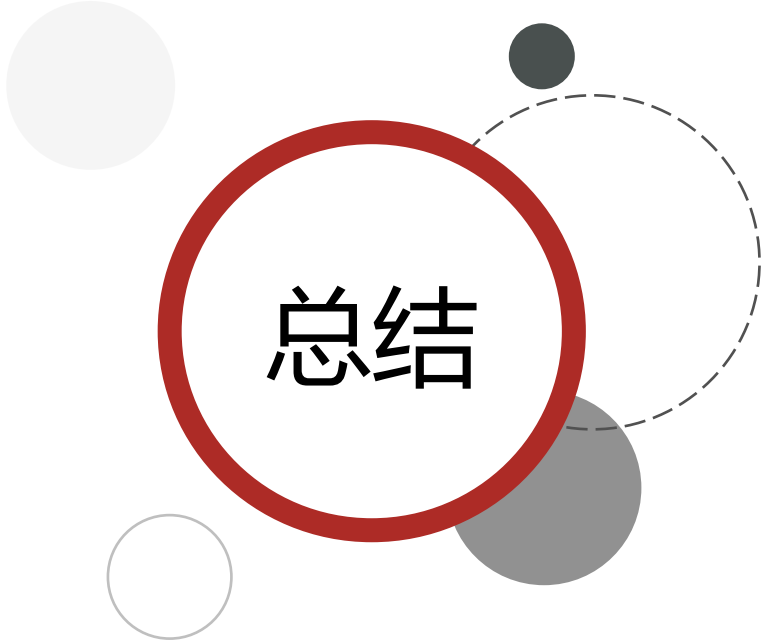
**建议格式：**

```
try{
    // 可能出现异常的代码！
}catch (Exception e){
    e.printStackTrace(); // 直接打印异常栈信息
}

Exception可以捕获处理一切异常类型！
```

## 异常处理方式3 —— 前两者结合

- 方法直接将异通过throws抛出去给调用者
- 调用者收到异常后直接捕获处理。




## 1、异常处理的总结

- **在开发中按照规范来说第三种方式是最好的：底层的异常抛出去给最外层，最外层集中捕获处理。**

- **实际应用中，只要代码能够编译通过，并且功能能完成，那么每一种异常处理方式似乎也都是可以的。**

# 目录

Contents

- **创建不可变集合**
- **Stream流**
- **异常处理**
  - 异常概述、体系
  - 常见运行时异常
  - 常见编译时异常
  - 异常的默认处理流程
  - 编译时异常的处理机制
  - 运行时异常的处理机制
  - 异常处理使代码更稳健的案例
  - 自定义异常

## 运行时异常的处理形式

- 运行时异常编译阶段不会出错，是运行时才可能出错的，所以编译阶段不处理也可以。
- 按照规范建议还是处理：建议在最外层调用处集中捕获处理即可。

# 目录

Contents

- **创建不可变集合**
- **Stream流**
- **异常处理**
  - 异常概述、体系
  - 常见运行时异常
  - 常见编译时异常
  - 异常的默认处理流程
  - 编译时异常的处理机制
  - 运行时异常的处理机制
  - 异常处理使代码更稳健的案例
  - 自定义异常

## 案例 异常处理使代码更稳健的案例

**需求**

- 键盘录入一个合理的价格为止（必须是数值）。

**分析**

- 定义一个死循环，让用户不断的输入价格。

# 目录

Contents

## 自定义异常的必要？

- Java无法为这个世界上全部的问题提供异常类。
- 如果企业想通过异常的方式来管理自己的某个业务问题，就需要自定义异常类了。

## 自定义异常的好处：

- 可以使用异常的机制管理业务问题，如提醒程序员注意。
- 同时一旦出现bug，可以用异常的形式清晰的指出出错的地方。

## 自定义异常的分类

### 1、自定义编译时异常

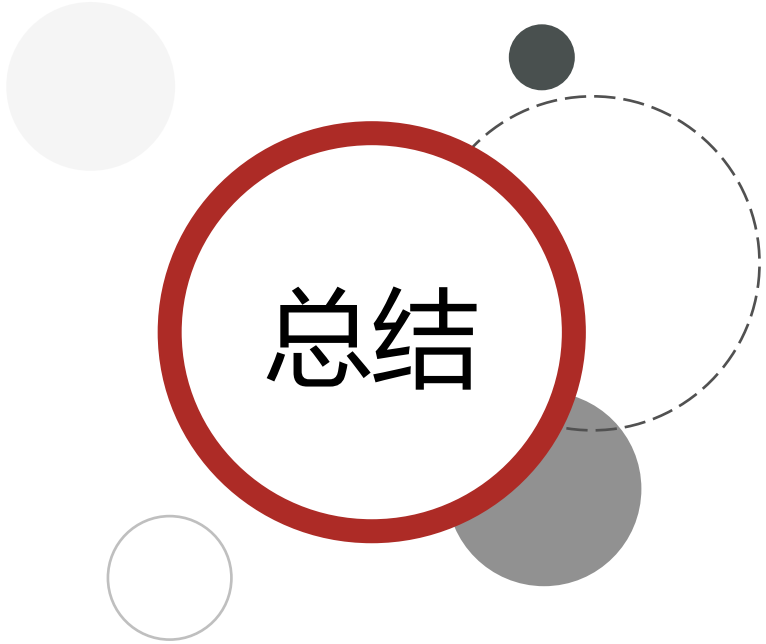
- 定义一个异常类继承Exception.
- 重写构造器。
- 在出现异常的地方用throw new 自定义对象抛出，

作用：编译时异常是编译阶段就报错，提醒更加强烈，一定需要处理！！

### 2、自定义运行时异常

- 定义一个异常类继承RuntimeException.
- 重写构造器。
- 在出现异常的地方用throw new 自定义对象抛出!

作用：提醒不强烈，编译阶段不报错！！运行时才可能出现！！




**1、自定义编译时异常**

- 定义一个异常类继承Exception.
- 重写构造器。
- 在出现异常的地方用throw new 自定义对象抛出，

作用：编译时异常是编译阶段就报错，提醒更加强烈，一定需要处理！！

**2、自定义运行时异常**

- 定义一个异常类继承RuntimeException.
- 重写构造器。
- 在出现异常的地方用throw new 自定义对象抛出!

作用：提醒不强烈，编译阶段不报错！！运行时才可能出现！！


传智教育旗下高端IT教育品牌