

# 面向对象进阶

泛型、异常

# 目录

Contents

## ➤ 泛型

- ◆ 泛型类
- ◆ 泛型接口
- ◆ 泛型方法

## ➤ 异常类



## 泛型

- 定义类、接口、方法时，同时声明了一个或者多个类型变量（如：`<E>`），称为泛型类、泛型接口，泛型方法、它们统称为泛型。

```
public class ArrayList<E>{  
    ...  
}
```

- 作用：泛型提供了在编译阶段约束所能操作的数据类型，并自动进行检查的能力！这样可以避免强制类型转换，及其可能出现的异常。
- 泛型的本质：把具体的数据类型作为参数传给**类型变量**。

## 泛型

Java的泛型类是一种允许在编译时提供**类型安全**的方式，它允许开发者定义类、接口和方法时不指定具体的类型，而是使用一个**类型参数**来代替。

这样，当创建类的实例或者调用方法时，可以指定具体的类型，从而提高代码的复用性和安全性。



## 问题的引入

在Java中，如果我们需要定义一个可以存储不同数据类型的**类**，  
例如坐标点类，它可能需要存储整型、浮点型或字符串类型，我们可能会首先想到使用  
Object类作为存储的类型，因为Object是所有Java类的根类。



```
package L10;

public class Point {
    private Object x;
    private Object y;

    public Point(Object x, Object y) {
        this.x = x;
        this.y = y;
    }
    public Object getX() { return x; }
    public Object getY() { return y; }
    public void setX(Object x) { this.x = x; }
    public void setY(Object y) { this.y = y; }
}
```



```
public class PointTest {  
    public static void main(String[] args) {  
  
        Point point1 = new Point(x: 10, y: 20);  
        System.out.println("x坐标: "+point1.getX());  
        System.out.println("y坐标: "+point1.getY());  
  
        Point point2 = new Point(x: 1.2, y: 2.3);  
        System.out.println("x坐标: "+point2.getX());  
        System.out.println("y坐标: "+point2.getY());  
  
        Point point3 = new Point(x: "a", y: "b");  
        System.out.println("x坐标: "+point3.getX());  
        System.out.println("y坐标: "+point3.getY());  
    }  
}
```



## 泛型

- 用Object来定义坐标类型：就可以代入**不同类型数据**了。
- 虽然这样做满足了需求，不过却隐含了一个不安全因素，为什么说是隐含呢？
  - 此时的数据还是Object型，也就是所有类型的父类，你必须**清醒的明白自己**传入的是什么类型，然后将其做向下转型处理（拆箱）才能使用
  - 例如：Point p=new Point(12.23,23.21);  
  
Double x = (Double)p.getX() ;
  - 又例如：Point p=new Point("北纬29度","东经113度");  
  
Double x = (**Double**)p.getX() ;
    - 这时编译会通过，但是一旦运行则会发生**类型转换异常**。



```
1 package L10;  
2  
3 public class PointTest {  
4     public static void main(String[] args) {  
5         // 这里传入了一个整数和一个字符串  
6         Point point = new Point(x: 10, y: "a");  
7         // 这里将尝试将getX的返回值转换为Integer  
8         int x = (Integer) point.getX();  
9         // 这里将尝试将getY的返回值转换为Integer  
10        int y = (Integer) point.getY();  
11    }  
12 }  
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:  
Exception in thread "main" java.lang.ClassCastException  
at L10.PointTest.main(PointTest.java:10)
```

class java.lang.String cannot be cast to class java.lang.Integer



## 问题的引入

因为Object是所有类的根类，所以任何类型的对象都可以被赋值给一个Object类型的变量。这使得Object看起来是一个非常通用的类型，可以用来存储任何类型的数据。

```
Object obj1 = 123; // 存储整型  
Object obj2 = 45.67; // 存储浮点型  
Object obj3 = "Hello"; // 存储字符串
```

Object类型可以接受任何类型的对象，看起来非常灵活。但是，使用Object类会**丧失类型检查**，导致在运行时出现ClassCastException。在这个例子中，obj变量存储了一个字符串，但在尝试将其转换为Integer类型时，程序会在运行时抛出ClassCastException。

```
Object obj = "Hello"; // 存储字符串  
String str = (String) obj; // 正确的类型转换  
Integer num = (Integer) obj; // 错误的类型转换，抛出ClassCastException
```



## 泛型类

```
修饰符 class 类名<类型参数, 类型参数, …> {  
}  
}
```

```
public class A<E>{  
    ...  
}
```

- 注意：类型参数建议用大写的英文字母，常用的有：**E、T、K、V** 等

Java 中泛型标记符：

- E - Element (在集合中使用，因为集合中存放的是元素)
- T - Type (Java 类)
- K - Key (键)
- V - Value (值)
- N - Number (数值类型)



## 1. 泛型类定义

“class 名称<类型参数>” 定义一个泛型类，为了与普通类区别，后面加一个“<>”，其内部是类型参数，也称泛型参数，多个参数之间用逗号分隔。下面是泛型类声明的例子。

```
class OneClass<E>;           // OneClass是泛型类，E是类型参数。  
class TwoClass<E1, E2>;     // TwoClass是泛型类，E1、E2是类型参数。
```

## 2. 泛型类的应用

使用泛型类声明对象时，必须要指明类中使用类型参数的实际类型。

```
Double objD = 10d;  
String objS = "Generic";  
OneClass<Double> obj1 = new OneClass(objD); // 创建泛型对象  
OneClass<String> obj2 = new OneClass(objS); // 创建泛型对象
```



## 泛型类

为了达到不存在安全隐患和代入各种数据类型的目的，看看如何用泛型改写上面的代码

使用泛型过后，可减少安全隐患的存在。如果此时我们刻意传入不一样的数据类型，在编译时就会报错

```
public class Point<T> {  
    private T x;  
    private T y;  
  
    public Point(T x, T y) {  
        this.x = x;  
        this.y = y;  
    }  
    public T getX() { return x; }  
    public T getY() { return y; }  
    public void setX(T x) { this.x = x; }  
    public void setY(T y) { this.y = y; }  
}
```

```
public class PointTest {  
    public static void main(String[] args) {  
        // 使用Integer类型来创建Point对象，可以存储整型数据  
        Point<Integer> pointInt = new Point<>(x: 10, y: 20);  
    }  
    Point<Integer> pointInt = new Point<>(x: 10, y: "a");  
}
```



## 泛型接口

```
修饰符 interface 接口名<类型参数, 类型参数, …> {  
}  
}
```

```
public interface A<E>{  
    ...  
}
```

- 注意：类型参数建议用大写的英文字母，常用的有：**E、T、K、V** 等



## 泛型接口

- 实现泛型接口的方式有两种，**一种是在实现的时候指定泛型类型**
- 另一种是依然使用泛型，在创建对象构造的时候确定泛型类型**

```
public interface Printer<T> {  
    void print(T item);  
}
```

定义一个泛型接口

```
public class StringPrinter implements Printer<String> {  
    @Override //StringPrinter类专门用于打印字符串  
    public void print(String item) { System.out.println(item); }  
}
```

方式一：具体类实现泛型接口

```
public class GenericPrinter<T> implements Printer<T> {  
    @Override //GenericPrinter类是一个泛型类，  
    //它可以打印任何类型的数据。  
    public void print(T item) {System.out.println(item); }  
}
```

方式二：泛型类实现泛型接口

```
public class PrinterTest {  
    public static void main(String[] args) {  
        Printer<String> printer1 = new StringPrinter();  
        printer1.print("Hello, World!");  
  
        //创建对象构造的时候确定泛型类型  
        Printer<Integer> printer2 = new GenericPrinter<>();  
        printer2.print(42);  
    }  
}
```

使用示例



## 泛型方法

- 泛型方法允许在方法的返回值类型前面定义一个或多个类型参数。
- 使方法能够接受任何类型的参数并返回任何类型的结果，具体的类型将在调用方法时确定。

```
修饰符 <类型变量, 类型变量, …> 返回值类型 方法名(形参列表) {  
}
```

```
public static <T> void test(T t){  
}
```



## 泛型方法

```
public class SimpleDisplay {  
    // 泛型方法，接受任意类型的对象并打印其信息  
    public static <T> void display(T obj) {  
        System.out.println("显示对象: " + obj);  
    }  
}
```

```
public class TestSimpleDisplay {  
    public static void main(String[] args) {  
        // 使用SimpleDisplay类的泛型方法display打印字符串  
        SimpleDisplay.<String>display(obj: "Hi");  
        // 通过在方法调用时指定类型参数（如 <String> 和 <Integer>）  
        // 使用SimpleDisplay类的泛型方法display打印整数  
        SimpleDisplay.<Integer>display(obj: 2024);  
        // 告诉编译器方法应该接受的确切类型  
        SimpleDisplay.<Integer>display(obj: "2024");  
    }  
}
```

使用示例





## 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
- 比如：数组索引越界、空指针异常、日期格式化异常，等…

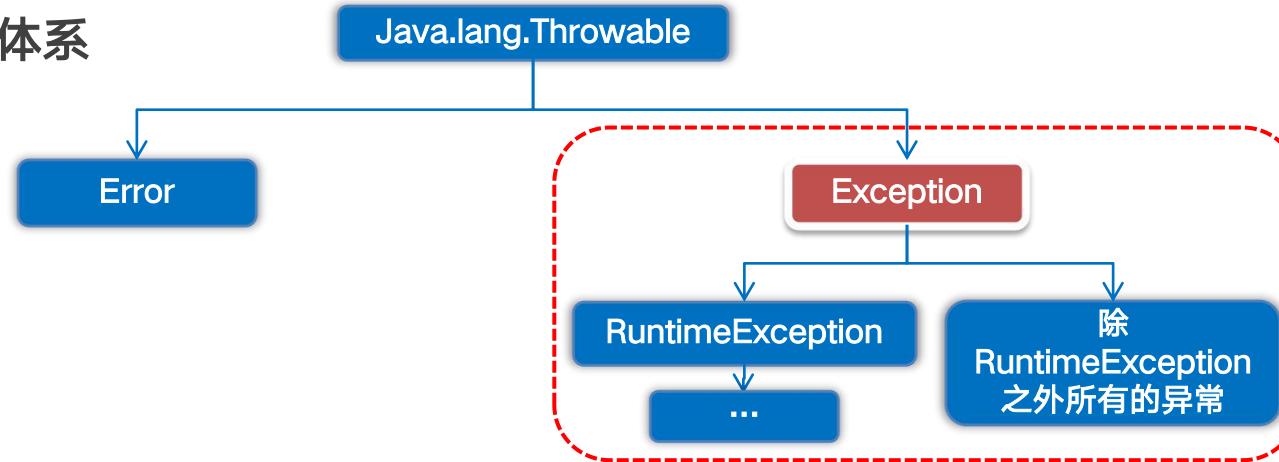
```
int[] arr = {10, 20, 30};  
  
System.out.println(arr[3]);
```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:E:\Program  
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException  
at Test1.Test1.main(Test1.java:5)
```

```
System.out.println(10 / 0);
```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:E:\Program Files\JetBrains\  
Exception in thread "main" java.lang.ArithmetricException Create breakpoint : / by zero  
at Test1.Test1.main(Test1.java:6)
```

## 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)。



## 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。

`java.lang`

### 类 `RuntimeException`

`java.lang.Object`

└ `java.lang.Throwable`

  └ `java.lang.Exception`

    └ `java.lang.RuntimeException`

所有已实现的接口：

`Serializable`

直接已知子类：

`AnnotationTypeMismatchException`, `ArithmaticException`,  
`ArrayStoreException`, `BufferOverflowException`, `BufferUnderflowException`,  
`CannotRedoException`, `CannotUndoException`, `ClassCastException`,  
`CMMException`, `ConcurrentModificationException`, `DOMException`,  
`EmptyStackException`, `EnumConstantNotPresentException`, `EventException`,  
`IllegalArgumentExcepcion`, `IllegalMonitorStateException`,  
`IllegalPathStateException`, `IllegalStateException`, `ImagingOpException`,  
`IncompleteAnnotationException`, `IndexOutOfBoundsException`,  
`JMRuntimeException`, `LSException`, `MalformedParameterizedTypeException`,



## 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



## 抛出异常 (throws)

- 在方法上使用throws关键字，可以将方法内部出现的异常抛出去给调用者处理。

```
方法 throws 异常1 , 异常2 , 异常3 ...{  
    ...  
}
```

## 捕获异常(try...catch)

- 直接捕获程序出现的异常。

```
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  
}...
```



## 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。
- 写在方法定义处，表示声明一个异常

抛出异常格式：

```
public void 方法() throws 异常类名1 , 异常类名2...{  
}
```

```
public static int divide() throws ArithmeticException {  
    .....  
}
```



## 异常处理方式1 —— throw

- **throw**: 当代码执行到某个条件下无法继续正常执行时，可以使用 **throw** 关键字抛出异常，以告知调用者当前代码的执行状态。
- **写在方法内，结束方法。方法中下面的代码不再执行**

```
public void 方法(){  
    throw new 异常类构造方法();  
}
```

```
public void checkNumber(int num) {  
    if (num < 0) {  
        throw new IllegalArgumentException("必须为正数");  
    }  
}
```

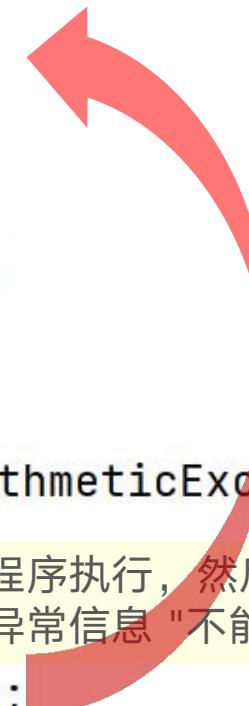


```
public class Test {  
    public static void main(String[] args) {  
        try {  
            System.out.println(divide(a: 4, b: 0));  
        } catch (ArithmetricException e) {  
            System.out.println(e.toString());  
            System.out.println("算术异常：除数不能为0");  
        }  
    }  
  
    public static int divide(int a, int b) throws ArithmetricException {  
        // 可能会抛出除以零异常  
        if (b == 0) {  
            throw new ArithmetricException("不能除以零");  
        } else {  
            ;  
        }  
    }  
}
```

java.lang.ArithmetricException是异常类的全名，不能除以零则是在抛出异常时传递的消息字符串。

这个异常被抛出时，会中断当前的程序执行，然后返回到调用这个方法的位置，并带着一个异常信息“不能除以零”。

"C:\Program Files\Java\jdk-21\bin\java.exe  
java.lang.ArithmetricException: 不能除以零  
算术异常：除数不能为0





## 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2 —— try...catch...

格式：

```
public class TryCatchDemo {  
    public static void main(String[] args) {  
        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  
            }...  
        }  
    }  
}
```

尝试访问数组 myNumbers 的第 10 个元素，但是这个数组的大小只有3，因此会出现 `ArrayIndexOutOfBoundsException` 异常，这个异常会被 `catch` 语句块捕获，然后输出"数组越界了"



## 异常处理方式2 —— try...catch...

```
public class TryCatchDemo {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            //这里不会出现数组越界的异常  
            System.out.println(myNumbers[0]);  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("数组越界了");  
        }  
    }  
}
```

如果try中没有遇到问题，会把try里面的代码全部执行，不会执行catch里面的代码

"C:\Program Files\Java\j



## 异常处理方式2 —— try...catch...

```
public class TryCatchDemo {  
    public static void main(String[] args) {  
        try {  
            int result = 10 / 0; //数学操作异常: ArithmeticException  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("数组越界了");  
        }  
    }  
}
```

如果try中遇到的问题没有被捕获，当前异常会交给虚拟机处理

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:E:\Program Files\JetBrains  
Exception in thread "main" java.lang.ArithemticException Create breakpoint : / by zero  
at L10.exception.TryCatchDemo.main(TryCatchDemo.java:6)
```



```
public class MultiCatchDemo {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            // 这会抛出 ArrayIndexOutOfBoundsException  
            System.out.println(myNumbers[10]);  
            // 这会抛出 ArithmeticException  
            int result = 10 / 0;  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("处理数组越界异常");  
        } catch (ArithmetricException e) {  
            System.out.println("处理算术异常：除数不能为0");  
        } catch (Exception e) {  
            System.out.println("处理未知异常");  
        }  
    }  
}
```

"C:\Program Files\Java\jdk-21\bin"

处理数组越界异常

Process finished with exit code 0

在一个 try 块中，一旦发生了第一个异常，try 块里面那个异常之后的代码就不会被执行了。

这样做是有道理的，因为异常表示了出现了一个问题，如果程序继续执行下去，可能会增加出现更严重错误的风险。



```
public class MultiCatchDemo {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            // 这会抛出 ArrayIndexOutOfBoundsException  
            System.out.println(myNumbers[10]);  
            // 这会抛出 ArithmeticException  
            int result = 10 / 0;  
        } catch (ArithmetricException e) {  
            System.out.println("处理算术异常：除数不能为0");  
        } catch (Exception e) {  
            System.out.println("处理未知异常");  
        }  
    }  
}
```

"C:\Program File:

处理未知异常

当异常被抛出时，Java虚拟机会从上到下查找对应的 catch 块。一旦找到一个与抛出的异常类型匹配的 catch 块，就由这个块来处理异常，并且其他的 catch 块不会被查找和执行。

因此，需要把捕捉更具体异常类型的 catch 块放在前面，捕捉更广泛异常类型（如 Exception 类型）的 catch 块放在最后。



## try-catch语句

- ✓ try 块控制其包含的语句，定义与其关联的异常处理程序的范围。
- ✓ try 块的后面可以有多个 catch 块，每一个 catch语句都处理一个不同的异 常类型。
- ✓ 如果一个 try 块有多重 catch语句，那么 catch语句的顺序很重要。这 是因 为从 try 块引发的异常由第一个能够处理该异常的 catch语句来 处理。
- ✓ catch 语句采用异常类的对象引用作为参数。
- ✓ 一旦异常捕获，执行 catch 块内的语句。
- ✓ catch 块的范围仅限制于在上述 try 块中的语句。



## finally子句

finally 语句定义一个总是被执行的代码块，而不考虑是否出现异常。不是必须的子句。

```
public class FinallyTester{
    static int method1(int x) throws Exception
    {
        if(x<0)throw new Exception("x<0");
        return x++;
    }
    public static void main(String args[]){
        try {
            System.out.println(method1(-1));
            System.out.println("end");
        } catch(Exception e)
        {
            System.out.println("Wrong");
        } finally {
            System.out.println("Finally");}
    }
}
```



## 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. 异常的代表是谁？分为几类？

- Exception，分为两类：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，运行时出现的。

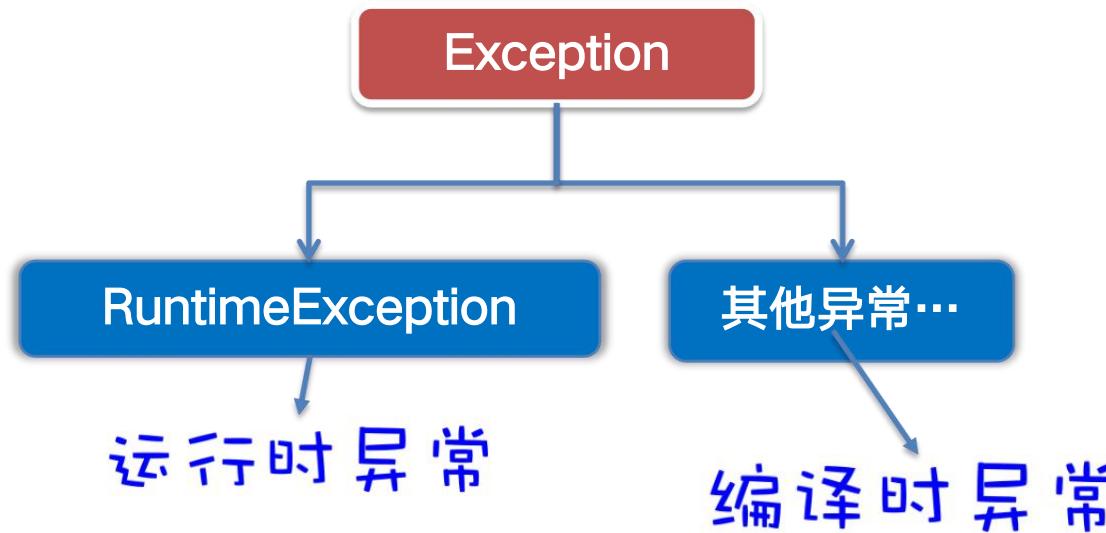
## 3. 异常代码层面的处理有几种方式？

- 用来查找bug；可以作为方法内部的特殊返回值，通知上层调用者底层的执行情况。

总结

## 自定义异常

- Java无法为这个世界上全部的问题都提供异常类来代表，如果自己的某种问题，想通过异常来表示，以便用异常来管理该问题，那就需要自己来定义异常类了。





## 自定义异常

- 定义一个异常类
- 写继承关系
- 重写构造方法。
- 在出现异常的地方用throw new 自定义对象抛出，

```
// 定义年龄异常类
public class AgeException extends RuntimeException {
    public AgeException(String message) {
        super(message);
    }
}
```



## 自定义异常

- 定义一个异常类
- 写继承关系
- 重写构造器。
- 在出现异常的地方用throw

new 自定义对象抛出，

```
public class Student {  
    private int age;  
    // 构造方法  
    public Student() {  
    }  
    public Student(int age) {  
        if(age < 16 || age > 30) {  
            throw new AgeException("年龄异常！");  
        }  
        this.age = age;  
    }  
  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) throws AgeException {  
        if(age < 16 || age > 30) {  
            throw new AgeException("年龄异常！");  
        }  
        this.age = age;  
    }  
}
```



## 自定义异常

- 定义一个异常类
- 写继承关系
- 重写构造器。
- 在出现异常的地方用throw

new 自定义对象抛出，

```
public class Test {  
    public static void main(String[] args) {  
        try {  
            Student s1 = new Student(10);  
        } catch (AgeException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Pro  
L10.exception.student.AgeException: 年龄异常!  
      at L10.exception.student.Student.<init>(Student.java:10)  
      at L10.exception.student.Test.main(Test.java:7)
```

Exception 类提供了如下方法：

toString()

getMessage(): 返回异常的消息信息。

printStackTrace(): 将对象的堆栈跟踪输出至错误输出流，作为System.err 的值。