□错

StringBuilder

- □种容器，提高字符串拼接效率

```
StringBuilder sb = new StringBuilder();
// □入数据
sb.append("aa");
// 反转数据
sb.reverse();
// 转换为字符串
String str = sb.toString();
```

ArrayList

- 集合，只能存引用数据类型，不能存基本数据类型(如int，double等)，要存需要用其对应的包装类**(如Integer,Character)**
- 可以自动扩容
- 成员方法

```
// 定义 只能存String类型
ArrayList<String> list = new ArrayList<String> ();
// 添加元素
list.add("1234");
// 删除元素
list.remove("1234");
// 删除指定索引元素，返回被删除元素
String str = list.remove(1);
// 获取指定索引的元素
String str = list.get(1);
// 修改指定索引下的元素，返回原来的元素
String str = list.set(1,"3445");
// 获取集合长度
int size = list.size();
```

static

- 静态修饰符，可以修饰成员变量和成员方法，修饰后变量和方法可以不用创建实例直接被调用，优先于对象存在
- 静态方法只能访问静态变量和静态方法
- 非静态方法可以访问静态变量或者静态方法，也可以访问非静态的成员变量和非静态成员方法
- 静态方法中无**this**关键字

继承

- 关键字：**extends**

  `public class Student extends Person{}`

- Student称为子类，Person称为父类

- java只支持单继承：一个子类只能继承一个父类，但支持多层继承
- 子类只能访问父类中非私有的成员，且不能继承父类的
    - 构造方法
    - 被private、static、final修饰的方法
- 子类不能继承父类构造方法，但是可以通过super()调用，且默认先访问父类中无参的构造方法，再执行自己

虚方法表

- 父类中未被**private、static、final**修饰的方法
- 会一级一级添加虚方法，完善虚方法表

super

出现重名变量

- `System.out.println(name)`：从局部位置往上找
- `System.out.println(this.name)`：从子类位置往上找
- `System.out.println(super.name)`：从父类位置往上找

方法重写

- 在子类中重写父类方法，重写后，在子类虚方法表中会覆盖父类的方法
- 在子类重写的方法前要加上 `@Override`

```
@Override // 重写注解
public void eat(){
    sout('我在吃饭');
}
```

多态

本质上就是用父类定义类型，子类进行实例化

- 调用成员变量：编译看左边，运行看左边
    - 找的是父类的成员变量
- 调用成员方法：编译看左边，运行看右边
    - 找的是子类中被重写的方法

```
Animal a = new dog();
System.out.println(a.name); // 打印的是Animal类中的name
a.show(); // 打印的是dog中被重写的方法show
```

优势

- 使用父类型作为参数，可以接受所有子类对象

弊端

- 不能使用子类特有功能，因为编译器会去父类找对应的方法，找不到就会报错，只能调用子类中重写的方法
- 用 `instanceof` 关键字进行判断

```
Animal a = new dog();
Dog d = (Dog) a;// 强制转换
if (a instanceof Dog)// 用instanceof进行判断
```

final

- 用final修饰的方法不能被重写
- 用final修饰的变量只能被赋值一次
- 用final修饰的类不能被继承

抽象类

- 使用 **`abstract`** 关键字修饰的类，称为抽象类。
- 抽象类不能直接实例化，但可以作为父类，让子类继承并实现其中的抽象方法。
- 抽象类可以包含抽象方法（没有方法体的方法）和普通方法（有方法体的方法）。
- 抽象类不一定包含抽象方法，但包含抽象方法的类一定是抽象类。

```
// 抽象类定义
abstract class Animal {
    String name; // 成员变量

    // 抽象方法（没有方法体）
    public abstract void eat();

    // 普通方法（可以有方法体）
    public void sleep() {
        System.out.println(name + " 在睡觉");
    }

    // 构造方法（用于初始化）
    public Animal(String name) {
        this.name = name;
    }
}

// 继承抽象类，并实现抽象方法
class Dog extends Animal {
    public Dog(String name) {
        super(name); // 调用父类的构造方法
    }

    @Override
    public void eat() {
        System.out.println(name + " 在吃狗粮");
    }
}
```

接口

- 接口（Interface）是 Java 中的一种特殊类，用于定义一组方法规范，但不提供具体实现
- 接口不能实例化，需要由类 `implements`（实现）后，提供具体实现
- 作用：接口用于实现多态和解耦，使代码更加灵活、可扩展

  使用 `interface` 关键字定义接口
- 接口中的方法默认是 **`public abstract`**（公共的、抽象的），可以省略这两个关键字
- 接口中的变量默认是 **`public static final`**（公共的、静态的、常量），必须赋值
- 类可以实现多个接口

```java
// 定义接口
public interface Animal {
    void eat();
}

// 定义接口
public interface Fly {
    void fly();
}

// 具体类实现接口
public class Bird implements Animal, Fly {
    @Override
    public void eat() {
        System.out.println("鸟在吃东西");
    }

    @Override
    public void fly() {
        System.out.println("鸟在天空中飞翔");
    }
}

// 测试
public class Main {
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.eat();  // 输出：鸟在吃东西
        bird.fly();  // 输出：鸟在天空中飞翔
    }
}
```

内部类

- 定义在类里面的类
- 内部类可以访问外部类成员（包括私有）

```
// 定义内部类
class car{
    string name;
    class enigne{
        string name;
    }
}

car.enigne e = new car().new enigne(); // 实例化engine类
System.out.println(e.name);
```

成员内部类

- 定义在类里面成员区域的类

获取成员内部类两种方法

- 外部类编写方法对外提供内部类

    -
      ```
      class Outer{
          string name;
          class Inner{
              string name;
          }
          public Inner getInner(){
              return new Inner(); // 返回的是Inner的地址
          }
      }
      // 用new Outer()直接调用它下面的成员方法
      Outer.Inner oi = new Outer().getInner();
      ```
- 直接创建（内部类私有时失效）
    - 格式：外部类名.内部类名 对象名 = 外部类对象.内部类对象
    - 例子：`Outer.Inner oi = new Outer().new Inner();`

静态内部类

- 用 `static` 修饰的成员内部类
- 只能访问外部类的静态方法和静态属性，要访问非静态成员要创建对象
- 创建静态内部类
    - 格式：外部类.内部类 对象名 = new 外部类名.内部类名()
    - 例子：`Outer.Inner oi = new Outer.Inner()`
- 调用静态方法
    - 格式：外部类.内部类.方法名()
    - 例子：`Outer.Inne.show()`

```
class Outer {
    private String name; // String 关键字首字母大写
    private static int age; // age 需要是 static 才能被静态内部类访问
```

```
    // 静态内部类
    static class Inner {
        private String name; // String 关键字修正

        // 获取外部类的静态变量 age
        public void getOuterAge() {
            System.out.println(age); // 直接访问静态变量
        }

        // 获取外部类的非静态变量 name
        public void getOuterName() {
            System.out.println(new Outer().name); // 需要先创建 Outer 的实例
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner();
        inner.getOuterAge();  // 输出 age（默认值 0）
        inner.getOuterName(); // 输出 Outer 的 name（默认值 null）
    }
}
```

局部内部类

- 将类定义在成员方法里面
- 外界无法直接使用，需要在方法内部创建对象并使用
- 可以访问外部类成员和方法里的局部变量

```
class Outer{
    int b = 20;
    public void show(){
        int a = 10;
        // 局部内部类
        class Inner{
            String name;
            int age;
            public void method1(){
                System.out.println(a);
                System.out.println(b);
            }
         // 创建实例
         Inner i = new Inner();
         System.out.println(i.name);
         System.out.println(i.age);
        }
    }
}
```

匿名内部类

在创建类的对象的时候直接重写方法，本质上是一个子类

```java
public class abstract Animal{
    public abstract void cry();
}


Animal a = new Animal(){
  @Override
    public void cry(){
        System.out.println("猫在叫---");
    }
};
a.cry();
```

API

String

- 以"..."方式写出的字符串对象，会存储到字符串常量池，相同内容的字符串只存一份
- 通过new方式创建字符串对象，每**new**一次都会产生一个新的对象放在堆内存中

System

静态方法

方法名	说明
public static void exit(int status)	终止当前运行的Java虚拟机
public static long currentTimeMillis()	返回当前系统的时间毫秒值形式
public static void arraycopy(数据源数组,起始索引,目的地数组,起始索引,拷贝个数)	数组拷贝

`status` 是退出状态码，一般有以下几种常见值：

- `0` ：正常退出（通常代表程序运行成功）
- `1` ：非正常退出（通常表示一般性错误）
- `2` ：错误的命令行参数
- `-1` ：非正常退出（通常表示未知错误）

```java
import java.util.Arrays;

public class SystemAPIExample {
    public static void main(String[] args) {
        // 1. 获取当前时间
        long startTime = System.currentTimeMillis();
        System.out.println("程序开始时间: " + startTime + " 毫秒");

        // 2. 数组拷贝示例
```

```
        int[] src = {1, 2, 3, 4, 5};  // 源数组
        int[] dest = new int[5];      // 目标数组

        System.arraycopy(src, 1, dest, 2, 3);
        // 从 src[1] 开始（即 2），复制 3 个元素到 dest[2] 开始的位置

        System.out.println("拷贝后的数组：" + Arrays.toString(dest));

        // 3. 结束程序
        int status = 1;  // 这里可以改成不同的状态码
        System.out.println("程序即将退出，状态码：" + status);
        System.exit(status);

        // 这行代码不会执行，因为 System.exit() 终止了 JVM
        System.out.println("这行不会被打印");
    }
}
```

Runtime

方法	说明
`public static getRuntime()`	获取 `Runtime` 实例（单例模式）
`public long maxMemory()`	返回 JVM 可用的最大内存
`public long totalMemory()`	返回 JVM 已分配的总内存
`public long freeMemory()`	返回 JVM 的空闲内存
`public Process exec(String command)`	运行外部程序
`public int availableProcessors()`	获取可用的 CPU 核心数
`public void exit(int status)`	终止 JVM，`0` 表示正常退出

Object

方法	说明
`public String toString()`	返回对象的字符串表现形式
`public boolean equals(Object obj)`	比较两个对象是否相等
`protected Object clone(int a)`	对象克隆

- 对于 `toSring` 和 `equals` 方法，在所有类中都可以重写这两个方法，因为所有类的父类都是 **`Object`**
- 对于 `clone` 方法也可以重写，但是要重写 `Clonaable` 接口并重写 `clone` 方法才能使用，并且 `Object` 的克隆方法默认是浅克隆
    - 浅克隆：对于引用数据类型，拷贝地址值
    - 深克隆：对于引用数据类型，拷贝地址存的值

BigInteger

方法名	说明
`public BigInteger(int num, Random rnd)`	获取随机大整数，范围为[0~2的num次方-1]
`public BigInteger(String val)`	获取指定的大整数
`public BigInteger(String val,int radix)`	获取指定进制的大整数

函数式编程

Lambda

- **Lambda**表达式只能替代函数式接口的匿名内部类
- 函数式接口是有且仅有一个抽象方法的接口，注解为**FunctionalInterface**

```
interface Swim{
    void swimming();
}
// 匿名内部类
Swim s1 = new Swim(){
    @Override
    public void swimming(){
        System.out.println("我正在游泳");
    }
}

/*
lambda表达式
(被重写方法的形参列表)->{
    被重写方法的方法体代码
}
*/

// 可以简化为
Swim s1 = ()->{
        System.out.println("我正在游泳");
}
```

还能继续简化，有如下规则：

- 参数类型可以省略不写
- 如果只有一个参数，可以同时省略参数类型和"()"，多个参数不可以
- 如果Lambda表达式只有一行代码，大括号可以不写，同时要省略分号";"；如果这行代码是**return**语句，必须去掉**return**

```
// 有一个JButton类型的按钮，为btn
btn.addActionListener(new ActionListener(ActionEvent e) ->{
    System.out.println("登录成功");
})
// 层层递进

btn.addActionListener((ActionEvent e) ->{
```

```
    System.out.println("登录成功");
})

btn.addActionListener(e ->{
    System.out.println("登录成功");
})

btn.addActionListener((e) ->{
    System.out.println("登录成功");
})

btn.addActionListener(e ->{
    System.out.println("登录成功");
})

btn.addActionListener(e-> System.out.println("登录成功"))
```

方法引用

- 静态方法的引用

使用场景：某个**Lambda**表达式里只是调用一个静态方法，并且**"->"**前后参数的形式一致，可以使用静态方法引用

格式：`类名::静态方法`

```
// Student类里面的静态方法
public static int compare(studnets o1,students o2){
    // 方法体
    return o1.getAge() - o2.getAge();
}

Arrays.sort(students,(o1,o2) -> o1.getAge() - o2.getAge());

Arrays.sort(students,(o1,o2) -> Student.compare(o1,o2));
//相当于
Arrays.sort(students,Student::compare)
```

- 实例方法的引用

使用场景：某个**Lambda**表达式里只是通过对象名调用一个实例方法，并且**"->"**前后参数的形式一致，可以使用实例方法引用

格式：`对象名::静态方法`

```
// Student类里面的实例方法
public int compareheight(studnets o1,students o2){
    // 方法体
    return o1.getHeight() - o2.getHeight();
}

Student student = new Student();

Arrays.sort(students,(o1,o2) -> o1.getHeight() - o2.getHeight());
```

```
Arrays.sort(students,(o1,o2) -> student.compareheight(o1,o2));
//相当于
Arrays.sort(students,student::compareheight)
```

- 特定类的方法引用（遇到了再完善）
- 构造器引用

使用场景：某个**Lambda**表达式里只是在创建对象，并且**"->"**前后参数的情况一致，可以使用构造器引用

格式：`类名::new`

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
- 作用：提供了在编译阶段约束所能操作的数据类型
- 不支持基本数据类型，只支持引用数据类型
    - 不支持**int**，**double**等，支持**Integr**，**Double**等
    - 包装类能够让基本数据类型和字符串类型互转

    -
      ```
      // 包装类定义
      // 从-126到127所指向的地址是一样的
      Integer i1 = Integer.valueOf(100);
      Integer i2 = Integer.valueOf(12);

      // 自动装箱成包装类型
      Integer i1 = 100;

      // 自动拆箱成基本数据类型
      int i = i1;

      // 包装类的新功能
      // 1、基本数据类型转换为字符串
      String str1 = Integer.toString(20);

      // 2、把字符串转为基本数据类型
      String str = "23";
      int i1 = Integer.parseInt(str);
      ```

泛型类

```
// 自定义泛型类
public class myArraylist<E>{
    private Arraylist list = new Arraylist();

    public boolean(E e){
        list.add(e);
        return true;
```

```
    }

    public void remove(E e){
        return list.remove(e);
    }
}
```

泛型接口

- 可以使接口里的方法和多种参数类型适配，可以通配一切类型

```
// 自定义泛型接口
public interface Data <T>{
    void add(T t);
    void delete(T t);
}
```

泛型方法、通配符、上下界

- 泛型方法

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

// 这种不是泛型方法
public E get(int index){    // 这个E是泛型类提供的
    return (E)arr[index];
}

// 这种是泛型方法
public static <T> T test(T t){

}
```

- 通配符
    - 即"?"，可以在使用泛型的时候代表一切类型
- 上下限
    - 泛型上限：`?extends car`：?能接受的必须是car或者其子类
    - 泛型下限：`?super car`：?能接受的必须是car或者其父类

```
// 例子
public static void go(ArrayList <?> cars){
    }
```

Collection(单列集合)

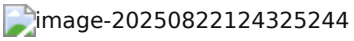

image-20250822124325244

Collection集合特点

- List系列：添加的元素有序、可重复、有索引
- Set系列：添加的元素无序、不重复、无索引

常见功能

方法名	说明
`public boolean add(E e)`	把给定的对象添加到当前集合中
`public void clear()`	清空集合中所有的元素
`public boolean remove(E e)`	把给定的对象在当前集合中删除
`public boolean contains(Object obj)`	判断当前集合中是否包含给定的对象
`public boolean isEmpty()`	判断当前集合是否为空
`public int size()`	返回集合中元素的个数
`public Object[] toArray()`	把集合中的元素，存储到数组中

三种遍历方式

- 迭代器遍历，代表是**Iterator**

```java
Collection<String> names = new Arraylist<>();
names.add("1");
names.add("2");
names.add("3");
names.add("4");
names.add("5");

// 得到这个集合的迭代器对象
Iterator<String> it = names.iterator();
// 取数据，取完数据进行移动
it.next();
// 删除数据
it.remove();

// 用循环遍历
while(it.hasNext()){
    String ele = it.next();
    System.out.println(ele);
}
```

- 增强**for**循环

```
/*
for(元素的数据类型 变量名:数组或者集合){

}
*/

// 例子
for(String s : c){
    System.out.println(s);
}
```

- **Lambda**表达式
    - 用**foreach**遍历

```
Collection<String> names = new Arraylist<>();
names.add("1");
names.add("2");
names.add("3");
names.add("4");
names.add("5");

names.foreach(new Consumer<String>(){
    public void accept(String s){
        System.out.println(s);
    }
});



// 等价于
names.foreach((String s)->{
    System.out.println(s);
});

// 简化
names.foreach(s->System.out.println(s));
```

List

独有方法

方法名	说明
`void add(int index,E element)`	在此集合的指定位置中插入指定元素
`E remove(int index)`	删除指定索引的元素，返回被删除的元素
`E set(int index,E element)`	修改指定索引的元素，返回被修改的元素
`E get(int index)`	返回指定索引的元素

经典代码（用多态）：`List <String> names = new ArrayList()`

ArrayList

- 底层用数组实现
- 初始化的时候数组长度为**0**，当插入第**1**个元素才进行扩容，扩容到10
- 正常扩容的时候是原来长度的**1.5**倍

LinkedList

- 基于链表存储数据，并且是双链表

首尾操作特有方法

方法名	说明
`public void addFirst(E e)`	在该列表开头插入指定元素
`public void addLast(E e)`	将指定元素追加到此列表末尾
`public E getFirst()`	返回此列表的第一个元素
`public E getLast()`	返回此列表的最后一个元素
`public E removeFirst()`	从此列表中删除并返回第一个元素
`public E removeaLast()`	从此列表中删除并返回最后一个元素

Set

无序、不重复、无索引

大多使用**Collection**的方法

HashSet

- 无序、不重复、无索引
- 基于哈希表存储的（数组加链表加红黑树）
    - 使用默认长度为**16**的数组，默认加载因子为**0.75**，一旦存的元素超过16*0.75=12，对哈希表进行扩容，扩容2倍
    - 使用元素的哈希值对数组长度做运算（取余）算出应该要存入的位置
    - 判断位置是否为null，不为null，直接存入，为null的话，用一个链表维护相同位置的不同元素
    - 当链表长度超过8，并且数组长度大于等于64时，链表转为红黑树
- 哈希值（int类型）：Java中的所有对象，都可以调用 `Object` 类的 `hashCode()` 方法返回该对象自己的哈希值

LinkedHashSet

- 有序、不重复、无索引
- 基于哈希表存储的（数组加链表加红黑树），但它的每个元素都额外的多了一个双链表的机制记录它前后元素的位置

TreeSet

- 可排序、不重复、无索引

    - 对于自定义类型的对象，TreeSet默认无法直接排序

        1. 对象实现一个 `Comparable` 接口，重写 `compareTo` 方法，制定比较规则（this是左边表示比较者，o是右边表示被比较者）

        ```
        public int CompareTo(Teacher o){
            if(this.getAge() > o.getAge()) return 1;
        ```

```
    if(this.getAge() < o.getAge()) return -1;
    return 0;
}
```

    - 如果左边大于右边 返回正整数
    - 如果左边小于右边 返回负整数
    - 如果左边等于右边 返回0

2. TreeSet集合自带 `Comparator` 对象，指定比较规则。就是new□个 `Comparator` 比较器，然后重写 `compare` 方法（匿名内部类）

```
Set<Teacher> teachers = new Treeset<>(new Comparator<Teacher>(){
    @Override
    public int compare(Teacher o1,Teacher o2){
        return o1.getAge() - o2.getAge();// 升序
    }

})

Set<Teacher> teachers = new Treeset<>((o1,o2)->{
   return o1.getAge() - o2.getAge();
});
```

- 基于红黑树实现的排序

Map(双列集合)

常用方法

- 键的获取方式的类型是 `Set`
- 值的获取方式的类型是 `Collection`

方法名称	说明
public V put(K key, V value)	添加元素
public int size()	获取集合的大小
public void clear()	清空集合
public boolean isEmpty()	判断集合是否为空，为空返回 true，反之返回 false
public V get(Object key)	根据键获取对应值
public V remove(Object key)	根据键删除整个元素
public boolean containsKey(Object key)	判断是否包含某个键
public boolean containsValue(Object value)	判断是否包含某个值
public Set<K> keySet()	获取全部键的集合
public Collection<V> values()	获取 Map 集合的全部值

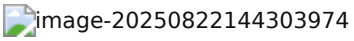

image-20250822144303974

存储的是键值对

- 键不能重复，值可以重复

HashMap（用的最多）

- 无序、不重复、无索引
- **new**一个**HashSet**本质上**new**的是**HashMap**！！！

LinkedHashMap

- 有序、不重复、无索引
- 原理和**LinkedHashSet**一样

TreeMap

- 可排列、不重复、无索引
- 只能对键排序，原理和**TreeSet**相同

```
// 用键，也就是老师对象来进行排序
Map<Teacher,String> map = new TreeMap<>((o1,o2)->{
    return o2.getAge() - o1.getAge();
});

// 遇到浮点类型，要使用Double.compare方法
Map<Teacher,String> map = new TreeMap<>((o1,o2)->{
    return Double.compare(o2.getSalary()-o1.getSalary);
});
```

遍历方式

- 键找值：先获取Map集合的全部键，再通过遍历键来找值

```
Map<String,Integer> map = new Hashmap<>();
// 1. 获取所有键
Set<Integer> keyset = map.keySet();
// 2. 遍历得到值
for(String key: keyset){
    Integer value = map.get(key);
    System.out.println(value);
}
```

- 键值对：把键值对看成一个整体进行遍历
    - 使用 `entrySet()` 获取map里面所有键值对的集合并打包成一个set
    - `Map.Entry<K,V>` 是一个类型，封装了两个方法，`getKey()` 获取键，`getValue()` 获取值

```
Map<String,Integer> map = new Hashmap<>();
Set<Map.Entry<K,V>> sets = map.entrySet();

for(Map.Entry<K,V> set: sets){
```

```
    String key = set.getKey();
    Integer value = set.getValue();
    System.out.println(key + ":" + value);
}
```

- **Lambda**方式: JDK8之后的新方法
  - 使用 `map.forEach()` 方法，然后里面要传的是匿名内部类
    - 类名为 `BiConsumer` ，泛型类型为map存的类型
    - 重写的方法为 `accept`
  - 可以根据函数式编程简化

```
map.forEach(new BiConsumer<String, Integer>(){
    @Override
    public void accept(String key,Integer value){
        System.out.println(key + ":" + value);
    }
});

// 简化
map.forEach((k,v) -> {
    System.out.println(k + ":" + v);
});
```

Stream

可以看作是一条生产流水线

- 原始数据支持数组和集合
- 支持链式编程， `.filter()` 为过滤，里面要实现一个匿名内部类
- 单列集合可以直接调用 `.stream()` 获得stream流
- 双列集合可以获得他的键流或者值流，要想获得键值对流要先对map用 `entrySet()` 打包成set
  - `Map.Entry<K,V>` ：代表map里面的一个键值对，用set包裹起来就指的是这个集合没有重复的键值对

```
// 获取键流
Set<K> keySet = map.keySet(); // 获取map里所有的键
Stream<K> keyStream = keySet.stream(); // 获取键流

// 获取值流
Collection<V> values = map.values(); // 获取map所有的值
Stream<V> valueStream = values.stream();

// 获取键值对流
Set<Map.Entry<K, V>> entrySet = map.entrySet(); // 把map包裹起来，然后放到set里面
Stream<Map.Entry<K, V>> entryStream = entrySet.stream();
```

- 数组要用 `Arrays.stream(数组名)` 或者 `stream.of(数组名)` 获得stream流
  - `Stream的泛型` 不支持基本数据类型

```
Integer []ages = {1,3,46,23,56,18};

Stream<Integer> stream = Arrays.stream(ages);

Stream<Integer> stream = Stream.of(ages);
```

常用的中间方法

支持链式编程

方法名	作用
`Stream <T> filter(匿名内部类)`	用于对流中的数据进行过滤
`Stream <T> sorted()`	对元素进行升序排序
`Stream <T> sorted(匿名内部类)`	按照指定规则排序
`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()`	去除流中重复元素
`Stream <T> map(映射方法)`	对元素进行加工，返回对应的新流

```
// scores是分数，用了map方法之后变成字符串:加10分后s+10
scores.stream().map(s -> "加10分后" + (s + 10));
```

终结方法

常用终结方法

流只能收集一次！！！！

方法名	名称
`void forEach(Consumer action)`	对此流运算后的数据进行遍历
`long count()`	统计此流运算过后的元素个数
`Optional<I> max(匿名内部类)`	获取此流运算后的最大值元素
`Optional<I> min(匿名内部类)`	获取此流运算后的最小值元素
`R collect(Collector collector)`	把流处理后的结果收集到□个指定的集合中
`Object[] toArray()`	把流处理后的结果收集到□个数组中
`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(匿名内部类)`	把元素收集到map集合中

`.reduce(`匿名内部类（可以简化为方法引用））	用于将流中的元素组合成一个单一的结果

- 通过max或者min终结后的值会放在Optional容器里，null也可以放
- 想获取对象要用 `Optional` 的 `.get()` 方法
- 匿名内部类要指定比较规则

```
// teachers是列表
// 要注意指定比较规则
Optional<Teacher> max = teachers.stream().max((t1,t2)-
>Double.compare(t1.getSalary(),t2.getSalary()));
// 获取老师对象
Teacher maxteacher = max.get();

// s是流
// 收集到list
List<User> list = s.collect(Collector.toList());
// 收集到set
Set<User> set = s.collect(Collector.toSet());
// 收集到数组
Object[] array = s.toArray();
// 收集到map
Map<User> map = s.collect(to.Map(t->t.getName(),t->getSalary()));
```

File

代表文件或者文件夹

```
// 获取文件对象
File f1 = new File(文件路径);

// 获取字符个数
f1.length();

// 获取文件名
f1.getName();

// 判断是不是文件
f1.isFile();

// 判断是不是文件夹
f1.isDirectory();

// 创建文件
f1.createNewFile();

// 创建文件夹(只能创建一级文件夹)
f1.mkdir();

// 创建多级文件夹
f1.mkdirs();
```

```
// 删除文件或者文件夹(只能删空的文件夹)
f1.delete();

// 获取某个目录下的所有一级文件名称,返回字符串数组
f1.list();

// 拿一级文件对象，返回一个File数组
f1.listFiles();

// 获取绝对路径
f1.getAbsoluteFile();
```

字符集

常见字符集

- 标准ASCII字符集（首位统一为0）：用1个字节（8位bit）存储
- GBK（汉字内码扩展规范）：兼容ASCII字符集，一个汉字编码为2个字节
  - 汉字的第一个字节的第一位必须是1，为了防止和ASCII字符混淆
- Unicode字符集（统一码）：可以容纳世界上所有文字、符号的字符集

当今最主流的字符集

- **UTF-8**（全世界语言的统一编码方案，兼容全球所有字符）：可变长编码，分成四个长度区：1个字节，2个字节，3个字节，4个字节
  - 做了前缀码，用来区分1、2、3、4字节区

多线程

实现方式

继承**Thread**类

- 重写 `run` 方法，在 `run` 方法编写线程的任务代码
- 调用 `start` 方法启动线程

```java
public class Demo1 extends Thread{
    @Override
    public void run(){
        System.out.println("线程执行");
    }
}

public class Test{
    public static void main(String[] args){
        Thread t1 = new Demo1();
        // 启动线程
        t1.start();
    }
}
```

实现**Runnable**接口

- 实现 `run` 方法
- 把线程任务对象 `r` 用 `Thread` 包装
- 调用 `start` 方法启动线程

```java
// 正常写法
public class Demo2 implements Runnable{
    @Override
    public void run(){
        System.out.println("线程执行");
    }
}

public class Test{
    public static void main(String[] args){
        // 多态创建
        Runnable r = new Demo2();
        // 包装
        Thread t2 = new Thread(r);
        // 启动线程
        t2.start();
    }
}


// 匿名内部类写法
public class Test{
    public static void main(String[] args){
        // 匿名内部类创建
        Runnable r = ()->{
            System.out.println("线程执行");
        };
        // 包装
        Thread t2 = new Thread(r);
        // 启动线程
        t2.start();
    }
}
```

实现Callable接口

- 可以返回线程执行完毕后的结果
- 实现 `Callable` 接口，重写 `call` 方法，封装要做的事和要返回的数据
  - 泛型类型是 `call` 方法的返回值类型
- 把 `Callable` 类型的对象封装成 `FutureTask` （线程任务对象）
  - 泛型类型是 `call` 方法的返回值类型
  - `FutureTask` 实现了 **`Runnable`** 接口
- 把线程任务对象交给 `Thread` 对象
- 调用 `Thread` 对象的 `start` 启动线程
- 线程执行完毕后，通过 `FutureTask` 对象的 `get` 方法去获取线程任务执行的结果

```
public class Demo3 implements Callable<Integer>{
    @Override
    public Integer call(){
        int a = 0;
        for(int i = 1;i <= 10;i++){
            a += i;
        }
        return a;
    }
}

public class Test{
    public static void main(String[] args){
        // 多态创建
        Callable<Integer> c = new Demo3();
        // 包装
        FutureTask<Integer> f = new FutureTask<>(c);
        Thread t2 = new Thread(f);
        // 启动线程
        t2.start();
        // 获取结果
        System.out.println(f.get());
    }
}
```

常用方法

- `public String getName()`：获取线程名
- `public void setName(String name)`：为线程创建名字
- `public static Thread currentThread()`：获取当前执行的线程
- `public final void join()`：让调用这个方法的线程先执行完

线程同步

是解决线程安全问题的解决方案

核心思想：让多个线程先后依次访问共享资源

同步代码块

作用：把访问共享资源的核心代码上锁

```
synchronized(同步锁){
    访问共享资源核心代码
}
```

- 对于当前同时执行的线程来说，同步锁必须是唯一对象
- 可以使用共享资源作为锁对象
    - 对于实例方法用 `this'` 作为锁对象
    - 对于静态方法用字节码 `类名.class` 作为锁对象

同步方法

作用：把访问共享资源的核心方法上锁

- 实例方法默认用 `this` 作为锁对象
- 静态方法默认用字节码 `类名.class` 作为锁对象

```
修饰符 synchronized 返回值类型 方法名称(形参列表){
        操作共享资源代码
}
```

Lock锁

作用：可以创建出具体的锁对象进行加锁和解锁

实现方式：使用 `Lock` （接口）的实现类 `ReentrantLock` 创建锁对象

- `void lock()`：上锁
- `void unlock()`：解锁

```
public final Lock l1 = new ReentrantLock();
```

线程池

提供了代表线程池的接口：`ExecutorService`

创建线程池对象：

1. 使用 `ExecutorService` 的实现类 `ThreadPoolExecutor` 创建□个线程池对象
2. 使用 `Executors` （线程池的工具类）调用方法（静态方法）返回不同特点的线程池对象

`ThreadPoolExecutor` 类提供的构造器

- 任务队列的任务类型是 `Runnable`

```
public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long
keepAliveTime,TimeUnit unit,BlockingQueue <Runnable> workQueue,ThreadFactory
threadFactory,RejectedExecutionHandler handler);

// 创建线程池
ExecutorService pool = new ThreadPoolExecutor(3,5,10,TimeUnit.SECONDS,new
ArrayBlockingQueue<>(3),Executors.defaultThreadFactory(),new
ThreadPoolExecutor.AbortPolicy());
```

- `corePoolSize`：指定线程池的核心线程的数量
- `maximumPoolSize`：指定线程池的最大线程数量
- `keepAliveTime`：指定临时线程的存活时间
- `unit`：指定临时线程存活的时间单位
- `workQueue`：指定线程池的任务队列
- `threadFactory`：指定线程池的线程工厂
- `handler`：指定线程池的任务拒绝策略

`ExecutorService` 的常用方法：

- `void execute(Runnbale command)`：执行 `Runnable` 任务

- `Future<T> submit(Callable<T> task)`：执行`Callable`任务，返回未来任务对象，用于获取线程返回的结果
- `void shutdown()`：等全部任务执行完毕后，再关闭线程池
- `List<Runnable> shutdownNow()`：立即关闭线程池，并且返回**队列中未执行的任务 **

什么是开始创建临时线程？

- 新任务提交时发现核心线程在忙，任务队列也满了，并且还可以创建临时线程

什么时候拒绝新任务？

- 核心线程和临时线程都在忙，并且任务队列也满了

任务拒绝策略

- `AbortPolicy`：丢弃任务并抛出异常，默认做法
- `DiscardPolicy`：丢弃任务但不抛出异常
- `DicardOldestPolicy`：抛弃队列中等待最久的任务，然后把当前任务加入队列
- `CallerRunsPolicy()`：由主线程负责调用人物的`run`方法从而绕过线程池执行

并发和并行

并发：CPU会轮询线程，但切换速度很快，给我们的感觉是在同时执行，这就是并发

并行：同一时刻上同时有多个线程在被CPU调度执行

IO流

按照流的内容，IO流可以分为

- 字节流：适合操作所有类型文件
- 字符流：只适合操作纯文本文件

按照流的方向可以分为

- 输入流
- 输出流

单元测试

针对最小的功能单元方法，编写测试代码进行测试

Junit单元测试框架

- 可以针对某个方法进行测试，也可以一键完成全部方法的自动化测试
- 需要对业务类编写对应的测试类，并为每个业务方法编写对应的测试方法
- 测试方法要加@Test注解

反射

核心：通过堆空间存储的**Class**对象去操纵类，甚至可以访问私有变量

- 加载类，并允许以编程的方式解剖类中的各种成分（成员变量、方法、构造器等）

步骤（类本身也是一个对象，其它同理）

1. 加载类，有三个方法获取类的字节码：**Class**对象
    - `Class c1 = 类名.class`
    - 调用Class提供方法：public static Class forName(String package)：**package**是类的全类名

        - `Class clazz = Class.forName("类的全类名")`
    - `Class c1 = 对象.getClass()`
2. 获取类的构造器：**Constructor**对象
3. 获取类的成员变量：**Field**对象
4. 获取类的成员方法：**Method**对象

```
// 示例
Class c1 = Student.class; //获取类对象
System.out.println(c1.getName()); // 获取类的全类名
System.out.println(c1.getSimpleName()); // 获取类的简名

// 获取构造器信息
Constructor [] cons = c1.getDeclaredConstructors(); //拿到所有的构造器
for(Constructor con: cons){
    System.out.println(con.getName() + "("+con.getParmeterCount()+")"); // 拿到每个构造器并打印参数个数
}

// 获取单个带参构造器
Constructor con2 = c1.getDeclaredConstructor(String.class,int.class); // 获取带参构造器

// 获取成员变量对象
Field field = c1.getDeclaredField("hobby");
System.out.println(field.getName() + "("+field.getType().getName()+")");

// 获取成员方法对象
Method method = c1.getDeclaredMethod("eat"); // 获取成员方法名字为eat的成员方法对象
Method method1= c1.getDeclaredMethod("eat",String.class); // 获取成员方法名字为eat，参数为String的成员方法对象
System.out.println(method.getName() + "("+method.getParmeterCount()+")");
```

重点：用反射拿到构造器

```
Class c1 = Student.class; //获取类对象
Constructor con = c1.getDeclaredConstructor(); // 获取无参构造器

// 暴力反射
con.Accessible(true); // 用这个方法临时绕过访问权限限制
Student c1 = (Student) con.newInstance(); // 使用无参构造器对象创建类的对象，然后强转
System.out.println(C1);
```

对成员变量进行取值和赋值

```
Student c1 = new Student("小明",12); // new对象
Field field = c1.getDeclaredField("hobby"); // 获取成员变量对象
field.setAccessible(true);
field.set(c1,"社交"); // c1是类的对象
```

```
String hobby = (String) field.get(c1); // 获取的是社交 相当于c1.hobby
```

执行成员方法

```
Method method = c1.getDeclaredMethod("eat");
Student c1 = new Student("小明",12); // new对象

method.setAccessible(true);
method.invoke(c1); // 要传参，参数为类的对象 相当于 c1.eat() 有参成员方法则带参数即可
```

反射的作用

- 可以得到□个类的全部成分然后操作
- 可以破坏封装性
- 可以绕过泛型约束（泛型只在编译的时候起作用，反射是拿到编译后的字节码**(class)**文件进行操作，因此可以绕过约束）

```
Arraylist <String> list = new Arraylist<>();
list.add("陈赫");
list.add("李晨");
list.add(88); // 报错

// 拿到成员方法构造器
Class c1 = list.getClass();
Method m1 = c1.getDeclaredMethod("add",Object o);
m1.invoke(list,11); // 不报错
m1.invoke(list,true);// 不报错
```

注解

- `@Override` ，`@Test` 都是特殊标记，即注解，标记做特殊处理
- 注解可以用在类上、构造器上、成员方法上等
- 本质是□个接口

自定义注解

- 特殊属性名：value
    - 使用时如果只有□个value，名称可以不写

```
/*
格式：
public @interface 注解名称{
    public 属性类型 属性名() default 默认值;
}

属性要加括号
*/

public @interface Mybook{
    String value() ;
```

```
}

@Mybook("delete") // 可以省略名称
@Mybook(value = "delete") // 可以不省略
```

元注解

注解注解的注解

- @Target: 声明被修饰的注解只能在哪些位置使用
    - TYPE: 类、接口
    - FIELD: 成员变量
    - METHOD: 成员方法
    - PARAMETER: 方法参数
    - CONSTRUCTOR: 构造器
    - LOCAL_VARIBLE: 局部变量
- @Retention: 声明注解的保留周期
    - **RUNTIME**: 一直保留到运行阶段

```
@Retention(RetentionPolicy.RUNTIME) // 注解注解的注解
@Target({ElementType.METHOD})
public @interface Mybook{ // 注解
    String value() ;
}
```

注解的解析

- 判断类上、成员变量上等是否存在注解，并把里面的内容解析出来
- 拿到谁上面的注解，就要把谁的对象拿到

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,ElementType.TYPE})
public @interface Mybook{ // 注解
    String value() ;
    double height () default 100;
    String []address();
}

@Mybook(value ="刘亦菲",address = {"上海","北京"})
public class Demo{

}

Class c1 = Demo.class; // 获取Demo的对象
if(c1.isAnnotationPresent(Mybook.class)) {  // 判断是否有Mybook这个类的注解
    // 获取注解对象
    Mybook b = (Mybook) c1.getAnotation(Mybook.class);
    // 获取注解信息
    String []address = b.address();
```

```
    double height = b.height();
    String value = b.value();
}
```

动态代理

一种设计模式

- 代理：找□个中介替你实现重复的业务逻辑。假设有□个真实对象，代理对象会包裹这个真实对象，客户端不调用真实对象，而是调用代理对象。在代理对象调用真实对象的方法之前或者之后，加入□些额外逻辑
- 动态：代理类不是预先写好的，而是通过反射动态生成的字节码
- 需要有抽象接口，真实类和代理类（使用**Proxy**创建代理对象），`java.lang.reflect.Proxy` 以及要重写方法的类 `java.lang.reflect.InvocationHandler`
  - 代理类对象是接口的实例！！！也就是说创建代理对象需要用接口声明
  - 真实类需要实现接口
  - 两个重要的包：
    - □个提供创建代理类的方法：`java.lang.reflect.Proxy`
    - □个提供代理类的业务逻辑的重写：`java.lang.reflect.InvocationHandler`

举个例子：

抽象接口

```
public interface Mess{
    public void sing(String name);
    public String dance();
}
```

真实类

```
@AllArgsConstructor
public class Real implements Mess{
    private String name;
    @Override
    public void sing(String name){
        System.out.println(this.name + "唱" + name);
    }
    @Override
    public String dance(){
        System.out.println(this.name + "跳江南Style");
        return "谢谢，谢谢";
    }
}
```

代理类

```
// 导入反射包
import java.lang.reflect.Proxy
// 导入要重写的类
import java.lang.reflect.InvocationHandler
```

```
public class ProxyUtil{
    public static Mess createproxy(Real r)){
        // 参数一：用于指定用哪个类加载器去加载生成的代理类
        // 参数二：用于指定代理需要实现的接口
        // 参数三：指定生成的代理要做什么事情
        Mess m = (Mess)Proxy.newProxyInstance(
            ProxyUtil.class.getClassLoader(),
            r.getClass().getInterfaces(),
            new InvocationHandler(){
                @Override
                public Object invoke(Object proxy , Method method,Object []args)
throws Throwable{

                    // 参数一：proxy表示接受到的代理对象本身
                    // 参数二：method表示正在被代理的方法
                    // 参数三：args表示正在被代理的方法的参数
                    // 方法执行前的逻辑
                    System.out.println("代理开始执行");
                    Object result = method.invoke(r,args);

                    // 方法执行后的逻辑
                    System.out.println("代理结束后开始执行");
                    return result;
                }
            }
        );
            return m;
    }
}
```

测试类

```
public class Test(){
    public static void main(String []args){
        Real r = new Real("钟显博");
        // 创建代理，用接口声明
        Mess proxy = ProxyUtil.createProxy(r);
        // 执行函数
        proxy.sing("江南");
        System.out.println(proxy.dance());
    }
}
```