前言：

面向对象与面向过程程序设计的区别：

**编程思路**

* **面向过程**：将问题分解为一系列的步骤或过程，按照顺序依次执行这些过程来解决问题。强调的是 “怎么做”，把程序看作是一系列函数或过程的集合，数据在这些过程之间传递和处理。例如，在一个简单的学生成绩管理系统中，可能会有计算平均分、打印成绩单等过程，每个过程都针对数据进行特定的操作。
* **面向对象**：将问题域中的事物抽象为对象，每个对象都包含数据（属性）和操作这些数据的方法（行为）。强调的是 “谁来做”，把程序看作是多个相互协作的对象的集合，通过对象之间的消息传递来完成任务。在学生成绩管理系统中，会有学生对象、课程对象等，学生对象具有姓名、成绩等属性，以及获取平均分、添加成绩等方法。

**数据和方法的关系**

* **面向过程**：数据和操作数据的方法是分离的。数据通常以全局变量或结构体的形式存在，方法（函数）则独立地对这些数据进行操作。例如，在一个计算几何图形面积的程序中，可能会有一个计算圆形面积的函数，它接受圆的半径作为参数，而半径和其他相关数据可能是在函数外部定义的全局变量。
* **面向对象**：数据和操作数据的方法封装在对象内部，形成一个紧密的整体。对象的方法可以直接访问和操作对象自身的数据，外部代码只能通过对象提供的公共接口来访问和修改对象的数据。例如，在一个表示圆形的对象中，半径是该对象的一个属性，而计算面积的方法则是该对象的一个成员函数，它可以直接访问和使用对象的半径属性来计算面积。

**程序结构**

* **面向过程**：通常采用线性的、层次化的结构，由一系列函数调用和控制流组成。程序的执行顺序由函数的调用顺序决定，容易出现复杂的控制逻辑和大量的函数参数传递。例如，一个大型的文件处理程序可能会由多个函数组成，这些函数按照打开文件、读取数据、处理数据、写入数据、关闭文件等顺序依次调用。
* **面向对象**：采用更加模块化、层次化的结构，由多个相互关联的对象组成。对象之间通过消息传递进行通信和协作，程序的结构更加清晰和易于维护。例如，在一个图形用户界面（GUI）应用程序中，可能会有窗口对象、按钮对象、文本框对象等，这些对象各自负责自己的功能，通过相互发送消息来实现整个应用程序的功能。

**可维护性和可扩展性**

* **面向过程**：当程序规模较大时，由于数据和方法的分离，以及复杂的控制逻辑，可能导致维护和扩展困难。修改一个功能可能需要在多个函数中进行相应的修改，容易引入新的错误。例如，在一个已经开发好的库存管理系统中，如果要添加一个新的库存操作功能，可能需要修改多个与库存相关的函数。
* **面向对象**：具有更好的可维护性和可扩展性。由于对象的封装性，修改一个对象的内部实现通常不会影响到其他对象，只需要关注该对象的接口是否发生变化。同时，通过继承和多态性，可以方便地扩展现有对象的功能或创建新的对象。例如，在一个动物模拟程序中，如果要添加一种新的动物，只需要创建一个新的动物类，继承自动物基类，并实现其特定的行为方法即可。

**代码复用性**

* **面向过程**：代码复用主要通过函数的调用和参数传递来实现，但这种复用方式相对有限，往往需要针对具体的问题进行修改和调整。例如，一个用于计算数组平均值的函数，在不同的项目中使用时，可能需要根据数组的类型、数据结构等进行修改。
* **面向对象**：通过继承和多态性可以实现更高效的代码复用。子类可以继承父类的属性和方法，并且可以根据需要进行重写和扩展。多态性使得不同的对象可以对相同的消息做出不同的响应，提高了代码的灵活性和复用性。例如，在一个图形绘制系统中，可以定义一个图形基类，其中包含绘制图形的通用方法，然后通过继承创建不同类型的图形类，如圆形、矩形、三角形等，这些子类可以复用基类的绘制方法，并根据自身的特点进行重写。

**Java开发环境的搭建**

**JDK与JRE**

JRE是Java程序的运行环境，其中包含了Java虚拟机，Java基础库类

JDK是Java的开发工具包，其中包含了JRE，同时包含了编译器以及很多Java程序的调试和分析工具

简而言之，只是测试Java程序，计算机中只有JRE也是可以的；但是开发Java程序则需要的是JDK。

**Java的基础**

* 1. Java代码内的自定义类名应该遵循“驼峰规则”（首字母大写，单词间的第一个字母也要大写）
  2. 命名时只能由字母、下划线、美元符号$、数字组成，且第一个字符不能为数字
  3. 一个Java源文件只有一个public声明的类，并且二者命名保持一致（idea内创建源文件时自动创建类）图形用户界面, 文本, 应用程序

     描述已自动生成
  4. Java代码对字母大小敏感，且为了代码美观，应注意设计程序时代码的缩进（缩进快捷键tab）
  5. Java中的注释分为三种，单行注释，多行注释与文档注释

1. 单行以//开头，不用结尾
2. 多行以/\*开头，\*/结尾
3. 文档注释以/\*\*开头，\*/结尾；注释内容包括一些说明文字以及JavaDoc标签
   1. Java常用开发工具：eclipse与idea

Java语言编程的基础

1. **常量与变量**

Java中的关键字与保留字

|  |  |
| --- | --- |
| 分类 | 关键字与保留字 |
| 基本数据类型关键字 | byte、short、int、long、float、double、char、boolean、void |
| 引用数据类型关键字 | class、interface、enum、extends、implements |
| 流程控制关键字 | if、else、switch、case、default、for、while、do、break、continue、return |
| 访问权限修饰符关键字 | public、private、protected、default（默认访问权限，无关键字，隐式存在） |
| 类、成员变量和方法相关关键字 | static、final、abstract、transient、volatile、synchronized、this、super |
| 异常处理关键字 | try、catch、finally、throw、throws |
| 包管理和导入关键字 | package、import |
| 其他关键字 | instanceof、new |
| 保留字 | const、goto |

Byte

Short

整型 int

Long

数值型 float

浮点型

基本数据类型 字符型char

Double

布尔型boolean

数据类型

类

引用数据类型 接口

数组

**变量的本质：可操纵的存储空间**

变量必须先声明再使用

变量声明的语法：数据类型 变量名；

（若一次声明多个变量，则相应的语法为：数据类型 变量名1,变量名2,…;）

同一个变量名（标识符）只能出现一次（前提：同一个代码块内）

变量的值可以被随意改变

**常量与变量的语法相似，区别在于定义常量时使用关键字final**

常量是固定值，除非修改该代码中语句值，否则不能通过其他方式改变

**Final也是一个修饰符，可以修饰类，类的属性方法等，但是不可以修饰构造方法**

Final修饰的类不可以被继承，即无子类，但是可以是其他类的子类

Final修饰的方法不可以重写

Final修饰的引用型变量不可以改变地址

**Java是强类型语言，数据类型转换方式：（1）自动类型转换（2）强制类型转换**

* + 1. 自动类型转换：占用字节数小数据类型的值可以直接赋给占用字节数大数据类型的变量

凡事都有特殊，记住int常量是可以直接赋值给char，short以及byte的，只要常量值在这三个类型的范围内即可（char，short是2B，byte是1B）

* + 1. 强制类型转换：将多的转换成少的（有精度丢失问题）

数据类型 变量名 = （数据类型）变量值；

* + 1. 引用类型也存在类型转换，即父类引用指向子类对象；子类引用指向父类对象（也属于强制类型转换）

**关于输入**

在C语言中，最简单的方式就是通过scanf函数实现输入功能，基本语法与c中的输出函数printf没有区别

类比到Java中，输出语句有三种

1. **System.out.println()**将指定的内容输出到控制台，并在输出后换行eg：System.out.println("Hello, World!");，这行代码会在控制台输出Hello, World!，并自动换行，后续的输出会在下一行开始
2. **System.out.print()**将指定的内容输出到控制台，但不会在输出后换行eg：System.out.print("Hello"); System.out.print(" World!");，这两行代码会在控制台输出Hello World!，两个字符串之间没有换行，是连续输出的
3. **System.out.printf()**按照指定的格式将内容输出到控制台

eg：int num = 10; double pi = 3.14159; System.out.printf("数字：%d，圆周率：%.2f", num, pi);，这行代码会按照指定的格式输出数字：10，圆周率：3.14。其中，%d是格式化占位符，表示输出一个整数，%.2f表示输出一个浮点数，并且保留两位小数

**在Java语言内为了实现复杂的输入功能，提供了Scanner类（Scanner类在java.util包下）**

语法：Scanner 变量名 = new Scanner（System.in）；

eg:

import java.util.Scanner;

public class A{

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("请输入一个整数: ");

// 使用nextInt方法获取用户输入的整数

int num = scanner.nextInt();

System.out.println("你输入的整数是: " + num);

// 关闭Scanner对象，释放相关资源，这是一个良好的编程习惯

scanner.close();

}

}

1. **运算符**

|  |  |  |
| --- | --- | --- |
| 运算符类型 | 运算符 | 功能 |
| 算术运算符 | + | 加法运算，也可用于字符串拼接 |
| - | 减法运算 |
| \* | 乘法运算 |
| / | 除法运算，舍去小数部分 |
| % | 取余运算，返回除法的余数 |
| ++ | 自增运算符，分为前置自增和后置自增。前置自增先将变量的值加 1，再参与其他运算；后置自增先参与其他运算，再将变量的值加 1 |
| -- | 自减运算符，分为前置自减和后置自减。前置自减先将变量的值减 1，再参与其他运算；后置自减先参与其他运算，再将变量的值减 1 |
| 关系运算符 | == | 相等 |
| ！= | 不相等 |
| > | 大于比较，返回布尔值true或false |
| < | 小于比较，返回布尔值true或false |
| >= | 大于等于比较，返回布尔值true或false |
| <= | 小于等于比较，返回布尔值true或false |
| 逻辑运算符 | && | 短路与运算，计算结果与&一致，但当第一个表达式为false时，第二个表达式不再运算 |
| || | 短路或运算，计算结果与||一致，但当第一个表达式为ture时，第二个表达式不再运算 |
| & | 与运算，当俩边表达式均为true时，结果为true，否则false |
| | | 或运算，当俩边表达式均为false时，结果为false，否则true |
| ^ | 异或运算，俩边结果不同，为true，不然为false |
| ! | 非运算 |
| 位运算符 | << | 将二进制数左移指定位数，移动n位就是\*2的n次方 |
| >> | 将二进制数右移指定位数，移动n位就是/2的n次方 |
| >>> | 无符号右移，符号位也跟着移动 |
| & | 与运算，同为1，结果为1，否则为0 |
| | | 或运算，同为0，结果为0，否则1 |
| ^ | 异或运算，同则结果为0，否则为1 |
| ~ | 按位取反，0变1，1变0 |

|  |  |  |
| --- | --- | --- |
| 赋值运算符 | = | 单纯赋值 |
| += | X+=1即x=x+1 |
| -= | x-=1即x=x-1 |
| \*= | X\*=1即x=x\*1 |
| /= | x/=1即x=x/1 |
| %= | x%=1即x=x%1 |

三元运算符：**条件表达式**？表达式1：表达式2

条件表达式为true，执行表达式1，否则执行表达式2

1. **三种结构**

从结构化设计程序的角度出发，代码应有三种结构（顺序结构，选择结构，循环结构）

顺序结构是最简单的算法结构，除非有特殊语句，不然程序就是按照从上到下一句一句执行

**选择结构**用于代码中做一些判断，当满足什么什么条件时，执行特定语句

这个过程是通过if语句（或者switch语句）进行操作的，可以实现各种各样的逻辑判断；

语法如下

if(条件表达式1){

代码块1

}else if(条件表达式2){

代码块2

}else{

代码块n

}

当条件表达式1为true时，则执行代码块1的内容，以此类推

Switch语句语法如下

Switch(变量){

Case 值1：

代码块1；

Break;

Case 值2：

代码块2；

Break;

Default：

代码块n；

Break;

}

Switch会根据表达式匹配的值从相匹配的case处开始执行，如果没一个匹配就会进入default下的代码块n

**选择结构在使用上可以嵌套**

循环结构，在满足一定要求下可以一直执行某段程序

For，whlie，do……while

For语句的语法如下

For（初始化表达式；循环范围；循环条件；）{

//循环体

}

While语句的语法如下

While（条件表达式）{

//循环体

}

条件表达式值为true或者false

Do……while语句语法如下

Do{

//循环体

}while（条件表达式）；

Do while语句执行时为先执行一次循环体再判断条件表达式（值为true或者false）（与for，whlie的区别）

Break语句—用于强制退出循环，不执行循环剩余语句

Continue语句只能终止某一次循环，继续循环下一次循环体

**循环语句也满足任意嵌套**

1. **方法**

方法，即函数，功能，过程；在项目中具有单独功能的一段程序

方法与变量相似，使用前必须先声明，才可以被其他代码中调用

声明语法如下

修饰符 返回类型 方法名（参数类型 参数名1，……（即形参列表））{

//方法体；

Return 返回值；

}

调用语法如下

方法名（实参名1，实参名2……）；

**方法重载**

问题：在同一个程序内，有方法A，形参类型为int，若要使用方法A，且传入参数类型为double，char等等又该怎么办？

解决办法—方法的重载，在同一个类内，可以存在一个及以上的同名方法，**但是参数的类型与参数的个数必须不同（1、方法名一致，2、参数类型不同，3、参数个数不同；满足1与2或者满足1与3）**

**注意！方法重载只会和参数，方法名相关，其他的不同均不构成重载**

eg.system.out.println方法就是一个典型的重载方法

**方法的可变参数—当方法中参数个数不确定，但是参数类型确定，可以不考虑重载（毕竟重载多写不少，提升了代码冗余度）使用方法的可变参数；实际处理中，可变参数会被当成数组进行处理**

语法如下

Public static 方法类型 方法名（数据类型 …参数名）

eg. public class VarargsExample {

public static void main(String[] args) {

int sum1 = sum(1, 2, 3);

int sum2 = sum(10, 20, 30, 40);

int sum3 = sum();

System.out.println("sum1: " + sum1);

System.out.println("sum2: " + sum2);

System.out.println("sum3: " + sum3);

}

public static int sum(int... numbers) {

int result = 0;

for (int num : numbers) {

result += num;

}

return result;

}

}

输出结果为6，100， 。

**方法递归**

若方法直接或者间接调用方法本身，则称该方法递归方法

典型例子—斐波那契数列

**5、数组**

数组—一种可以存放相同数据类型的有序集合，本质上就是一片存储多个数据的存储空间

数组语法如下

数据类型[] 数组名 = new 数据类型[数组长度]；

数据类型[] 数组名 = new 数据类型[]{元素1，元素2……}；

数据类型[] 数组名 = {元素1，元素2……}；

**数据类型可以是任意基本类型或者引用类型**

通过索引操作数组元素

public class Array{

public static void main(String[] args) {

// 声明并初始化一个整型数组

int[] numbers = {10, 20, 30, 40, 50};

// 通过索引访问数组元素并输出

System.out.println("索引为2的元素是: " + numbers[2]);

// 通过索引修改数组元素的值

numbers[3] = 45;

}

}

数组的遍历

For循环：

public class Array {

public static void main(String[] args) {

int[] numbers = {1, 2, 3, 4, 5};

for (int i = 0; i < numbers.length; i++) {

System.out.print(numbers[i] + " ");

}

}

}

Foreach循环：语法是for (数据类型 变量名: 数组名){ System.out.print （变量）；}

public class Array {

public static void main(String[] args) {

int[] numbers = {1, 2, 3, 4, 5};

for (int num : numbers) {

System.out.print(num + " ");

}

}

}

Foreach与for的不同：不需要给数组元素赋值，不需要索引操作，只是想要数组的每一个元素

**使用Arrays.asList()结合forEach()方法遍历数组（针对引用类型数组更常用）**

import java.util.Arrays;

import java.util.List;

public class Array {

public static void main(String[] args) {

String[] words = {"Hello", "World", "Java"};

List<String> wordList = Arrays.asList(words);

wordList.forEach(word -> System.out.print(word + " "));

}

}

String类型的数组words通过Arrays.asList()方法转换为List集合，然后调用forEach方法，并传入一个 Lambda 表达式，在表达式中定义了对每个元素（这里用word表示）的操作，即输出该元素。

提到数组，其实第一个应该想到的就是算法，最典型的例子莫过于数组的排序算法，8大排序算法算是接触过数据结构的，都耳熟能详的地步了

（这里只是粗浅的介绍一下冒泡排序和插入排序，以后会有专门的内容整合）

**冒泡排序**

动画演示图表, 直方图

描述已自动生成

**算法步骤**

1.比较相邻的元素。如果第一个比第二个大，就交换他们两个。

2.对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

3.针对所有的元素重复以上的步骤，除了最后一个。

4.持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

5.每一轮之后都判断这一轮是否进行了至少一次交换，如果没有进行交换则说明数组已经有序。

**代码实现**

public class BubbleSort {

public static void main(String[] args) {

int[] arr = {5, 3, 8, 6, 7};

bubbleSort(arr);

System.out.println("排序后的数组：");

for (int num : arr) {

System.out.print(num + " ");

}

}

public static void bubbleSort(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// 交换元素

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

}

**插入排序**

动图演示图表

描述已自动生成

算法**步骤**

1. 从第一个元素开始，该元素可以认为已经被排序；
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描；
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置；
4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
5. 将新元素插入到该位置后；
6. 重复步骤2~5。

**代码实现**

public class InsertionSort {

public static void main(String[] args) {

int[] arr = {5, 3, 8, 6, 7};

insertionSort(arr);

System.out.println("排序后的数组：");

for (int num : arr) {

System.out.print(num + " ");

}

}

public static void insertionSort(int[] arr) {

int n = arr.length;

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

// 将 key 与已排序的元素从后往前依次比较，并移动元素，为 key 找到合适的插入位置

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

}

**二维数组**

二维数组每一个元素都是一个数组，即“数组的数组”

语法如下

数据类型[][] 数组名 = new 数据类型[m][n]

M即这个数组的长度，n即每个元素的长度，n可以声明时不指定，待后续动态创建

**Arrays工具类**

Arrays是Java提供的操作数组的工具类，有许多方法可以调用方便操作数组

|  |  |
| --- | --- |
| 方法名 | 描述 |
| asList() | 返回由指定数组支持的固定大小的列表。 |
| sort() | 将数组排序（升序） |
| parallelSort() | 将指定的数组按升序排序 |
| binarySearch() | 使用二分搜索法快速查找指定的值（前提是数组必须是有序的） |
| compare() | 按字典顺序比较两个数组 |
| compareUnsigned() | 按字典顺序比较两个数组，将数字元素处理为无符号 |
| copyOf() | 填充复制数组 |
| copyOfRange() | 将数组的指定范围复制到新数组 |
| equals() | 比较两个数组 |
| deepEquals() | 比较两个数组深度 |
| toString() | 将数组转换为字符串 |
| deepToString() | 将一个多维数组转换为字符串 |
| fill() | 将指定元素填充给数组每一个元素 |
| mismatch() | 查找并返回两个数组之间第一个不匹配的索引，如果未找到则返回-1 |
| parallelPrefix() | 使用提供的函数对数组元素进行操作 |
| parallelSetAll() | 用提供的生成器函数并行设置指定数组的所有元素以计算每个元素 |
| setAll() | 使用提供的生成器函数设置指定数组的所有元素以计算每个元素 |

**6、JVM**

JVM是基于堆栈的虚拟机，堆栈是一种数据结构，用来存储数据

意思就是说，可以把Java程序的运行看成通过对堆栈的操作完成的

图示

描述已自动生成

**编程语言中数据类型传递的方式有值传递和引用传递，but，Java中只有值传递**

面向对象编程

Java中使用class关键字定义一个类

**类由属性和方法组成**

属性，即变量或者字段

对象，即一堆属性和功能的集合体

类，即一群对象的抽象化

通常是由类创造对象

位于某个类内的变量和方法也可称为成员变量和成员方法

**对象的创建与使用**

类名 对象名 = new 类名（）；（new关键字调用对象的构造方法）

注意！成员变量和方法属于对象，不同对象之间的成员占用的存储空间不同，互不干扰

匿名对象—当方法参数只需要一个对象，或者只是想要使用这个对象的某个方法而已，可以使用匿名对象

eg. class MyClass {

public void doSomething() {

System.out.println("这是MyClass中的doSomething方法被调用了。");

}

}

class Main {

public static void main(String[] args) {

// 直接创建MyClass的匿名对象，并调用其doSomething方法

new MyClass().doSomething();

}

}

**构造方法—Java要求每个类都必须有（没有定义时，编译器会自动给默认无参构造方法，该方法也可以重载）**

也称为构造器，用于对象的初始化，即为变量成员赋初值

**构造方法重点—1、必须与类名相同，2、不能有返回值，3、构造方法调用必须通过new关键字调用**

语法如下

修饰符 类名（形参列表）{}

**所以，由此可知，new关键字不只是创建对象，更是会调用对应的构造方法，并执行该方法内的方法体**

**This关键字**

This的使用范围：成员方法和构造方法

调用的内容：成员变量，成员方法，构造方法

可以说。他就是代表当前对象

**This关键字的注意事项**

**使用范围**

this 关键字只能在实例方法、构造函数和实例初始化块中使用，不能在静态方法或静态代码块中使用，因为静态成员不属于任何特定的对象，而 this 是用于引用当前对象的。

**避免循环引用**

在使用 this 调用构造函数或方法时，要注意避免出现循环引用的情况，即一个构造函数或方法通过 this 无限地调用自身或其他方法，最终导致栈溢出错误。

**构造函数中调用时机**

在构造函数中使用 this 调用其他构造函数时，this(...) 必须是构造函数中的第一条语句，否则会导致编译错误。

**方法返回值类型**

当使用 this 作为方法的返回值时，要确保方法的返回类型与当前类的类型一致，否则会出现编译错误。

**内部类中的使用**

在内部类中使用 this 时，需要明确区分是引用内部类自身的对象还是外部类的对象。如果要引用外部类的对象，需要使用 外部类名.this 的形式。

**谨慎修改引用**

虽然可以在方法中修改 this 引用所指向的对象，但这种做法不常见且可能导致代码逻辑混乱，一般不建议这样做。通常情况下，this 引用应该保持指向当前正在操作的对象。

**Static关键字（静态变量）**

static关键字的作用

简而言之，方便在没有创建对象的情况下进行调用(方法/变量)—所以不依赖对象，也就没有this什么事情了

显然，被static关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问。

static可以用来修饰类的成员方法、类的成员变量，另外也可以编写static代码块来优化程序性能

**一般使用 类名 变量名/类名.方法名调用静态变量和方法**

静态代码块一般用于做一些全局的初始化，因为一般加载一次使用一次而已，故常用于游戏地图加载。

带有static的方法，变量，代码块只能调用带有static的方法，变量；而不带static的就可以随便使用了

**Package包**

Import关键字导入包名

面向对象编程的三大特性

三大特性有：封装，继承，多态

Java的访问修饰符：private、default、protected、public

|  |  |
| --- | --- |
| Public  （访问权限：全部可见）  称为项目可见性 | Default  （同一个类，同一个包内）  称为包可见性 |
| Private  （同一个类/私有）  称为类可见性 | Protected  （同一个类，子类，同一个包内）  称为子类可见性 |

**继承—extends关键字**

Java中只有单继承，但是继承是可以延续的，Java中所有类均有一个父类—object类

语法如下

Class 子类 extends 父类{}

**方法的重写**

1. **方法签名必须相同**：子类重写的方法必须与父类中被重写的方法具有相同的方法名、参数列表和返回类型（在 Java 5 及以后的版本中，返回类型可以是被重写方法返回类型的子类，这被称为协变返回类型）。
2. **访问修饰符限制**：子类中重写的方法不能比父类中被重写的方法有更严格的访问限制（例如，父类中方法是 protected，子类中重写的方法不能是 private）。
3. **不能重写的方法**：private 方法、static 方法、final 方法不能被重写。private 方法在子类中无法访问，所以不能重写；static 方法属于类而不属于对象，不存在多态性的概念，不能被重写；final 方法被声明为最终的，不允许子类重写。
4. **抛出异常的限制**：子类重写的方法抛出的异常不能比父类被重写方法抛出的异常更宽泛，即子类可以抛出与父类相同的异常，或者是父类异常的子类，也可以不抛出异常。

**重写的作用**

**实现多态行为**：通过方法重写，不同的子类可以根据自身的特点来实现相同的方法，从而在运行时根据对象的实际类型来调用相应的重写方法，实现多态性。这样可以提高代码的可扩展性和维护性，使得代码更加灵活和通用。

**优化或扩展父类方法功能**：当父类的方法不能满足子类的需求时，子类可以重写该方法，在保持方法签名不变的情况下，对方法的实现进行修改或扩展，以提供更符合子类特定业务逻辑的功能。

**Super关键字的使用**

Super关键字是直接对父类对象的引用，即可以通过super访问被子类覆盖的父类的方法或者属性

Super可以调用的是父类的属性，构造方法，成员方法

**Super关键字指向的其实是该对象的父类存储空间，this关键字指向的是整个对象**

|  |  |  |
| --- | --- | --- |
| 区别点 | This | Super |
| 本质 | 代表本类对象的引用 | 代表父类的存储空间 |
| 使用方式 | This.属性，方法，（） | Super.属性，方法（），（） |
| 调用构造方法 | 调用本类构造方法 | 调用父类构造方法 |
| 查找范围 | 本类找，无则找父类 | 父类 |

**多态—同一个方法调用，不同的结果**

多态的前提是必须有继承，有发生重写，父类引用指向了子类对象

只有每个孩子都重写了这个方法，然后父类引用指向子类对象，相当于有一个办法但是多个不同实体产生多种不同的想法和行为

父类A

继承 继承

继承

子类c

子类a

子类b

结果a 结果b 结果c

关于多态非静态方法的调用，遵循规则“编译看左边，运行看右边”，即在编译时看左边类是否有该方法，而实际上运行时执行的是右边的类的方法

关于多态静态方法，静态变量，成员变量调用，编译与运行均看左边

**抽象**

抽象---关键字是abstract

**注意！抽象方法只有方法声明，没有方法体，必须交由子类重写，即实现抽象方法**

**子类如果也是抽象，那就不一定要重写，可以丢给子类的子类**

**抽象必须重写，所以不能是私有的，即private**

**抽象的没有方法体，所以他就不能被调用，所以抽象类没有对象，没有实体**

**抽象类还是有构造方法，是给子类创建对象时给抽象类属性赋值的**

**接口**

接口：比抽象还抽象

接口的所有方法都是抽象的（起码抽象的还有一些具体实现不是）

接口是由关键字interface修饰的

接口也算是类，比较特殊罢了

一个类可以继承多个接口的功能

接口和接口之间也可以继承，而且比类更灵活，可以多继承

JDK8之后，接口使用static和default关键字给接口内方法修饰，让其可以有方法体

**内部类**

内部类和成员变量，成员方法地位差不多

要访问内部类，通常需要通过外部类的对象来调用

**外部访问内部类的成员变量---内部类对象名.变量名**

**内部访问外部类的成员变量---外部类名.this.变量名**

**异常**

**异常的处理方式**

**捕获异常**：使用try-catch语句块来捕获异常并进行处理。try块中放置可能会抛出异常的代码，catch块用于捕获并处理特定类型的异常。

try {

// 可能会抛出异常的代码

int result = 10 / 0;

} catch (ArithmeticException e) {

// 处理异常

System.out.println("发生了算术异常：" + e.getMessage());

}

**抛出异常**：使用throw关键字在方法内部抛出异常，还可以使用throws关键字在方法签名中声明该方法可能会抛出的异常，将异常抛给调用者处理。

public void divide(int a, int b) throws ArithmeticException {

if (b == 0) {

// 抛出异常

throw new ArithmeticException("除数不能为0");

}

int result = a / b;

System.out.println("结果：" + result);

}

**自定义异常**

可以通过继承Exception类或其子类来创建自定义异常类，以便在特定的业务场景中抛出和处理特定类型的异常。

// 自定义异常类

class MyException extends Exception {

public MyException(String message) {

super(message);

}

}

public class Main {

public static void main(String[] args) {

try {

// 抛出自定义异常

throw new MyException("这是一个自定义异常");

} catch (MyException e) {

// 处理自定义异常

System.out.println("捕获到自定义异常：" + e.getMessage());

}

}

}

**Java的基本数据类型并非对象，Java也提供了某些类来让其变为面向对象的状态，故有了包装类**

|  |  |
| --- | --- |
| **基本数据类型** | **包装类** |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

**包装类的用途**

**作为方法参数和返回值**：在许多情况下，方法需要接收对象作为参数或返回对象。包装类使得基本数据类型能够以对象的形式在方法间传递，例如，在集合类中只能存储对象，使用包装类就可以将基本数据类型放入集合中。

**提供实用方法**：包装类提供了许多实用的方法，方便对基本数据类型进行操作和处理。比如 Integer 类的 parseInt() 方法可以将字符串解析为整数，Character 类的 isDigit() 方法可以判断一个字符是否为数字等。

**自动装箱和拆箱**：从 Java 5 开始引入了自动装箱（Autoboxing）和自动拆箱（Unboxing）机制。自动装箱是指将基本数据类型自动转换为对应的包装类对象，自动拆箱则是将包装类对象自动转换为基本数据类型。

**String类**

String类表示不可变的字符串。任何对String对象的修改都会创建一个新的String对象。String类适用于字符串内容不经常变化的场景。

特点：

不可变性：String对象一旦创建，其内容不可更改。

线程安全：由于String对象不可变，因此它是线程安全的。

public class StringExample {

  public static void main(String[] args) {

      String str1 = "Hello";

      String str2 = "World";

      String str3 = str1 + " " + str2; // 字符串拼接

​

      System.out.println(str3); // 输出: Hello World

  }

}

StringBuffer类

StringBuffer类表示可变的字符串，适用于频繁修改字符串的场景。StringBuffer是线程安全的，因为它的方法都是同步的。

特点：

可变性：StringBuffer对象的内容可以修改。

线程安全：StringBuffer的方法是同步的，因此它是线程安全的。

public class StringBufferExample {

  public static void main(String[] args) {

      StringBuffer sb = new StringBuffer("Hello");

      sb.append(" World"); // 字符串拼接

​

      System.out.println(sb.toString()); // 输出: Hello World

  }

}

StringBuilder类

StringBuilder类也表示可变的字符串，但不保证线程安全。StringBuilder的性能比StringBuffer更高，适用于单线程环境下频繁修改字符串的场景。

特点：

可变性：StringBuilder对象的内容可以修改。

非线程安全：StringBuilder的方法不是同步的，因此它不是线程安全的。

高性能：由于没有同步开销，StringBuilder的性能比StringBuffer更高。

public class StringBuilderExample {

  public static void main(String[] args) {

      StringBuilder sb = new StringBuilder("Hello");

      sb.append(" World"); // 字符串拼接

​

      System.out.println(sb.toString()); // 输出: Hello World

  }

}

String、StringBuffer和StringBuilder的比较

特性StringStringBufferStringBuilder

可变性 不可变 可变 可变

线程安全 线程安全 线程安全 非线程安全

性能 低 中 高

适用场景 字符串内容不经常变化 多线程环境下频繁修改字符串 单线程环境下频繁修改字符串

字符串拼接的性能比较

在频繁进行字符串拼接的场景下，StringBuilder的性能通常优于StringBuffer和String。

日期时间类

Java提供了Date、Calendar和SimpleDateFormat类来处理日期和时间。

Date类

Date类：Date类表示特定的时间点，通常用于表示当前的日期和时间。

import java.util.Date;

​

public class DateExample {

  public static void main(String[] args) {

      Date date = new Date();

      System.out.println("Current date and time: " + date);

  }

}

Calendar类

Calendar类：Calendar类是一个抽象类，用于操作日期和时间。Calendar类提供了丰富的方法来处理日期和时间。

import java.util.Calendar;

​

public class CalendarExample {

  public static void main(String[] args) {

      Calendar calendar = Calendar.getInstance();

      int year = calendar.get(Calendar.YEAR);

      int month = calendar.get(Calendar.MONTH) + 1; // 月份从0开始

      int day = calendar.get(Calendar.DAY\_OF\_MONTH);

​

      System.out.println("Current date: " + year + "-" + month + "-" + day);

  }

}

SimpleDateFormat类

SimpleDateFormat类：SimpleDateFormat类用于格式化和解析日期和时间。

import java.text.SimpleDateFormat;

import java.util.Date;

​

public class SimpleDateFormatExample {

  public static void main(String[] args) {

      Date date = new Date();

      SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

      String formattedDate = sdf.format(date);

​

      System.out.println("Formatted date: " + formattedDate);

  }

}

**其他常用类**

Math类

Math类：Math类提供了许多静态方法用于执行数学运算，如绝对值、平方根、三角函数等。

常用方法：

abs()：返回绝对值。

sqrt()：返回平方根。

pow()：返回幂运算结果。

sin()、cos()、tan()：返回三角函数值。

log()、log10()：返回对数。

max()、min()：返回两个数中的最大值或最小值。

random()：返回一个0.0到1.0之间的随机数。

public class MathExample {

  public static void main(String[] args) {

      double absValue = Math.abs(-10.5); // 绝对值

      double sqrtValue = Math.sqrt(16); // 平方根

      double powValue = Math.pow(2, 3); // 幂运算

​

      System.out.println("Absolute value: " + absValue);

      System.out.println("Square root: " + sqrtValue);

      System.out.println("Power: " + powValue);

  }

}

Random类

Random类：Random类用于生成随机数。

常用方法：

nextInt()：返回一个随机整数。

nextInt(int bound)：返回一个0到指定范围（不包括）之间的随机整数。

nextDouble()：返回一个0.0到1.0之间的随机浮点数。

nextBoolean()：返回一个随机布尔值。

import java.util.Random;

​