目 录

第一章　JavaSE

Java 官方文档

快捷键：

1.ctrl+ 鼠标左键：把光标放在 `String` 上，打开对应的源码

1.1　环境

1.1.1　基础概念

- JDK：Java 开发工具包，开发 Java 代码必备
- JRE：Java 运行时环境，运行 Java 程序必备
- JVM：Java 虚拟机

三者的关系：JDK 中包含了 JRE，JRE 中又有 JVM。

1.1.2　编译运行 Java 程序（命令行）

- javac: 编译，把 `.java` 文件变成 `.class` 二进制字节码文件
- java: 让 JVM 解释执行字节码文件

1.1.3　集成开发环境

使用 IDEA 集成开发环境进行 Java 程序的开发

1.2　变量和类型

1.2.1　变量

变量：表示程序运行时可以改变的量，创建一个变量，就会为其分配一定的内存空间

1.2.2　常量

- 字面值常量：例如 $1, 1.0, 1.0f, "hello", true, false$ 等。
- `final` 关键字修饰的量，在程序运行中无法修改其值。

1.2.3　内置类型

- 整数：`byte`, `short`, `int`, `long`
- 浮点数：`float`, `double`，不能使用 = 直接判断两个浮点数是否相等。
- 字符：`char`，Java 中的 char 占两个字节，使用 unicode 编码。
- 布尔：`boolean`

内置类型都有对应的包装类来描述他们。

1.2.4 引用类型

- 字符串 `String`
- 数组
- 类

引用相当于是一个低配指针，没有指针加减的一些功能。

1.2.5 类型转换

隐式类型转换：把一个表示范围小的类型赋值给表示范围大的类型

强制类型转换：用加括号的方式来进行强制类型转换。要注意部分类型之间无法进行强制类型转换，如布尔和整型，如果强制转换，则会出错。

类型提升：把不同类型的数值放一起进行计算，会发生类型提升，`byte`、`short` 类型进行计算时，通常先提升为 `int` 类型进行计算。

1.2.6 变量的作用域

局部变量：作用域为当前代码块。

成员变量：取决于**访问权限控制**字符。`public`、`private`、`protected`、`default`（就是空白）

1.2.7 变量的命名规范

1. 必须由数字、字母、下划线、$ 组成。
2. 变量名尽量有意义，采用驼峰命名法。（变量名小驼峰，类名大驼峰）

1.3 运算符

1. 算数运算符：`+`、`-`、`*`、`/`、`%`。
2. 关系运算符：`<`、`<=`、`==`、`>`、`>=`、`!=`，关系运算符的返回值一定是 `boolean` 类型。
3. 逻辑运算：`&&`、`||`、`!`，逻辑运算符支持短路求值。

`&` 和 `|` 的操作数为 `boolean` 类型，也表示逻辑运算，但不支持短路求值。

4. 位运算：&、| 操作数是整数类型，~ 表示取反，^ 表示异或。
5. 移位运算：`<<`、`>>`（算术右移，左侧补符号位）、`>>>`（逻辑右移，左侧补 0）。左移 1 位，相当于乘 2，算数右移 1 位，相当于除 2.
6. 条件运算：`? :`
7. 运算符的优先级：不知道优先级时加括号。

1.4 注释

3 种。

1.5 关键字

1.6 分支和循环语句

1.6.1 分支语句

1. `if`
2. `switch`

1.6.2 循环语句

1. `while` 语句
2. `for` 语句
3. `do...while` 语句: 至少执行一次
4. `for...each`
5. `continue`：跳过本次循环，直接进入下一循环
6. `break`：直接结束循环。

1.7 输入和输出

1.7.1 输出

1. `System.out.println()`：打印换行
2. `System.out.printf()`：格式化输出
3. `System.out.print()`：不格式化输出不打印换行

1.7.2 输入

1. `System.in`：需要手动处理异常
2. 借用 `Scanner` 类来进行输入

1.8 方法

1. 基本语法：方法必须在类中，不能单独存在

修饰限定符，返回值，方法名（驼峰命名），参数列表，方法体（只有在调用的时候才会执行方法体）

2. 方法调用：方法名 +（实参列表），代码要进入方法体内部执行，并且完成参数传递过程，方法执行完毕，回到调用位置继续执行。

3. 方法重载：方法名相同，方法在同一个作用域中，方法的参数个数不同，参数类型不同。但是方法返回值的类型不影响重载。（重写和重载是两个完全不一样的概念）

1.9 数组

概念：批量创建一组相同类型的变量

创建方式：

- `int[] a = {1, 2, 3, 4};`

- `int[] a = new int[]{1, 2, 3, 4};`
- `int[] a = new int[num];` `num` 为数组的大小。

使用：

- `arr.length` 用于求数组长度
- `arr[i]`，数组的下标是从 0 开始的，如果超出范围，则会抛出数组下标越界异常

引用：

1.10 面向对象

1.10.1 类和对象的概念

类相当于是图纸，对象是根据图纸造出来的房子。类相当于 C 语言中的结构体，而对象则是该自定义类型创建出来的变量。

1.10.2 基本语法

类的创建：

- 修饰符：`public`
- `class`
- 类名：`public` 修饰的类名必须和文件名一致，通常以大驼峰的形式命名。
- 类体：用一对大括号括起来，里边包括成员变量和成员函数/方法

类的实例化（对象的创建）：使用 `new` 关键字

1.10.3 `this` 关键字

`this` 关键字：

- 获取到当前对象的引用，即指向当前对象的引用。
- `this` 的类型就是当前类的类型。
- `this` 这个引用的指向不能修改。
- `this` 也可以用来调用方法
- `this` 不能是 `null`

当成员的名字和参数名字相同时，通过 `this.` 的方式显示的区分出方法的参数名和成员的名字。

1.10.4 对象的初始化

1. 默认值初始化：整数浮点数都初始化为 0，布尔类型初始化为 `false`，引用类型初始化为空指针
2. 就地初始化：在创建成员的时候直接使用 `=` 来赋值进行初始化。
3. 代码块初始化：在类里面用大括号进行初始化
4. 构造方法。（快捷键：alt+ins）

- 构造方法的方法名和类名一致。
- 构造方法不需要写返回值的类型，内部也不需要写 `return` 语句。
- 构造方法不需要显示调用，`new` 的时候自动被调用
- 构造方法支持重载

1.10.5 toString

用于打印对象的信息。

toString 方法

```
@Override
public String toString(){
    return this.name + "," + this.gender//假设这个类里面有name和gender两个成员变量
}
```

也可以用 alt+ins 快捷键直接用.

1.11 包

包就相当于是目录，当代码中文件太多，就需要放在不同的目录中，即放在不同的包中。

导入包中的类：`import 包名`

1.11.1 常见的系统包

1. `java.lang`：系统常用基础类 `String`、`Object`，此包从 JDK1.1 后自动导入。
2. `java.lang.reflect:java`：反射编程包
3. `java.net`：进行网络编程开发包
4. `java.sql`：进行数据库开发的支持包
5. `java.util`：是 Java 提供的工具程序包。

1.11.2 static 关键字

1. 类中的某个成员加上 `static`，说明这个成员是一个类属性/类方法。，如果没有 `static`，成员是一个实例属性/实例方法。

2. 类属性可以用类名访问。如：`Cat.n`(这里 `Cat` 是个类名，`n` 是类里边由 `static` 关键字修饰的成员变量)。即不需要创建实例来通过类名访问类成员变量。

3. 修饰方法：静态方法。通过类名来访问，即调用静态方法不需要创建实例。`this` 关键字是只当前对象的引用，调用静态方法没有创建实例，所以在静态方法中不能使用 `this`。（静态方法和实例无关，只和类有关）同理也无法在 `static` 方法中访问非 `static` 的变量和方法。

4. 修饰代码块：例如

static 修饰代码块

```
static {
    //这个加上static的代码块叫做“静态代码块”
    //静态代码块只在类加载的时候执行一次
    //一般用来初始化静态成员
    //类加载始终是在创建实例之前
    //static修饰的代码块始终是在普通的代码块之前执行的
}
```

1.11.3 静态导入包

`import static 包名`

1.12 访问限定符

面向对象的特性包括：类和对象，抽象，封装，继承，组合，多态，反射/自省等。其中封装、继承、多态最具有代表性。

1.12.1 `public`

修饰的成员可以被外部的类随意访问。

1.12.2 `private`

修饰的成员只能在自己类的内部使用

1.12.3 `protected`

修饰的成员

1.12.4 `default`

即成员前不加任何修饰，这种成员只能在当前的包里被使用。（包级访问权限）

1.13 内部类

比较少用，仅作了解。
把类的定义写在另一个类中。
1. 普通的内部类/成员内部类
2. 静态内部类（内部类前用 `static` 修饰）
静态内部类不依赖外部的 `this`，可以随意创建
3. 匿名内部类（相对比较常用）

匿名内部类

```
//假设当前包里有个A类。
//此时创建了一个匿名内部类，这个类没有名字
//这个类是A类的子类（继承自A类）
A a = new A(){
    //定义相关属性和方法
}
```

4. 局部内部类
把类定义到方法里。

1.14 栈堆

JVM 中的内存区除了堆和栈之外，还有方法区，方法区存的是“类相关的信息”。

对于属性来说，如果是**实例属性**，其信息跟着实例走，如果是**类属性**，其信息在方法区中，对于方法来说，无论是否有 `static` 修饰，对应的内容都是在方法区中。

1.15 String 类

1.15.1 创建字符串

创建字符串

```
//方式1
String str = "Hello";
//方式2
String str = new String("Hello");
//方式3
char[] array = {'a', 'b', 'c'};
String str = new String(array);
```

Java 中的字符串（ String ）和字符数组（ char[] ）之间没有关联关系

1.15.2 字符串比较

Java 中如果针对引用类型使用 == ，此时比较的是两个引用的地址是否相等。

对于：

字符串比较 1

```
String str1 = "Hello";
String str2 = "Hello";
```

上边的 str1 和 str2 创建的的地址一样即 str1 == str2 为 true ，但是如果通过上边的方式 2、3 创建的，则不相等。

Java 中字符串常量会被保存到字符串常量池中，此时字符串只需要保存一份即可。

比较字符串的内容是否相等应使用 .equals() 方法。例如：

字符串比较 2

```
String str1 = "Hello";
String str2 = "Hello";
System.out.println(str1.equals(str2));
```

如果字符串变量和字符串常量比较，应按如下格式书写：

字符串比较 3

```
String str1 = "Hello";

//不建议写成这种方式，一旦str1是null，此时就会抛出空指针异常
if(str1.equals("Hello")){
    //代码块
}
//建议写成如下方式，如果str1是null，不会抛出异常，而是返回false
if("Hello".equals(str1)){
    //代码块
}
```

1.15.3 字符串常量池

池：计算机中一个非常重要的术语。例如内存池，线程池，进程池，数据库连接池，对象池……等。

池的目的就是为了降低开销，提高效率。本质是把频繁使用的东西保存好，以备用到的时候随时就能用。

Java 会把一些字符串常量放到内置的“字符串常量池”中。

可以使用 `String` 中的 `intern()` 的方法手动把 `String` 对象加入到字符串的常量池中。即：

加入字符串到常量池

```
String str1 = new String("Hello").intern();
```

此段代码的含义是，`new` 了一个 `String` 对象，该 `String` 对象的内容是 `"hello"`，然后调用 `intern` 方法，该方法拿着当前的字符串去字符串常量池中找，看当前这个内容是否存在于池中。

如果存在，直接返回该池中字符串对应的地址，如果不存在，则把当前的字符串内容加到常量池中，然后返回池中的地址。

1.15.4 字符串不可变

Java 的 `String` 是不可变对象（对象本身不能修改）。

`final` 修饰的是常量，如果修饰的是一个引用类型，表示的是该引用的指向（引用中存的地址不能改），若修饰类，表示这个类不能被其他类继承。

在 `String` 的实现源码中，是用 `private` 修饰的字符数组来存放字符串内容的，所以无法在 `String` 类的外部通过 [] 的形式来获取或修改该字符数组的内容。

Java 的 `String` 设计成不可变的原因

- 方便放入池中，如果是可变的对象，一旦池中的内容发生改变，就会影响到所有引用这个池对象的结果
- 对象内容不可变，则对象的 hashCode 也不可变。方便和 hash 表这样的结构配合使用
- 对象不可变，线程安全更有保证

Java 中为了方便修改，提供了 `StringBuilder` 和 `StringBuffer` 这样的类。

通过“反射/自省”的方式修改字符串内容，例如

修改字符串

```
String str1 = "Hello";
//通过反射的方式修改"Hello"的内容
//反射是特殊手段，而不是常规手段

//1.获取到String的类对象
//2.根据“value”这个字段名字，在类对象中拿到对应的字段（仍然是图纸的一部分，相当于是局部放大图）
Field valueField = String.class.getDeclaredField("value");
//让value这个private的成员也能被访问
valueFiled.setAccessible(true);
//3.根据图纸，把str这个对象拆开，取出里面的零件
char[] value = (char[])valueFiled.get(str);
//4.修改零件的内容
value[value.length] = 'a';
```

C++ 的 std::string 就是可变对象

- 可变对象方便修改（修改起来比较高效）

C++ 为了高效的保存字符串，也有类似的“池”机制，由于是可变对象，涉及到“写时拷贝”机制。

1.15.5 字符串、字符、字节

1.15.5.1 字符转字符串

使用一个数组，构造字符串。例如：

字符转字符串

```
char s = {'a', 'b', 'c'};
String str = new String(s);
```

1.15.5.2 字符串转字符

使用 charAt 来获取到指定下标位置的字符。

字符串转字符

```
String str = "abc";
System.out.println(str.charAt(0));
//使用.length()方法获取到字符串长度，而数组的长度.length是一个属性
System.out.println(str.length());
```

toCharArray 方法，该方法相当于是在内部创建了一个新的字符数组并返回，修改这个返回值并不会影响到原字符串的内容。

1.15.5.3 字节转字符串

使用一个字节数组来构造字符串（要求字节数组中保存的内容符合 Java 字符的编码方式）。

1.15.5.4 字符串转字节

使用 getBytes 方法完成，在网络编程中经常会用到。

字符串转字节

```
String str = "abc";
str.getBytes();//可以把当前字符串的内容转换到字节数组中
```

1.15.6 字符串常见操作

1.15.6.1 字符串的比较

1. 比较相等： equals 是区分大小写的比较， equalsIgnore 是不区分大小写的比较
2. 比较大小：（按字典序来比较大小） compareTo()

字符串比较大小

```
String str1 = "abc";
String str2 = "abcd";
int result = str1.copmareTo(str2);
```

- 如果 str1 大于 str2，返回一个正数
- 如果 str1 等于 str2，返回 0
- 如果 str1 小于 str2，返回一个负数

同理，`compareToIgnoreCase` 为忽略大小写的比较。

1.15.6.2 字符串查找

```
//判断一个字符串是
public boolean contains(CharSequence s)
//从头开始查找指定字符串的位置，查到了返回位置的开始索引，如果查不到返回-1
public int indexOf(String str)
//从指定位置开始查找子字符串的位置
public int indexOf(String str, int fromIndex)
//从后向前查找子字符串的位置
public int lastIndexOf(String str)
//由指定位置从后向前查找
public int lastIndexOf(String str, int fromIndex)
//判断是否以指定字符串开头
public boolean startsWith(String prefix)
//从指定位置判断是否以指定字符串开头
public boolean startsWith(String prefix)
//判断是否以指定字符串结尾
public boolean endsWith(String suffix)
```

开头结尾函数用法：

- 判断某个链接的协议类型，用 `startsWith`
- 判断某个文件的类型，会用 `endsWith` 判断其扩展名

1.15.6.3 字符串替换

```
//替换所有指定内容
public String replaceAll(String regex, String replacement)
//替换首个内容
public String replaceFirst(String regex, String replacement)
```

由于字符串是不可变对象，替换不会修改原字符串，而是产生一个新的字符串。

Tips

`String regex`：正则表达式

正则表达式使用一些特殊的字符来描述某些字符串的筛选标准。正则表达式也是一个字符串。

1.15.6.4 字符串拆分

将一个字符串按照指定的分隔符划分为若干个子字符串

```
//将字符串全部拆分
public String[] split(String regex)
//将字符串部分拆分，该数组长度就是limit极限
public String[] split(String regex, int limit)
```

注意事项：

- 字符 "|","*","+" 都得加上转义字符，前面加上 "\"。
- 如果是 "."，那么就得写成 "\\."。
- 如果一个字符串有多个分隔符，可以用 "|" 作为连字符。

1.15.6.5 字符串截取

从一个完整的字符串中截取出部分内容

```
//从指定索引截取到结尾
public String substring(int beginIndex)
//截取部分内容
public String substring(int beginIndex,int endIndex)
```

注意事项：

- 索引是从 0 开始的。
- `substring(int beginIndex,int endIndex)`，包含下标为 `beginIndex` 的字符，不包含 `endIndex` 的字符。

1.15.6.6 其他操作方法

```
//去掉字符串的左右空白符（空格，换行，回车，制表符等），保留中间空格
public String trim()
//字符串转大写
public String toUpperCase()
//字符串转小写
public String toLowerCase()
//字符串入池
public native String intern()
//字符串链接，等同于+，不入池
public String concat(String str)
//求字符串长度
public int length()
//判断是否为空字符串""，但不是null，而是长度为0
public boolean isEmpty()
```

`native` 关键字修饰的方法称为本地方法，表明此方法不是由 Java 实现的，而是由实现 JDK 的语言实现的。

`String` 类并没有提供首字母大写操作，需自己实现。

```
public static String firstUpper(String str){
    if ("".equals(str) || str == null){
```

```
        return str;
    }
    if (str.length() > 1){
        return str.substring(0,1).toUpperCase() + str.substring(1);
    }
    return str.toUpperCase();
}
```

1.15.7 StringBuffer 和 StringBuilder

如果需要可变版本的 String，就需要 StringBuffer 或 StringBuilder，这两个类的用法基本一致，下边以其中一个为例。

StringBuilder 用法

```
//1.append字符串的追加，等同于String的+=
//Stringde +=会产生新的String对象，如果在循环中使用则极其低效
StringBuilder stringBuilder = new StringBuilder("hello");
for(int i = 0; i < 100; i++){
    stringBuilder.append(i);
}
//字符串反转
stringBuffer.rereverse();
System.out.println(stringBuilder.toString());
//字符串删除
stringBuilder.delete(2,4);//从2删除到4（包含2不包含4）
System.out.println(stringBuilder.toString());
//插入数据
stringBuilder.insert(0,"word");在位置0插入字符串word
```

String、StringBuffer 和 StringBuilder 的区别

- String 的内容不可以修改，StringBuffer 与 StringBuilder 的内容可以修改
- StringBuffer 与 StringBuilder 大部分功能相似。
- StringBuffer 采用同步处理，属于线程安全操作，StringBuilder 未采用同步处理，属于线程不安全操作。

synchronized 关键字：同步，一个方法加上该关键字，很可能就是线程安全的。

1.16 面向对象

主要是关于继承、组合、多态、抽象类、接口。

1.16.1 继承

目的就是为了让代码能够很好的被重复利用，为了把类进行代码重用。
两个核心概念：父类（被继承）（又叫基类、超类）、子类（继承）（又叫派生类）
extends 关键字

继承语法

```
class 子类 extends 父类{

}
```

- 继承也有一层“扩展”的意思，即在类保持现有功能的前提下，加入新的功能。
- 使用 `extends` 指定父类。
- 子类会继承父类的所有的属性和方法
- 对于父类的 `private` 的字段和方法，子类中是无法访问的
- 子类的实例中，也包含着父类的实例。可以使用 `super` 关键字得到父类实例的引用

继承就是为了代码重用。

每个类都有构造方法，如果不显示的创建构造方法，编译器就会给这个生成一个没有参数的构造方法。

当父类里面没有写构造方法的时候，就被生成了没参数版本的构造方法。如果直接 `new` 子类实例，就会调用刚才父类这个没参数版本的构造方法。

当父类里有构造方法的时候，并且这个构造方法带有参数的时候，编译器就不再自动生成无参数版本的构造方法，此时再创建子类实例，就需要显示的调用父类的构造方法，并且进行传参，否则创建不出父类的实例，就会编译出错。因此需要在子类的构造方法中显示的调用父类构造方法即可。（使用 `super` 关键字）

从父类继承过来的属性，既可以用 `this.` 的方式获取，也可以使用 `super.` 的方式获取，获取到的是同一个属性。如果子类中新创建了一个相同的属性，则，只能通过 `super.` 的方式获取父类的该属性，`this.` 获取到的是子类的对应属性。

子类继承父类后，子类需要先构造父类：创建子类实例的时候，先构造父类对象（执行父类构造方法的逻辑），再构造子类对象（执行子类构造方法的逻辑）。父类的构造方法必须放在第一行。

对象初始化的顺序：

- 把父类的实例全部创建完再创建子类
- 就地初始化和代码块是并列关系，按照在代码中出现的顺序来进行初始化。
- 最后再执行构造方法中的代码

1.16.1.1 `protected` 关键字

`protected` 关键字修饰的变量可以被子类（无论是不是在同一个包内）访问，也可以被同一包下的其他类访问。

类前面加 `public` 表示这个类可以被其他包使用,不加表示只能在当前包使用,不能加 `protected` 和 `private`

1.16.1.2 `final` 关键字

`final` 关键字加在类前，可以显示的禁止类继承，以防止类继承被滥用。

1.16.2 组合

组合也是为了代码重用，也是面向对象的一个重要特性。一个类的成员也可以是其他的类。

将多个类的实例作为另一个类的字段。

组合表示 `has-a` 语义 可以理解为一个学校**包含**若干老师、学生和教室等。实际开发中大部分场景使用组合。

继承表示 `is-a` 语义，即**猫**是一种动物。

1.16.3 多态

1.16.4 向上转型

使用父类的引用指向一个子类的实例。向上转型可以省略强制类型转换。

不同类型的引用之间不能相互赋值，除非两者之间是父子关系。

向上转型发生的时机：

- 直接赋值
- 方法传参
- 方法返回

向上转型后，无法通过父类引用访问子类特有的属性和方法

1.16.5 动态绑定

如果父类中包含的方法在子类中有对应的同名同参数的方法，就会进行动态绑定。（这里静态指的是编译期，动态指的是运行时。）运行时决定要调用哪个方法。

如果方法只在父类存在，此时调用的方法就是父类的方法（不涉及动态绑定），如果方法只在子类中存在，此时调用方法就会编译报错（不涉及动态绑定）。

如果某个方法在父类和子类中都存在，并且参数相同，此时调用该方法就会涉及到动态绑定。在程序运行时，看当前引用究竟指向的是父类实例还是子类实例，指向父类实例就调用父类中的那个方法，指向子类就调用子类中的那个方法。

Tips

需要给子类的方法加注解 `@Override`，显式的告诉编译器当前这个子类方法是重写了父类的方法。

注解是为了让编译器进行更好的检查和校验工作，加上注解 `@Override` 就是明确的告诉编译器，我们的目的就是进行重写，防止无意中写出了这种方法重写的代码

如果某个方法在父类和子类中都存在，但是参数不同，此时调用方法，不会涉及动态绑定，而是相当于方法重载。

1.16.6 方法重写

子类实现父类的同名方法，并且参数的类型和个数完全相同，这种情况称为覆写/重写/覆盖（Override）。

- 重写和重载不一样。
- 普通方法可以重写，`static` 方法不能重写
- 重写中子类的方法的访问权限不能低于父类方法的访问权限
- 重写的方法返回值类型不一定和父类相同。（父类子类返回值要有一定的关系，假设返回值类型互不相干，就会编译出错，假设返回类型具有父子关系，就可以编译通过）

1.16.7 理解多态

多态是一种程序设计的思想方法，具体的语法体现，向上转型、方法重写、动态绑定。

多态：一个引用，对应到多种形态。多态的设计思想，本质上是“封装”的更进一步。

1.16.8 向下转型

把父类的引用转为子类的引用，向下转型必须保证操作合理，否则会存在问题。

使用场景：

- 有些方法只有子类有，父类没有，此时使用多态的方式则无法执行到对应的子类方法，就必须把父类的引用转回成子类的引用，然后调用对应的方法

使用 `instanceof` 做出判断（ `if(父类引用 instanceof 子类引用)` ），判定当前的父类的引用是不是指向该子类，如果不是就不进行向下转型。

1.16.9 在构造方法中调用重写的一个方法

1.A 是 B 的父类，构造 B 的时候，就需要先构造 A 的实例

2. 构造 A 的实例，就会调用 A 的构造方法

3. 调用 A 的构造方法（该方法里调用了另外一个方法 `func()` ， `func()` 在 B 中重写）的时候，就会调用到 `this.func()` ，而此时的 `this` 指向的是子类 B 的实例，触发了方法的动态绑定。

4. 此时 B 中的初始化代码（包括就地初始化、代码块初始化以及构造方法的初始化都没有执行到）

因此，要避免在构造方法中调用其他可能被重写的方法。

1.16.10 抽象类和接口

抽象类和接口是搭配多态来使用的。

不需要实例化的类称为抽象类，本身没有方法体，只是为了被子类重写的方法，这种方法称为抽象方法。

给类前面加上 `abstract` 关键字表示为抽象类，此时如果创建抽象类的实例则会编译报错。抽象类中可以有普通的属性和方法，可以有静态的属性和方法，可以继承其他的类，也可以被其他的类继承。

给方法前加上 `abstract` 关键字表示为抽象方法，此时的方法不需要方法体，抽象方法只能在抽象类中存在（也可以在接口中存在），抽象方法的存在就是为了让子类进行重写。抽象方法不能是 `private` 。

接口：抽象类的进一步，抽象类只是不能实例化，其他方面与普通类类似。接口不仅不能实例化，同时也不具备类的各种特性。接口的作用是为了解决 Java 不能多继承的问题。

- `interface` 关键字表示接口。 `public interface Shape{}`
- 接口中可以放抽象方法（不用写 `abstract` 关键字）, 接口中不能放普通的方法。
- 接口中不能放普通的属性，只能放 `public static final` 修饰的属性, 同理， `public static final` 也可以省略。
- 接口不能继承自其他的类，但是可以继承自其他的接口。
- 接口不能被其他的类继承，只能被其他的类实现（使用 `implements` 关键字）。

  `public class Circle implements Shape{ }`

抽象类和接口的对比：

- 抽象类和普通类差不多，只是不能实例化，接口与普通类相差很多（属性，方法和其他类的关系等）
- 一个类只能继承自一个抽象类，但是一个类可以实现多个接口

接口命名一般使用 I 作为前缀，接口命名的时候，一般使用形容词词性的单词进行命名。

```
1 //动物类
2 abstract pubilc Animal{
```

```
    protected String name;
    public Animal(String name){
        this.name = name;
    }
}
//跑的接口
public interface IRunning{
    abstract void run();
}
//飞的接口
public interface IFlying {
    void fly();
}

//小鸟类（继承动物的类并实现跑的接口）
public class Bird extends Animal implements IRunning,IFlying{ // 多个接口以逗号隔开
    public Bird(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println(this.name + "一跳一跳的跑");
    }

    @Override
    public void fly() {
        System.out.println(this.name + "飞起来了");
    }
}
```

Tips

接口相当于是一种约束，要求了实现该接口的类，必须重写所有接口中的抽象方法。

接口和接口之间是可以继承的。（说是继承，表示成组合更合适点）

```
//跑的接口
public interface IRunning{
    abstract void run();
}
//游泳的接口
public interface ISwimming {
    void swim();
}

//两栖类接口
public interface IAmphibious extends IRunning ,ISwimming{
    //此时该接口就同时包含了IRunning中的抽象方法
    //也同时包含了ISwimming中的抽象方法
}
```

1.17 处理异常

异常是程序**运行**过程中出现的一种错误。

防御式编程：

- LBYL(Look Before You Leap)：操作之前做充分的检查，检查完上一步之后，再来做下一步的操作。如果上一步失败，就不继续执行
- EAFP(It's Easier Ask Forgiveness than Permission)：事后获取原谅比事前获取许可更简单，先斩后奏。

```
//EAFP风格的语法

try{
    //有可能出现异常的语句
}[catch (异常类型 异常对象){
}......]
[finally {
    //异常出口
}]
```

异常的具体语法：

- `try` 关键字：`try` 语句块中放置可能会抛出异常的代码。
- `catch` 语句块中放置用来处理异常的代码，当 `try` 中出现异常的时候，就去会进入 `catch` 中执行
- `throw`：主动抛出一个 Java 异常（Java 的异常本质上就是一个一个的对象）。
- `throws`：某个方法可能会抛出某些异常。
- `finally`：一般用于异常处理完毕后的收尾工作。

```
try{
    System.out.println("try 中异常之前的代码");
    int[] a = null;
    System.out.println(a[0]);
    System.out.println("try中异常之后的代码");
}catch (NullPointerException e){ //e为形参
    System.out.println("catch中的代码");
    System.out.println("e中的信息");
    //这个方法能够打印出当前出现异常的代码对应的调用栈的信息
    e.printStackTrace();
}
```

上边代码中 e 类似于一个形参，当 `try` 的代码抛出一个异常之后，e 就对应着这个异常。通过 e 就可以获取到异常的一些具体信息（哪个代码中出现了异常）

如果 `try` 中可能抛出多种异常的化，也就需要多个 `catch` 语句来进行处理，多个 `catch` 语句与多分支语句类似。

```
try{
    //有可能出现异常的语句
}catch (异常类型1 异常对象1){
    //处理异常代码1
}catch (异常类型2 异常对象2){
```

```
    //处理异常代码2
}
```

使用一个 `catch` 语句捕获多个异常：如果程序对于多个异常的处理逻辑是一样的，就可以使用此方式。抛出这若干个异常中的任何一个，都会触发 `catch`

```
try{
    //有可能出现异常的语句
}catch (异常类型1 | 异常类型2 异常对象1){
    //处理异常代码1
}
```

`finally` 中的逻辑无论是前面的代码中是否触发异常，都会执行到

`throw`：20 视频的 20 分钟-24 分钟

`throws`：标注当前的方法，可能抛出什么样的异常

```
public static int divide(int x, int y) throws ArithmeticException{
    if(y == 0){
        throw new ArithmeticException("抛出0异常");
    }
    return x / y;
}
```

1.18 泛型

```
class 泛型类名称<类型形参列表>{
    //可以使用的参数列表
}

class ClassName<T1, T2, T3, ...,Tn>{
    //类体
}
```

类型形参常用的名称

- E 表示 Element
- K 表示 Key
- V 表示 Value
- N 表示 Number
- T 表示 Type
- S，U，V 等等：第二、第三、第四个类型

实例：

```
package Wsn;

public class Myarray<E> {
    private E[] array = null;
    private int size;
```

```
    private int capacity;

    public Myarray(int capacity) {
        //由于E的类型不确定，无法创建E的实例，所以要创建Object类型然后强制转换成E
            类型
        array = (E[]) new Object[capacity];
        size = 0;
        this.capacity = capacity;
    }
    public void add(E data){
        if (size < capacity) {
            array[size++] = data;
        }
    }
    public E get(int index){
        return array[index];
    }
    public int size(){
        return size;
    }

    public static void main(String[] args){
        Myarray<String> str = new Myarray<>(10);
        str.add("hello");
        str.add("word");
        int size = str.size();
        String str1 = str.get(0);
        String str2 = str.get(1);
        System.out.println("当前数组元素的个数是: " + size);
        System.out.println("它们分别是: "+ "[" + str1 + "," + str2 + "]");
    }
}
```

- 使用泛型后，针对对象实例化的时候就需要填写泛型参数的实际类型
- Java 的泛型只能是引用类型，如果是内置类型，必须使用其对应的包装类（包装类也是引用类型）

1.18.1 类型边界

定义泛型类的时候，对未来实例化的时候能传入的实参类型作出限制

```
class 泛型类名称<E extends U>{
    //类体
}
```

实例化时，E 只能是 U 或 U 的子类，否则会报错。