Day01 笔记

SDK （software development kit），软件开发包，主要包含函数库或者工具等
JDK （Java development kit），Java 程序开发工具包，面向 Java 程序的开发者
JRE （Java runtime enviroment），Java 程序运行环境，面向 Java 程序的使用者
JVM（Java Virtual Machine），Java 虚拟机，核心组件，负责解释和执行 Java 字节码文件
API （application program interface），应用程序编程接口
API Documentation，API 说明文档，描述 API 中的类、方法等使用的方式
bin 目录
存放 JDK 中提供的 Java 各种工具命令（可执行程序），如 java、javac、javap 等
db 目录
JDK 自带的一个小型数据库，纯 Java 实现
include 目录
JVM 启动时需要引入的一些特定头文件（C 语言实现）
jre 目录
JDK 自带的一个 Java 运行环境
lib 目录
"library"的缩写，目录中提供了一些 Java 类库和库文件，即 jar 包
src.zip 文件
压缩文件，目录存放 JDK 核心类的源代码，也就是 JavaSE-API 的源代码

Day02 笔记

常见关键字：

public static void class

3.标识符

在 Java 中给类、方法、变量起的名字，就是标识符，其可以用来标识这个类、方法、变量

命名规则：

标示符可以由字母、数字、下划线_ 、美元符号$组成
标示符开头不能是数字
标识符中的字符大小写敏感
标识符的长度没有限制

标示符不能使用 Java 中的关键字或保留字
5.变量
变量的定义有 2 种格式，分别如下：
格式 1
数据类型 变量名;
变量名 = 数据值;
格式 2（推荐）
数据类型 变量名 = 数据值;
注意事项：变量一定要求先声明、再赋值、之后才能使用
6.数据类型
Java 语言是强类型语言，每一种数据都定义了明确的数据类型，不同类型变量占用内存大小不同，取值范围不同。
Java 数据类型可以分为两大类：
基本数据类型（本章重点讲解）
整形 byte、short、int、long
浮点型 float、double
字符类型 char
布尔类型 boolean
引用数据类型（面向对象部分补充）
数组
类
接口

Day03 笔记

操作符：对字面值常量或变量进行操作的符号，也称 运算符 。
表达式：用操作符把字面值常量或变量连接起来的式子（符合 Java 语法），就称之为表达式
右移：
>> 低位抛弃，高位补符号位的值
>>> 低位抛弃，高位补 0
左移：
<< 高位抛弃，低位补 0
在一个程序执行的过程中，各条语句的执行顺序对程序的结果是有直接影响的。所以，我们必须清楚每条语句的执行流程。而且，很多时候要通过控制语句的执行顺序来实现我们想要的功能。
程序中，需要执行的代码，其结构主要分为以下三种
顺序结构
分支结构
if 语句、switch 分支语句
循环结构
for 循环、while 循环、do while 循环
for 循环

循环语句可以在满足循环条件的情况下，反复执行某一段代码，这段被重复执行的代码被称为循环体语句。
当反复执行这个循环体时，需要在合适的时候把循环判断条件修改为 false，从而结束循环，否则循环将一直执行下去，形成死循环

Day04 笔记

数组，表示一块连续的内存空间，可用来存储多个数据(元素)，要求元素类型要一致。

数组定义
数组的定义有 2 种格式，分别如下：
数据类型[] 数组名； （推荐用法）
示例：
数据类型 数组名[]；
示例

```
int arr[];
double arr[];
char arr[];
```

■数组初始化
定义数组(开辟栈空间内存)的同时，给其赋上初值，就叫做数组的初始化！
数组下标
数组的下标的区间为 [0, 数组长度-1] 。
如果数组长度为 length，那么数组下标的最小值为 0，下标最大值为 length-1 。
通过数组下标给数组元素赋值：

```
System.out.println("-------------------");
//数组的遍历
for(int i = 0; i < 4; i++) {
System.out.println(arr[i]);
}
}
}
//数组长度为 4，那么其下标就是 0~3
int[] arr = new int[4];
//可以通过下标获取数组元素值
System.out.println(arr[0]);
System.out.println(arr[3]);
int[] arr = new int[4];
arr[0] = 337;
arr[1] = 340;
```

```
arr[2] = 348;
arr[3] = 352;
```
50 结合循环来赋值或者取值：

3.4 数组长度

数组长度，是指在一个数组中，可以存放同一类型元素的最大数量。

获取数组长度固定格式： 数组名.length

数组长度注意事项：

数组长度，必须在创建数组对象的时候就明确指定

数组长度，一旦确定，就无法再改变

数组长度，可以>=0（一般大于 0），但不能为负数

```
int[] arr = new int[4];
//数组下标的取值范围，从 0 开始，到数组长度-1
for(int i = 0; i < 4; i++){
arr[i] = 10 + i;
}
//获取数组每个元素的值，并且输出
for(int i = 0; i < 4; i++){
System.out.println(arr[i]);
}
```

Day05 笔记

选择排序

选择排序(Selection Sort)的原理有点类似插入排序，也分已排序区间和未排序区间。但是选择排序每次会从未排序区间中找到最小的元素，将其放到已排序区间的末尾，最终完成排序。

选择排序算法描述：

1. 初始状态：无序区间为 Arr[0.1..n]，有序区间为空;
2. 第 i==1 趟排序开始，从无序区中选出最小的元素 Arr[k]，将它与无序区的第 1 个元素交换，从而得到有序区间 Arr[0..i-1],无序区间 Arr[i..n];
3. 继续后面第 i 趟排序(i=2,3…n-1)，重复上面第二步过程;
4. 第 n-1 趟排序结束，数组排序完成。

插入排序

插入排序(Insertion Sort)，一般也被称为直接插入排序。对于少量元素的排序，它是一个有效的算法。

插入排序算法描述：

1. 将数组分成两部分，已排序、未排序区间，初始情况下，已排序区间只有一个元素，即数组第一个元素;
2. 取未排序区间中第一个元素，插入到已排序区间中合适的位置，这样子就得到了一个更大的已排序区间;
3. 重复这个过程，直到未排序区间中元素为空，算法结束。

希尔排序

希尔(shell)排序是 Donald Shell 于 1959 年提出的一种排序算法。希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为缩小增量排序，同时该算法是冲破 O(n2）的第一批算法之一。
希尔排序对直接插入排序改进的着眼点：
若待排序序列中 元素基本有序 时，直接插入排序的效率可以大大提高
如果待排序序列中 元素数量较小 时，直接插入排序效率很高
希尔排序算法思路：
将整个待排序序列分割成若干个子序列，在子序列内部分别进行直接插入排序，等到整个序列 基本有序 时，再对全体成员进行直接插入排序！

.数组拷贝

数组的长度确定后便不能修改，如果需要数组存放更多元素，可以通过创建长度更长的新数组，然后先复制老数组内容到新数组中，再往新数组中放入额外的元素。
在 java.lang.System 类中提供一个名为 arraycopy 的方法可以实现复制数组中元素的功能
二维数组
如果把普通的数组（一维数组），看作一个小盒子的话，盒子里面可以存放很多数据，那么二维数组就是像一个大点的盒子，里面可以存放很多小盒子（一维数组）。
定义格式
二维数组固定定义格式有 2 种，具体如下：
格式 1：
数据类型[][] 数组名 = new 数据类型[一维长度 m][二维长度 n]；
m：表示二维数组的元素数量，即可以存放多少个一维数组
n：表示每一个一维数组，可以存放多少个元素

Day06 笔记

面向对象基础

面向对象

OOP，面向对象编程
OOP（object oriented programming），面向对象编程
是一种以对象为中心的编程思想，通过借助对象实现具体的功能
将大问题拆分成小问题，然后借助不同对象分别解决，最终实现功能
POP（procedure oriented Programming），面向过程编程
是一种以过程为中心的编程思想，靠自己一步一步去实现功能，需要对每个步骤精确控制
强调按步骤实现功能，先分析解决问题所需步骤，再自定义方法实现每个

步骤功能，然后依次调用方法，最终实现功能

面向对象特点：

更符合人类思想习惯的思想
利用对象去实现功能
将复杂事情简单化
针对要解决问题的用户而言，可以把大问题拆解成小问题，分别指挥不同的对象去解决小问题
程序员的角色由执行者变成了指挥者

面向对象开发：

就是不断创建对象，使用对象，指挥对象做事情实现功能
原则：如果有对象，就指挥对象实现功能；如果没有，就创建对象，然后再指挥

面向对象语言特征：

封装（encapsulation） 信息隐蔽
继承（inheritance） 代码重用
多态（polymorphism） 灵活、接口统一

this 关键字用法：

对成员变量和局部变量进行区分
固定格式： this.数据成员;
调用类中的成员方法
固定格式： this.成员方法(实际参数列表);
调用类中的其他构造器（后面章节补充）

```
public void setId(String accountId) {
id = accountId;
}
```

成员变量与局部变量的区分：
方法的形参如果与成员变量同名
不带 this 修饰的变量指的是形参
如果要表示成员变量，则必须加 this 修饰
方法的形参与成员变量不同名
则不带 this 修饰的变量指的就是成员变量

Day07 笔记

Static

static 是一个修饰符，表示静态的意思，可以修饰属性、方法、代码块
静态成员
static 修饰类中的数据成员，该成员就成了静态数据成员，也称为类成员；

类成员，是属于类的，为这个类所有对象共享，只占用一块内存空间

static 成员特点：

被类的所有对象共享
随着类的加载而加载，先于对象存在
对象需要类被加载后，才能被创建出来
可以通过类名调用，也可以通过对象名调用，推荐使用类名调用
格式： 类名.静态数据成员;
方法区中有一块专门的区域：静态区，专门用来存储类的 static 成员

静态成员访问方式，案例描述：

静态数据成员是属于类的，并且为这个类所有对象共享，案例描述：

```
class Demo {
public static int num;
}
public static void main(String[] args) {
//可以使用类名来访问，推荐用法
Demo.num = 10;
Demo demo = new Demo();
//也可以对象来访问，但不推荐
demo.num = 20;
```

可以看出，无论是使用类访问静态属性，还是使用这个类的某个对象访问静态属性，效果是一样的，这个属性对这个类的所有对象都是可见的、共享的

静态方法

在类中，使用 static 修饰的方法，就是静态方法；
static 方法的作用，主要是用来操作 static 成员；

static 方法可以使用类名来调用，也可以使用对象来调用，但推荐使用类名。

Day08 笔记

访问控制

对象中的属性和方法，可以根据不同的权限修饰符（public > protected > default > private）来进行访问控制。
类中的属性和方法，可以使用以下四种权限修饰符进行访问控制：
public > protected > default > private
public，公共的，在所有地方都可以访问
昆山杰普软件科技有限公司 No. 41/77protected，受保护的，当前类中、子类中，同一个包中其他类中可以访问
default，默认的，当前类中、同一个包中的子类中可以访问

注意，default 默认指的是没有修饰符，并不是使用 default 关键字修饰
例如， String name; 在类中，这种情况就是默认的修饰符
private，私有的，当前类中可以访问

方法重写

1）重写应用场景
父子类继承关系中，当子类需要父类的功能，而继承的方法不能完全满足子类的需求，子类里面有特殊的功能，此时可以重写父类中的方法，这样，即沿袭了父类的功能，又定义了子类特有的内容。
2）方法重写细节
前提：父子类继承关系中
子类新增方法，和从父类继承的方法，方法名完全相同
两个方法的参数列表完全相同
重写方法的访问权限修饰符可以被扩大，但是不能被缩小
public > protected > default > private
方法的返回类型可以相同，也可以不同（暂时先按相同处理，后续再补充）
方法抛出异常类型的范围可以被缩小，但是不能被扩大（超纲内容，暂时先忽略）
一般情况下，子类进行方法重写时，最好方法的声明完全一致

结论：子类继承父类，在调用方法的时候，如果子类中没用重写，那么调用从父类继承的方法，如果子类重写了这个方法，那么调用到子类重写的方法。

重写 Object 类中 toString()

Java 中的类，如果类定义时没有指定父类，那么这个类会默认继承 Object 类
Java 中的每个类都直接或间接继承 Object 类，Object 类是 Java 继承体系中的最顶层父类

final 修饰变量

则变量就成了常量，初始化以后其值不能改变
变量是基本类型: 数据值不能发生改变
变量是引用类型: 地址值不能发生改变,但是地址里面的内容是可以发生改变的

Day09 笔记

abstract

父类 Animal 中 eat、sleep ，这两个方法的功能是不明确的，如何通过代码体现这种不明确
实际开发中，一般不会实例化 Animal 类对象，主要用 Animal 声明引用指向子类对象，如何通过代码保证 Animal 类不能实例化对象

抽象，简单可理解为不具体、高度概括的，专业描述为：抽象是一种将复杂的概念和现实世界问题简化为更易于理解和处理的表示方法。在计算机科学和编程中，抽象是一种关注问题的本质和关键特征，而忽略具体实现细节的方法。在面向对象编程中，抽象是通过定义类、接口和方法来实现的

抽象方法：
将共性的行为（方法）抽取到父类之后，发现该方法的实现逻辑无法在父类中给出具体的实现，就可以将该方法定义为抽象方法。
昆山杰普软件科技有限公司 No. 3/83 抽象类：
如果一个类中存在抽象方法，那么该类就必须声明为抽象类。
抽象类和抽象方法的关系：
使用 abstract 修饰的类就是抽象类
抽象类可以包含，也可以不包含抽象方法
包含抽象方法的类，一定要声明为抽象类
抽象类和普通类区别：
抽象类必须使用 abstract 修饰符
抽象类相对普通类，多了包含抽象方法的能力
抽象类相对普通类，失去了实例化创建对象的能力
抽象类和普通类相同点：
符合继承关系特点，能够使用多态机制
子类可以重写从抽象类继承的方法
实例化子类对象需要借助父类构造器实现父类部分的初始化

Day10 笔记

hashCode

该方法的具体细节，我们在集合章节再补充，现阶段大家大致了解即可！
该方法返回一个 int 值，该 int 值是 JVM 根据对象在内存的中的特征（地址值），通过哈希算法计算出的一个结果。Hash，一般翻译做“散列”，也可以音译为“哈希”，就是把任意长度的数据输
入，通过散列算法，变换成固定长度的输出，该输出就是散列值。
一个任意长度的输入转为一个固定长度的输出，是一种压缩映射，也就是说，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。
注意：通常情况下我们认为 Object 中 hashCode 方法返回的是对象的内存地址值，但实际上并不是
引用变量的 hashCode 值：
俩个引用变量指向同一个对象，则它们的 hashCode 值一定相等
俩个引用变量的 hashCode 值相同，则它们有可能指向同一个对象，也可能指向不同对象
俩个引用变量的 hashCode 值不同，则它们肯定不可能指向同一个对象
在自定义类中，如果需要重写 equals 方法的话，我们一般会同时重写

hashCode()，建议使用 STS 自动生成重写代码

getClass

该方法的具体细节，我们在反射章节再补充，现阶段大家大致了解即可！
该方法是非常重要的一种方法，它返回引用变量在运行时所指向的字节码对象。
该方法是 native 修饰的本地方法，不是 Java 语言实现的。
注意：子类中不能重写 getClass，调用的一定是 Object 中的 getClass 方法。

Day11 笔记

集合概述

在 Java 中，集合（Collection）是一种用于存储和操作一组对象的数据结构。它提供了一组接口和类，用于处理和操作对象的集合。
集合框架（Collection Framework）是 Java 中用于表示和操作集合的一组类和接口。它位于 java.util 包中，并提供了一系列的接口和类，包括集合接口（Collection）、列表接口（List）、集合类（Set）、映射接口（Map）等。
集合框架的主要目标是提供一种通用的方式来存储和操作对象的集合，无论集合的具体实现方式如何，用户都可以使用统一的接口和方法来操作集合。
集合和数组都可以存储多个元素值，对比数组，我们来理解下集合：
数组的长度是固定的，集合的长度是可变的
数组中存储的是同一类型的元素，集合中存储的数据可以是不同类型的
数组中可以存放基本类型数据或者引用类型变量，集合中只能存放引用类型变量
数组是由 JVM 中现有的 类型+[] 组合而成的，除了一个 length 属性 ，还有从 Object 中继承过来的方法 之外，数组对象就调用不到其他属性和方法了
集合框架由 java.util 包下多个接口和实现类组成，定义并实现了很多方法，功能强大
框架体系
集合框架主要有三个要素组成：
接口
整个集合框架的上层结构，都是用接口进行组织的。接口中定义了集合中必须要有的基本方法。
通过接口还把集合划分成了几种不同的类型，每一种集合都有自己对应的接口。
实现类
对于上层使用接口划分好的集合种类，每种集合的接口都会有对应的实现类。
每一种接口的实现类很可能有多个，每个的实现方式也会各有不同。
数据结构每个实现类都实现了接口中所定义的最基本的方法，例如对数据的存储、检索、操作等方法。但是不同的实现类，它们存储数据的方式不同，也就是使

用的数据结构不同。

集合分类：

单列集合（Single Column Collection）
根接口： java.util.Collection
单列集合是指每个集合元素只包含一个单独的对象，它是集合框架中最简单的形式
多列集合（Multiple Column Collection）
根接口： java.util.Map
多列集合是指每个集合元素由多个列（字段）组成，可以同时存储和操作多个相关的值

Day12 笔记

Collection

Collection 接口是单列集合类的父接口，这种集合可以将数据一个一个的存放到集合中。它有两个重要的子接口，分别是 java.util.List 和 java.util.Set
Collection 是父接口，其中定义了单列集合（List 和 Set）通用的一些方法，Collection 接口的实现类，都可以使用这些方法。

Collection 集合基础方法

```
package java.util;
public interface Collection<E> extends Iterable<E> {
//省略...
//向集合中添加元素
boolean add(E e)
//清空集合中所有的元素。
void clear()
//判断当前集合中是否包含给定的对象。
boolean contains(Object o)
//判断当前集合是否为空。
boolean isEmpty()
//把给定的对象，在当前集合中删除。
boolean remove(Object o)
//返回集合中元素的个数。
int size()
//把集合中的元素，存储到数组中。
Object[] toArray()
```

HashSet

java.util.HashSet 是 Set 接口的实现类，它使用哈希表（Hash Table）作为其底层数据结构来存储数据。
HashSet 特点：

无序性：HashSet 中的元素的存储顺序与插入顺序无关
HashSet 使用哈希表来存储数据，哈希表根据元素的哈希值来确定元素的存储位置，而哈希值是根据元素的内容计算得到的，与插入顺序无关。
唯一性：HashSet 中不允许重复的元素，即每个元素都是唯一的
允许 null 元素：HashSet 允许存储 null 元素，但只能存储一个 null 元素，HashSet 中不允许重复元素
高效性：HashSet 的插入、删除和查找操作的时间复杂度都是 O(1)
哈希表通过将元素的哈希值映射到数组的索引来实现快速的插入、删除和查找操作。

元素插入过程：
当向 HashSet 中插入元素时，先获取元素的 hashCode()值，再检查 HashSet 中是否存在相同哈希值的元素，如果元素哈希值唯一，则直接插入元素
如果存在相同哈希值的元素，会调用元素的 equals()方法来进一步判断元素是否是相同。如果相同，则不会插入重复元素；如果不同，则插入

Day13 笔记

泛型

泛型（Generics）的概念是在 JDK1.5 中引入的，它的主要目的是为了解决类型安全性和代码复用的问题。
泛型是一种强大的特性，它允许我们在定义类、接口和方法时使用参数化类型 MyClass 是一个泛型类，使用类型参数 T 。我们可以在创建对象时指定具体的类型，例如 MyClass<Integer> 或 MyClass<String> 。
泛型能够使我们编写出来通用的代码，提高代码的可读性和重用性。通过使用泛型，我们可以在类、接口和方法中使用类型参数，使得代码可以处理不同类型的数据，同时保持类型安全。
Collection 是一个泛型接口，泛型参数是 E，add 方法的参数类型也是 E 类型。
在使用 Collection 接口的时候，给泛型参数指定了具体类型，那么就会防止出现类型转换异常的情况，因为这时候集合中添加的数据已经有了一个规定的类型，其他类型是添加不进来的。
例如下面案例中，我们指定了集合 c 只能存储 String 类型数据，则 Integer 类型的 1 就无法添加成功。
可以看出，传入泛型参数后，add 方法只能接收 String 类型的参数，其他类型的数据无法添加到集合中，同时在遍历集合的时候，也不需要我们做类型转换了，直接使用 String 类型变量接收就可以了，JVM 会自动转换的
Collection<String> c = new ArrayList<String>();
可简写为菱形泛型形式：
Collection<String> c = new ArrayList<>();
菱形泛型（Diamond Operator）是 JDK7 中引入的一种语法糖，用于简化泛型的类型推断过程
元注解

在我们进行自定义注解的时候，一般会使用到元注解，来设置自定义注解的基本特点。所以，元注解也就是对注解进行基本信息设置的注解。
常用的元注解：
@Target，用于描述注解的使用范围，例如用在类上面还是方法上面
@Retention，用于描述注解的保存策略，是保留到源代码中、Class 文件中、还是加载到内存中
@Documented，用于描述该注解将会被 javadoc 生产到 API 文档中
@Inherited，用于表示某个被标注的类型是被继承的，如果一个使用了@Inherited 修饰的 annotation 类型被用于一个 class，则这个 annotation 将被用于该 class 的子类
保持
类中使用的注解，根据配置，可以保持到三个不同的阶段：
SOURCE，注解只保留在源文件，当 Java 文件编译成 class 文件的时候，注解被遗弃
CLASS，注解被保留到 class 文件，但 jvm 加载 class 文件时候被遗弃
RUNTIME，注解不仅被保存到 class 文件中，jvm 加载 class 文件之后，仍然存在
@Override 功能解析：
编译器在编译 Person 类 的时候，会读取到 toString 方法上的注解 @Override ，从而帮我们检查这个方法是否是重写父类中的，如果父类中没有这个方法，则编译报错。
注解和注释的区别：
注解是给其他程序看的，通过参数的设置，可以在编译后 class 文件中【保留】注解的信息，其他程序读取后，可以完成特定的操作
注释是给程序员看的，无论怎么设置，编译后 class 文件中都是【没有】注释信息，方便程序员快速了解代码的作用或结构
类型擦除
泛型类型仅存在于编译期间，编译后的字节码和运行时不包含泛型信息，所有的泛型类型映射到同一份字节码。
由于泛型是 JDK1.5 才加入到 Java 语言特性的，Java 让编译器擦除掉关于泛型类型的信息，这样使得 Java 可以向后兼容之前没有使用泛型的类库和代码，因为在字节码（class）层面是没有泛型概念的。
1. 元数据信息：注解本身不会直接影响程序的执行，而是提供了一种用于存储和传递元数据的方式。元数据是关于程序中元素的描述信息，可以包含名称、类型、范围、约束等信息，用于辅助程序的开发、编译和执行。
2. 内置注解：Java 提供了一些内置的注解，例如 @Override 、@Deprecated 、 @SuppressWarnings 等。这些注解用于提供关于方法覆盖、过时方法、警告抑制等信息，编译器和工具可以根据注解来执行相应的处理。
3. 自定义注解：Java 也允许开发者自定义注解，通过 @interface 关键字定义新的注解类型。自定义注解可以包含元素（成员变量），可以为这些元素指定默认值，并可以在程序中使用注解，并提取注解中的元素值。
4. 应用领域：注解在 Java 中被广泛应用于各个领域，例如框架开发、测试工具、持久化框架、Web 开发等。通过注解，可以提供更丰富的配置和行为控

制，简化了代码的编写和配置。
5. 本章对于注解的掌握仅限于元数据信息的认识、内置注解的认识等，在后期学习反射的时候会完善自定义注解的学习，并且使用反射机制获取注解并进行操作。

Day14 笔记

异常

程序在运行过程中，由于意外情况导致程序发生异常事件，默认情况下发生的异常会中断程序的运行。
在 Java 中，把常见的异常情况，都抽象成了对应的异常类型，那么每种异常类型都代表了一种特定的异常情况。
当程序中出现一种异常情况时，也会创建并抛出一个异常类型对象，这个对象就表示当前程序所出现的问题。
异常体系
异常体系中的根类是： java.lang.Throwable ，该类下面有俩个子类型，java.lang.Error 和 java.lang.Exception
注意， Throwable 表示可以被抛出的
Error ，表示错误情况，一般是程序中出现了比较严重的问题，并且程序自身并无法进行处理。
Exception ，表示异常情况，程序中出了这种异常，大多是可以通过特定的方式进行处理和纠正的，并且处理完了之后，程序还可以继续往下正常运行
异常种类
我们平时使用的异常类型，都是 Exception 类的子类型，它们把异常划分成了俩种：
编译时异常
运行时异常
编译时异常
继承自 Exception 类，也称为 checked exception
编译器在编译期间，会主动检查这种异常，如果发现异常则必须显示处理，否则程序就会发生错误，无法通过编译
运行时异常
RuntimeException 类及其子类，也称为 unchecked exception
编译器在编译期间，不会检查这种异常，也不要求我们去处理，但是在运行期间，如果出现这种异常则自动抛出
异常传播
如果一个方法中出现了异常的情况，系统默认的处理方式是：自动创建异常对象，并将这个异常对象抛给当前方法的调用者，并一直向上抛出，最终传递给 JVM，JVM 默认处理步骤有 2 步：
把异常的名称，错误原因及异常出现的位置等信息输出在了控制台
程序停止执行

finally 语句
finally 关键字可以和 try、catch 关键字一起使用，固定搭配为： try catch-finally ，它可以保证指定 finally 中的代码一定会执行，无论是否发生异常！
固定格式：
finally 块的主要作用：
资源释放：在 try 块中打开的资源（例如文件、数据库连接、网络连接等）可以在 finally 块中关闭或释放，以确保资源的正确释放，即使在发生异常的情况下也能够执行释放操作。
清理操作： finally 块可以用于执行一些清理操作，例如关闭打开的流、

释放锁、取消注册监听器等。

Day15 笔记

多线程

进程

进程指一个内存中运行的应用程序，它是系统运行程序的基本单位。
一个程序从创建、运行到消亡，这样整个过程就是一个进程。
一个操作系统中可以同时运行多个进程，每个进程运行时，系统都会为其分配独立的内存空间。
线程
线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程，一个进程中也可以有多个线程，此时这个应用程序就可以称之为多线程程序。
结论：一个程序运行后至少有一个进程，一个进程中可以包含一个(main 线程)或多个线程！
并发并行
并发：指两个或多个事件在同一个时间段内发生
线程的并发执行，是指在一个时间段内（微观），俩个或多个线程，使用同一个 CPU 交替运行。
并行：指两个或多个事件在同一时刻发生（同时发生）
线程的并行执行，是指在同一时刻，俩个或多个线程，各自使用一个 CPU 同时运行。
线程调度
时间片
并发多线程只有一个 CPU，某个微观时刻，当指定线程拥有 CPU 的使用权，则该线程代码就可以执行，而其他线程阻塞等待。
一个线程不可能一直拥有 CPU 的使用权，不可能一直执行下去，它拥有 CPU 执行的时间是很短的，微秒纳秒级别，这个时间段我们就称之为 CPU 时间片。
线程执行时如果一个时间片结束了，则该线程就会停止运行，并交出 CPU 的使

用权，然后等待下一个 CPU 时间片的分配。
在宏观上，一段时间内，我们感觉俩个线程在同时运行代码，其实在微观中，这俩个线程在使用一个 CPU 的时候，它们是交替着运行的，每个线程每次都是运行一个很小的时间片，然后就交出 CPU 使用权，只是它们俩个交替运行的速度太快了，给我们的感觉，好像是它们俩个线程在同时运行。
线程创建
java.lang.Thread 是 java 中的线程类，所有线程对象都必须是 Thread 类或其子类的实例。
每个线程的作用，就是完成我们给它指定的任务，实际上就是执行一段我们指定的代码。我们只需要在 Thread 类的子类中重写 run 方法，完成相应的功能
线程名称
默认线程名：
不管是主线程，还是我们创建的子线程，都是有名字的。默认情况下，主线程的名字为 main ，main 线程中创建出的子线程，它们名字命名规则如下：
其中， "Thread-" + nextThreadNum() 就是在拼接出这个线程默认的名字，
比如第一个子线程 Thread-0，第二个为 Thread-1，第三个为 Thread-2，以此类推。
获取当前线程对象：
public static native Thread currentThread();
注意，这里说的当前线程，指的是执行当前方法的线程。
获取线程名：
public final String getName();
常见用法：
String name = Thread.currentThread().getName();
//JavaAPI-Thread 构造器源码
public Thread() {
init(null, null, "Thread-" + nextThreadNum(), 0);
设置线程名：
通过线程对象设置线程名
public final synchronized void setName(String name);
创建对象时，设置线程名
public Thread(String name);
public Thread(Runnable target, String name);

Day16 笔记

sleep 方法

线程类 Thread 中的 sleep 方法：
//JavaAPI-Thread 源码
public class Thread implements Runnable {
public static native void sleep(long millis) throws
InterruptedException;

该静态方法可以让当前执行的线程暂时休眠指定的毫秒数。
}

join 方法

Thread 类中 join 方法：

```
//JavaAPI-Thread 源码
public class Thread implements Runnable {
//
public final synchronized void join(long millis)throws
InterruptedException{
//.
}
//
public final void join() throws InterruptedException{
/
}
```

作用：
join 方法，可以让当前线程阻塞，等另一个指定线程运行结束后，当前线程才可以继续运行
join() 一直等待到线程结束，死等
join(long millis) 只等待参数毫秒，时间到了后，继续运行
状态转换：
线程执行了 join()方法后，会从 RUNNABLE 状态进入到 WAITING（无限期等待）状态
线程执行了 join(long million)方法后，会从 RUNNABLE 进入到 TIMED_WAITING（有限期等待）状态

线程安全

如果有多个线程，它们在一段时间内，并发访问堆区中的同一个变量（含写入操作），那么最终可能会出现数据和预期结果不符的情况，这种情况就是线程安全问题。
我们经常会描述：这段代码是线程安全的，那段代码是非线程安全的。 其实就是在说，这段代码在多线程并发访问的环境中，是否会出现上述情况。

线程同步

Java 中提供了线程同步的机制，来解决上述的线程安全问题。
Java 中实现线程同步，主要借助 synchronized 关键字实现。
线程同步方式：
同步代码块
同步方法
锁机制

Day17 笔记

```
package java.io;
public class File
implements Serializable, Comparable<File>
{
//通过将给定路径名字符串来创建新的 File 实例
public File(String pathname) {
if (pathname == null) {
throw new NullPointerException();
}
this.path = fs.normalize(pathname);
this.prefixLength = fs.prefixLength(this.path);
}
//从**父抽象路径名和子路径名字符串**创建新的 File 实例
public File(String parent, String child) {
//省略...
}
public File(File parent, String child) {
1. 一个 File 对象代表硬盘中实际存在的一个文件或者目录。
2. 无论该路径下是否存在文件或者目录，都不影响 File 对象的创建。
路径操作
绝对路径：从盘符开始的路径，这是一个完整的路径。
相对路径：相对于项目目录的路径，这是一个便捷的路径，开发中经常使
用
package com.briup.chap11.test;
public class Test016_File {
public static void main(String[] args) {
//1.准备目录文件
String dirPath = "D:\\test";
File dirFile = new File(dirPath);
//2.获取目录中所有子文件名称，并遍历输出
String[] list = dirFile.list();
for (String s : list) {
System.out.println(s);
}
System.out.println("----------");
//3.准备普通文件
String fileName = "readme.pdf";
File file = new File(dirFile,fileName);
//4.普通文件调用 list 返回 null
String[] list2 = file.list();
昆山杰普软件科技有限公司 No. 9/85 输出结果：
System.out.println(list2);
```

```
System.out.println("----------");
//5.获取目录中所有子文件对象，并遍历输出
File[] listFiles = dirFile.listFiles();
for (File f : listFiles) {
System.out.println(f);
}
System.out.println("----------");
//6.使用文件过滤器，获取目录下所有普通文件，并遍历输出
File[] listFiles2 = dirFile.listFiles(new FileFilter()
{
@Override
public boolean accept(File f) {
if(f.isFile())
return true;
return false;
}
});
for (File f : listFiles2) {
//输出文件名即可
System.out.println(f.getName());
}
}
}
```

Day18 笔记

File 类

java.io.File 类是文件和目录路径名的抽象表示，主要用于文件和目录的创建、查找和删除等操作。

IO 流

内存输出流
使用文件流，我们可以操作文件中的数据。
使用内存流，我们可以操作内存中字节数组中的数据。
内存字节流，也称为字节数组流，主要有下面两种：
java.io.ByteArrayOutputStream
内存输出流，负责把数据写入到内存中的字节数组中

```
// 3.将字符串转换为字节数组，并追加到 a.txt 文件尾
byte[] arr = line.getBytes();
os.write(arr);
// 4.关闭资源
os.close();
```

}
}
内存输入流，负责从内存中的字节数组中读取数据
如果要操作文件，则需要使用文件流
如果要操作内存中的数据，则可以选择内存流
内存字节流提供了一种在内存中进行数据读取和写入的便捷方式。内存流在实际开发中有专门的应用场景，我们使用不多，了解即可。
以下是一些常见的应用场景：
1. 数据的临时存储：内存字节流可以将数据暂时存储在内存中的字节数组中，而不需要写入到磁盘或网络中。这在一些临时性的数据处理场景中非常有用，例如在内存中对数据进行加密、解密、压缩、解压缩等操作。
2. 数据的转换：内存字节流可以用于将数据从一种格式转换为另一种格式。例如，可以将一个对象序列化为字节数组，然后再将字节数组反序列化为对象。这在一些需要将数据在内存中进行格式转换的场景中非常有用。
3. 测试和调试：内存字节流可以用于测试和调试目的，例如模拟输入流或输出流的行为。通过将数据写入内存字节流，可以方便地检查数据的内容和格式，而无需依赖外部资源。
4. 单元测试：内存字节流在单元测试中也非常有用。可以使用内存字节流模拟输入和输出流，以便在测试中验证代码的正确性和可靠性，而无需依赖外部文件或网络连接。
需要注意的是，由于内存字节流将数据存储在内存中的字节数组中，因此在处理大量数据时可能会占用较多的内存。在这种情况下，需要谨慎使用内存字节流，以避免内存溢出的问题

Day19 笔记

标准流

在 Java 中，标准流（Standard Streams）是指三个预定义的流对象，用于处理标准输入、标准输出和标准错误。这些标准流在 Java 中是自动创建的，无需显式地打开或关闭。
以下是 Java 中的标准流：
1. 标准输入流（System.in）
它是一个字节流（InputStream），用于从标准输入设备（通常是键盘）读取数据。可以使用 Scanner 或 BufferedReader 等类来读取标准输入流的数据。
2. 标准输出流（System.out）
它是一个字节流（PrintStream），用于向标准输出设备（通常是控制台）输出数据。可以使用 System.out.println() 或 System.out.print() 等方法来输出数据到标准输出流
也是一个字节流（PrintStream），用于向标准错误设备（通常是控制台）输出错误信息。与标准输出流相比，标准错误流通常用于输出错误、警告或异常信息。
PrintStream 打印流：

PrintStream 是一个字节流，也是一个增强流，相对于普通节点流，它提供了一系列便捷的方法，可以方便地输出各种数据类型的值到输出流。
它提供了一系列的 print 和 println 方法，可以输出各种数据类型的值，并自动转换为字符串形式。通常用于向标准输出流（System.out）输出数据。

转换流

1） 字符编码和字符集
计算机中储存的信息都是用二进制数表示的，而我们在屏幕上看到的数字、英文、标点符号、汉字等字符是二进制数转换之后的结果。
按照某种规则，将字符存储到计算机中，称为编码 。然后将存储在计算机中的二进制数按照某种规则解析显示出来，称为解码 。
比如说，按照 A 规则存储，同样按照 A 规则解析，那么就能显示正确的文本符号。但如果按照 A 规则存储，按照 B 规则解析，就会导致乱码现象。
字符编码 Character Encoding : 一套自然语言字符与二进制数（码点）之间的对应规则。
字符集 Charset ：也叫编码表。是一个系统支持的所有字符的集合，包括各国家文字、标点符号、图形符号、数字等。
计算机要准确的存储和识别各种字符集符号，需要进行字符编码，一套字符集必然至少有一套字符编码。常见字符集有 ASCII 字符集、GBK 字符集、Unicode 字符集等
可见，当指定了编码，它所对应的字符集自然就指定了，所以编码才是我们最终要关心的。
2） 编码引出的问题
使用 FileReader 读取项目中的文本文件。由于软件的设置，都是默认的 UTF-8 编码，所以没有任何问题。但是，当读取 Windows 系统中创建的文本文件时，由于 Windows 系统的默认是 GBK 编码，就会出现乱码

Day20 笔记

IP 地址

全称”互联网协议地址”，也称 IP 地址。是分配给上网设备的数字标签。常见的 IP 分类为：ipv4 和 ipv6
IPv4：共 32 位，表示范围 43 亿左右，一般使用点分 4 段表示法
IPv6：共 128 位，表示范围更大，号称可以为地球上每一粒沙子分配一个 IP
IP 域名：
本质上也是一个 IP 地址，可读性更好，更容易记忆，需要使用 dns 域名解析服务器解析
公网、内网 IP：
公网 IP，是可以连接互联网的 IP 地址
内网 IP，也叫局域网 IP，只能组织机构内部使用
192.168.开头的就是常见的局域网地址，范围即为 192.168.0.0--192.168.255.255，专门为组织机构内部使用
DOS 常用命令：
ipconfig：查看本机 IP 地址

ping IP 地址：检查网络是否连通
特殊 IP 地址：
127.0.0.1：是回送地址，可以代表本机地址，一般用来测试使用

端口号

端口号用来标记正在计算机设备上运行的应用程序，其范围是 0~65535。其中，0~1023 之间的端口号用于系统内部的使用，我们自己普通的应用程序要使用 1024 以上的端口号即可，同时也要避免和一些知名应用程序默认的端口冲突，例如：oracle 启动后默认占用端口号 1521，mysql 启动后默认占用端口号 3306，redis 启动后默认占用端口号 6379，tomcat 启动后默认占用端口号 8080。
端口分类：
周知端口
0~1023，被预先定义的知名应用占用，如: HTTP 占用 80，FTP 占用 21，MySQL 占用 3306
注册端口
1024~49151，分配给用户进程或某些应用程序
动态端口
49152 到 65535，之所以称为动态端口，是因为它 一般不固定分配某种进程，而是动态分配
注意事项：
自己开发的程序一般选择使用注册端口
同一时刻一个设备中两个程序的端口号不能重复，否则出错
OSI（Open System Interconnect），即开放式网络互连标准。 一般叫 OSI 参考模型，是 ISO（国际标准化组织）在 1985 年研究的网络互连模型，它共包含七层，具体可参考下图。
疑问：为什么要分层？
要解决计算机网络中数据的传输，涉及的问题很多很复杂，分层可以将大问题拆分成小问题，更利于问题的解决。
TCP/IP 网络模型，是事实上的国际标准，它被简化为了四个层，从下到上分别依次是应用层、传输层、网络层、网络接口层。
应用层：主要负责应用程序的协议，例如 HTTP 协议、FTP 协议等。
传输层：主要使网络程序进行通信，在进行网络通信时，可以采用 TCP 协议，也可以采用 UDP 协议。
网络层：网络层是整个 TCP/IP 协议的核心，它主要用于将传输的数据进行分组，将分组数据发送到目标计算机或者网络。
链路层：链路层是用于定义物理传输通道，通常是对某些网络连接设备的驱动协议，例如针对光纤、网线提供的驱动。

TCP 和 UDP

UDP，用户数据报协议(User Datagram Protocol) （了解）
UDP 是无连接通信协议，在数据传输时，数据的发送端和接收端不建立连接，也不能保证对方能接收成功。
例如，当一台计算机向另外一台计算机发送数据时（UDP），发送端不会确认接收端是否存在，就会直接发出数据，同样接收端在收到数据时，也不会向发送端反馈是否收到数据。
由于使用 UDP 协议消耗资源小，通信效率高，所以通常都会用于音频、视频

和普通数据的传输，因为这种情况即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。
但是在传输重要数据时，不建议使用 UDP 协议，因为它不能保证数据传输的完整性。
TCP，传输控制协议 (Transmission Control Protocol) （重要）
TCP 协议是面向连接的通信协议，即传输数据之前，在发送端和接收端建立连接，然后再传输数据，它提供了两台计算机之间可靠的、无差错的数据传输。
在 TCP 连接中，将计算机明确划分为客户端与服务器端，并且由客户端向服务端发出连接请求，每次连接的创建都需要经过“三次握手”的过程，四次挥手断开连接
TCP 的三次握手：
TCP 协议中，在发送数据的准备阶段，客户端与服务器之间的三次交互，以保证连接的可靠
第一次握手，客户端向服务器端发出连接请求，等待服务器确认
第二次握手，服务器端向客户端回送一个响应，通知客户端收到了连接请求
第三次握手，客户端再次向服务器端发送确认信息，确认连接
TCP 的四次挥手：
用于断开连接

Day21 笔记

如果一个类加载器收到类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，最终加载请求会到达顶层的启动类加载器 Bootstrap ClassLoader 。
如果顶层类加载器可以完成加载任务，则进行 class 文件类加载，加载成功后返回。如果当前类加载器无法加载，则向下委托给子类加载器，此时子类加载器才会尝试加载，成功则返回，失败则继续往下委托，如果所有的加载器都无法加载该类，则会抛出 ClassNotFoundException，这就是双亲委托机制。
反射
4.1 反射概述
Java 反射机制是指在 Java 程序在运行状态下，动态地获取、检查和操作类的信息和对象的能力。
反射机制作用：
对于任意一个类，都能够知道这个类的所有属性和方法
对于任意一个对象，都能够调用它的任意一个方法和属性
这种动态获取信息以及动态调用对象方法的功能称为 Java 语言的反射机制。
当一个类被使用的时候，类加载器会把该类的字节码文件装入内存（类加载），同时在堆空间创建一个 字节码对象（Class 类对象） ，这个对象是 Java 反射机制的核心，它包含了一个类运行时信息。
4.2 反射核心类

在 Java 中， Class 类是一个重要的核心类，它用于表示一个类或接口的运行时信息。每个类在 Java 虚拟机中都有一个对应的 Class 对象，可以通过该对象获取类的构造函数、方法、属性等信息，并且可以进行实例化对象、方法调用和数据成员访问等操作。

Class 核心类 JavaSE 源码：

```
package java.lang;
//字节码类
public final class Class<T> implements java.io.Serializable,
GenericDeclaration,
Type,
AnnotatedElement {
//省略...
//获取类的所有构造方法
@CallerSensitive
public Constructor<?>[] getConstructors() throws
SecurityException {
checkMemberAccess(Member.PUBLIC,
Reflection.getCallerClass(), true);
return
copyConstructors(privateGetDeclaredConstructors(true));
}
//获取类的所有数据成员
@CallerSensitive
public Field[] getFields() throws SecurityException {
checkMemberAccess(Member.PUBLIC,
Reflection.getCallerClass(), true);
return copyFields(privateGetPublicFields(null));
}
//获取类的所有成员方法
@CallerSensitive
public Method[] getMethods() throws SecurityException {
checkMemberAccess(Member.PUBLIC,
Reflection.getCallerClass(), true);
return copyMethods(privateGetPublicMethods());
}
}
```

在 Java 反射中， Class 、 Constructor 、 Method 和 Field 是表示类的不同部分的关键类。它们提供了访问和操作类的构造函数、方法和字段的方法。

Class 类：表示一个类的运行时信息。通过 Class 类可以获取类的构造函数、方法和字段等信息。可以使用 Class.forName() 方法获取一个类的 Class 对象，也可以通过对象的 getClass() 方法获取其对应的 Class 对象。

Constructor 类：表示一个类的构造函数。通过 Constructor 类可以创建类的实例。可以使用 Class 对象的 getConstructors() 或

getConstructor() 方法获取构造函数的对象。
Method 类：表示一个类的方法。通过 Method 类可以调用类的方法。可以使用 Class 对象的 getMethods() 或 getMethod() 方法获取方法的对象。
Field 类：表示一个类的字段。通过 Field 类可以访问和修改类的字段的值。可以使用 Class 对象的 getFields() 或 getField() 方法获取字段的对象。
字节码对象
JVM 虚拟机对类进行加载时，会在堆空间创建一个 字节码对象（Class 类对象） 。
通过该字节码对象程序员可以获取类的构造函数、方法、属性等信息，并且可以进行实例化、方法调用和属性访问等操