static 关键字

- **特点：**
  - 使用 `static` 修饰的方法称为 **静态方法**，静态方法属于 **类本身**，而不是对象。
  - 静态方法在内存中只有 **一份数据**，会被所有对象共享。
  - 调用静态方法时 **不需要创建对象**，可以直接通过类名调用：

    ```
    类名.方法名();
    ```

  - 静态方法中 **不能直接访问非静态成员**（实例变量/实例方法），因为非静态成员依赖对象存在。
  - 静态方法、静态变量、类的字节码等存放在 **方法区（Method Area）**，不在堆中。

栈（Stack）与堆（Heap）

1. 堆（Heap）

- 存放所有 `new` 出来的对象和数组。
- 由垃圾回收机制（GC）管理内存回收。
- 容量大，一般不会轻易溢出，但如果创建过多对象或出现内存泄漏，会导致 **堆溢出（OutOfMemoryError）**。

2. 栈（Stack）

- 存放方法运行时的数据，如：方法参数、局部变量、返回值、方法调用信息等。
- 每次调用一个方法，都会在栈中压入一块栈帧；方法执行完，栈帧弹出。
- 栈空间较小，如果递归层数太深或无限递归，会发生 **栈溢出（StackOverflowError）**。

总结记忆：

- `new` 出来的 → **堆**
- 方法执行的数据（参数、局部变量）→ **栈**
- **爆栈比爆堆更常见**（因为递归简单就能爆栈）

---

extends & implements

extends（继承）

- `extends` 用于 **类继承类** 或 **接口继承接口**。
- **类继承类**：单继承，一个子类只能继承一个父类。
- **接口继承接口**：可多继承。

示例：

```
class A {}
class B extends A {}     // 类继承类

interface A {}
interface B extends A {}  // 接口继承接口
```

> 子类继承父类后，可以进行**方法重写（override）**。

---

implements（实现）

- `implements` 用于 **类实现接口。**
- 一个类可以实现多个接口 → **支持多实现。**
- 类必须实现接口中所有抽象方法（除非该类是抽象类）。

示例：

```
interface A {
    void test();
}

class B implements A {
    @Override
    public void test() {}
}
```

> 顺序必须是：**先 extends，再 implements**

示例：

```
class C extends Parent implements A, B {}
```

---

instanceof 关键字

`instanceof` 用于判断一个对象是否为某个类或接口的实例，返回 `true` 或 `false`。

```
对象 instanceof 类型
```

- 判断对象实际类型，避免类型转换异常（ClassCastException）。
- 常用于多态类型判断。

```
class Animal {}
class Dog extends Animal {}

Animal a = new Dog();

System.out.println(a instanceof Dog);    // true
System.out.println(a instanceof Animal); // true
System.out.println(a instanceof Object); // true
```

- `instanceof` 左边必须是对象，右边必须是类型。
- 父类引用指向子类对象时，`instanceof` 判断子类为 true。
- 接口也可以用 `instanceof` 判断：

```
interface A {}
class B implements A {}

A obj = new B();
System.out.println(obj instanceof B); // true
```

抽象类（abstract）

- **没有方法体的方法称为抽象方法**，使用 `abstract` 修饰。
- **包含抽象方法的类一定是抽象类**，但抽象类不一定必须包含抽象方法。
- **抽象类不能创建对象**，只能由其子类继承后创建子类对象。
- 抽象类可以有 **构造方法**，用于子类创建对象时初始化父类成员。
- 子类必须 **重写父类的所有抽象方法**，否则子类也必须是抽象类。
- 抽象类存在的意义：**让子类继承并实现抽象方法，规范子类行为。**

抽象类 vs 接口

对比项	抽象类	接口
构造方法	☑ 有	❌ 没有
成员变量	可有普通成员变量	只能是常量（默认 `public static final`）
继承/实现	单继承（类只能继承一个抽象类）	多实现（类可以实现多个接口）
设计目的	提供对类的部分实现	定义规范，强调行为约束

> 口诀：**有构造、有变量=抽象类；无构造、多实现=接口**

接口（Interface）

- **接口是比抽象类更加彻底的抽象**，体现的是**规范与约束。**
- 接口要求实现类 **必须实现接口中所有抽象方法**，否则实现类必须定义为抽象类。

- 作用：对实现类形成**强制性规范**，强调“能做什么”而不是“怎么做”。

枚举类（enum）

- 使用 `enum` 定义枚举类型，表示固定且有限的常量集合。

示例：

```
enum Weekday {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

- 枚举变量取值必须为枚举中定义的常量，类型安全、可读性高。

String类

string

- 拼接性能问题**：`String s = "xxx"` 反复拼接会创建大量临时对象，耗内存。
  → **用 StringBuilder 或 StringJoiner** 优化。

StringJoiner示例：

```
StringJoiner sj = new StringJoiner(",");
sj.add("A").add("B").add("C");
System.out.println(sj.toString()); // A,B,C
```

- **常用方法**：`str.substring()` 截取 | `str.replace()` 替换
- **equalsIgnoreCase忽略大小写比较字符串是否相等**
- **字符串池（String Pool）**： 字面量方式创建的字符串会存入字符串常量池，减少内存重复占用。
- **== 与 equals 区别**：
  - `==` 比较：基本类型比值；引用类型比**地址**
  - `equals()`：比较字符串**内容**

StringBuilder

- 可变字符串容器。
- 常用方法：
  - `append()` 拼接
  - `length()` 长度
  - `reverse()` 反转
  - `toString()` 转 String
- 拼接和反转字符串常用。

StringJoiner

- 类似 StringBuilder，可变容器（可变容器 + 分隔符 + 可选前后缀）
- 用于高效拼接字符串（可自定义分隔符）。

常用构造方法

```
StringJoiner sj = new StringJoiner(",");          // 指定分隔符
StringJoiner sj2 = new StringJoiner(",", "[", "]"); // 指定分隔符 + 前缀 + 后缀
```

常用方法

方法	作用
`add(String s)`	添加一个字符串
`length()`	返回当前拼接结果的长度
`setEmptyValue(String s)`	当没有元素时，返回自定义空值
`merge(StringJoiner other)`	合并另一个 StringJoiner
`toString()`	转换为最终的字符串

字符串拼接底层原理

- **常量拼接**：`"a" + "b"` → 编译期计算，复用字符串常量池。
- **变量参与拼接**：每次都会生成新字符串，浪费内存。

字符串存储内存

- **直接赋值**：复用字符串常量池
- **new String()**：新开内存空间，不复用

- # * 包装类*

定义

  - 将 **基本数据类型** 转换为 **对象类型。**
  - 主要包装类：

基本类型	包装类
int	Integer
double	Double

基本类型	包装类
char	Character
boolean	Boolean
long	Long
float	Float
byte	Byte
short	Short

常用功能

(1) 基本类型 → 包装类对象

```
Integer i1 = Integer.valueOf(10);      // int 转 Integer
Integer i2 = Integer.valueOf("123");   // String 转 Integer
```

(2) 包装类 → 基本类型

```
int x = i1.intValue();   // Integer → int
double d = Double.valueOf("3.14"); // String → Double
```

(3) 字符串解析为基本类型

```
int n = Integer.parseInt("123");
double d = Double.parseDouble("3.14");
boolean b = Boolean.parseBoolean("true");
```

(4) 进制转换

```
Integer.toBinaryString(10); // "1010"
Integer.toOctalString(10);  // "12"
Integer.toHexString(255);   // "ff"
```

- **valueOf()** → 基本类型 / 字符串 → 包装类对象
- **parseXXX()** → 字符串 → 基本类型
- **toXXXString()** → 数值 → 指定进制字符串

常用 API

Math 类

方法	功能
`abs(a)`	绝对值
`ceil(a)`	大于或等于 a 的最小整数（向上取整）
`floor(a)`	小于或等于 a 的最大整数（向下取整）
`round(a)`	四舍五入，返回最接近的整数
`max(a,b)`	两值中较大值
`min(a,b)`	两值中较小值
`pow(a,b)`	a 的 b 次幂
`random()`	返回 `[0.0,1.0)` 的随机值

System 类

方法	功能
`currentTimeMillis()`	获取当前时间毫秒值（UTC 时区）
`exit(status)`	终止 JVM，0 正常退出，非 0 异常退出
`arraycopy(src, srcPos, dest, destPos, length)`	数组元素复制

Runtime 类

方法	功能
`getRuntime()`	获取当前 JVM 运行环境对象
`exit(status)`	停止 JVM
`availableProcessors()`	CPU 核心数
`maxMemory()`	JVM 可获取最大内存（字节）
`totalMemory()`	JVM 已获取总内存（字节）
`freeMemory()`	JVM 剩余内存（字节）
`exec(command)`	执行系统命令（cmd）

Object 类（重点）

常用方法

方法	功能
`toString()`	返回对象的字符串表示（可视为对象内存地址值）
`equals(Object obj)`	比较两个对象是否相等（默认比较地址值）
`clone()`	对象克隆（默认浅克隆，需要实现 `Cloneable` 接口）

浅克隆示例（默认 Object.clone()）

```
class Person implements Cloneable {
    String name;
    int age;
    int[] scores; // 引用类型

    public Person(String name, int age, int[] scores) {
        this.name = name;
        this.age = age;
        this.scores = scores;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone(); // 默认浅克隆
    }
}

public class TestClone {
    public static void main(String[] args) throws CloneNotSupportedException {
        int[] scores = {90, 80};
        Person p1 = new Person("Alice", 20, scores);
        Person p2 = (Person) p1.clone();

        System.out.println(p1 == p2);         // false，对象不同
        System.out.println(p1.scores == p2.scores); // true，引用类型共用数组

        // 修改 p2 的 scores
        p2.scores[0] = 100;
        System.out.println(p1.scores[0]);     // 100，说明浅克隆共享引用
    }
}
```

深克隆示例

```
class PersonDeep implements Cloneable {
    String name;
    int age;
    int[] scores;

    public PersonDeep(String name, int age, int[] scores) {
        this.name = name;
        this.age = age;
        this.scores = scores;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        PersonDeep cloned = (PersonDeep) super.clone();
        cloned.scores = this.scores.clone(); // 手动克隆引用类型数组
        return cloned;
    }
}

public class TestDeepClone {
    public static void main(String[] args) throws CloneNotSupportedException {
        int[] scores = {90, 80};
        PersonDeep p1 = new PersonDeep("Bob", 25, scores);
        PersonDeep p2 = (PersonDeep) p1.clone();

        System.out.println(p1.scores == p2.scores); // false，引用类型已经独立

        p2.scores[0] = 100;
        System.out.println(p1.scores[0]);         // 90，不受 p2 修改影响
    }
}
```

类型	描述
浅克隆	基本数据类型复制值，引用类型复制地址值（默认 Object.clone()）
深克隆	基本数据类型复制值，引用类型也会创建新对象，完全独立

> **记忆点**：
>
> - 浅克隆 = 拷贝引用
> - 深克隆 = 重新创建对象

- **Math** → 数学运算
- **System** → 系统操作、数组复制、时间
- **Runtime** → JVM 环境、内存、CPU、执行命令

BigInteger（大整数）

- **概述**：支持超出 long 范围的整数运算
- **注意**：字符串中的数字必须符合指定进制

构造 / 静态方法	功能
`BigInteger(int num, Random rnd)`	随机大整数 `[0, 2^num-1]`
`BigInteger(String val)`	指定大整数
`BigInteger(String val, int radix)`	指定进制的大整数
`static valueOf(long val)`	静态方法获取 BigInteger 对象（内部优化）

算术方法	功能
`add(BigInteger val)`	加法
`subtract(BigInteger val)`	减法
`multiply(BigInteger val)`	乘法
`divide(BigInteger val)`	除法
`divideAndRemainder(BigInteger val)`	除法，返回商和余数
`pow(int exponent)`	次幂、次方
`max(BigInteger val)` / `min(BigInteger val)`	返回较大值 / 较小值

其他方法	功能
`equals(Object x)`	比较是否相等
`intValue()`	转为 int（超出范围可能错误）

BigDecimal（高精度小数）

- **用途**：处理浮点数精度问题，常用于金融计算
- **常用方法**：
  - 构造：`BigDecimal(String val)`、`BigDecimal(double val)`
  - 算术：`add()`, `subtract()`, `multiply()`, `divide()`
  - 比较：`compareTo(BigDecimal val)`、`equals()`
  - 精度控制：`setScale(int scale, RoundingMode mode)

1. **Object / 克隆**
    - 浅克隆 = 共享引用，深克隆 = 引用独立
    - `toString()` → 对象表示，`equals()` → 比较对象
2. **BigInteger**
    - 支持大整数运算
    - 构造 / valueOf → 获取对象
    - 算术方法 + pow/max/min
3. **BigDecimal**
    - 精确小数运算，金融计算用
    - `setScale()` 控制小数位和舍入方式

我帮你把你提供的 **Java 时间类**内容整理成 **精简笔记 + 对照表**，方便快速理解和记忆，同时加入重点提示：

时间类

SimpleDateFormat（旧版时间格式化）

- **用途**：在 `Date` 与 `String` 之间转换
- **常用方法**：

```
String format(Date date)  // Date → String
Date parse(String source) // String → Date
```

Calendar（旧版日历类）

- 表示日历，可以进行 **日期运算**
- 示例：

```
Calendar cal = Calendar.getInstance();
cal.add(Calendar.DAY_OF_MONTH, 5); // 当前日期 +5天
Date date = cal.getTime();
```

时间类（jdk8）

类名	作用
ZoneId	时区
Instant	时间戳（秒 + 纳秒，精确到纳秒）
ZonedDateTime	带时区的时间

类名	作用
DateTimeFormatter	时间格式化与解析
LocalDate	年、月、日
LocalTime	时、分、秒
LocalDateTime	年、月、日、时、分、秒
Duration	时间间隔（秒、纳秒）
Period	时间间隔（年、月、日）
ChronoUnit	时间间隔（支持所有单位）

Instant（时间戳）

- 精确到纳秒
- **常用方法**：

```
Instant.now()               // 当前时间
Instant.ofEpochMilli(long ms) // 根据毫秒获取
instant.atZone(ZoneId zone)   // 指定时区
instant.plusXxx()/minusXxx()  // 增加/减少时间
instant.isXXX(Instant other)  // 比较方法
```

ZonedDateTime（带时区的时间）

- **常用方法**：

```
ZonedDateTime.now()         // 当前时间
ZonedDateTime.of(...)       // 指定时间
zonedDateTime.withXxx(...)   // 修改时间
zonedDateTime.plusXxx(...)   // 增加时间
zonedDateTime.minusXxx(...)  // 减少时间
```

DateTimeFormatter（时间格式化）

- **获取格式对象**：

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
```

- **格式化时间**：

```
String str = formatter.format(LocalDateTime.now());
```

LocalDate / LocalTime / LocalDateTime（重点）

- **LocalDate** → 年、月、日
- **LocalTime** → 时、分、秒
- **LocalDateTime** → 年、月、日、时、分、秒
- **常用方法**：
  - `now()` → 获取当前时间
  - `of(...)` → 获取指定时间
  - `plusXxx()` / `minusXxx()` → 增减时间

ChronoUnit（时间间隔）

- **常用方法**：

```
ChronoUnit.DAYS.between(startDate, endDate)  // 两个时间间隔（long）
ChronoUnit.SECONDS.getDuration()             // 获取时间单位 Duration 对象
ChronoUnit.DAYS.toString()                   // 返回单位名称
```

💡 **小结记忆点**：

- **旧版**：`Date` + `Calendar` + `SimpleDateFormat` → 可用，但不推荐
- **JDK8+**：`LocalDate/Time/DateTime` + `ZonedDateTime` + `Instant` + `DateTimeFormatter` → 推荐
- **ChronoUnit / Duration / Period** → 时间间隔计算

多线程

线程创建方式

方式	实现	示例
继承 Thread 类	重写 `run()` 方法	`java class MyThread extends Thread { public void run(){...} }`
实现 Runnable 接口	实现 `run()` 方法	`java class MyRunnable implements Runnable { public void run(){...} }`

方式	实现	示例
实现 Callable 接口 + FutureTask	可以有返回值	`java FutureTask<Integer> task = new FutureTask<>(new MyCallable()); new Thread(task).start();`

线程启动与运行

```
thread.start();  // 启动线程，调用 run() 方法
thread.run();   // 普通方法调用，不启动新线程
```

线程状态

状态	说明
NEW	新建线程，尚未启动
RUNNABLE	可运行状态（CPU 可调度）
BLOCKED	阻塞等待锁
WAITING	等待线程通知，无时间限制
TIMED_WAITING	等待线程通知，有超时
TERMINATED	线程结束

常用线程方法

方法	作用
`sleep(ms)`	当前线程休眠指定毫秒
`join()`	等待线程结束后再继续执行
`yield()`	暂停当前线程，给其他线程机会
`setName()` / `getName()`	设置或获取线程名
`setPriority()` / `getPriority()`	设置或获取线程优先级
`isAlive()`	判断线程是否存活

线程安全关键字与类

关键字/类	功能
`synchronized`	修饰方法或代码块，实现互斥访问
`volatile`	保证变量在多线程间可见，禁止指令重排序
`ReentrantLock`	显示锁，可替代 synchronized
`AtomicXXX`	原子操作类（如 AtomicInteger）
`ThreadLocal`	每个线程独立存储变量

线程通信方法

- `wait()` → 让当前线程进入等待（需持有对象锁）
- `notify()` → 唤醒一个等待线程
- `notifyAll()` → 唤醒所有等待线程

💡 **记忆小技巧**：

1. **创建线程** → Thread / Runnable / Callable
2. **线程方法** → start/run/sleep/join/yield
3. **线程安全** → synchronized / volatile / Lock / Atomic
4. **线程通信** → wait/notify/notifyAll

TCP 与 UDP 多线程处理方法

TCP（面向连接）

- 面向连接，每个客户端需要单独 **Socket**
- 服务器通常 **多线程处理每个客户端

```
// 1. 服务端主线程负责监听端口
ServerSocket server = new ServerSocket(8888);

while(true){
    Socket client = server.accept(); // 阻塞等待客户端连接
    // 2. 为每个客户端创建线程处理
    new Thread(() -> {
        // 处理客户端请求
        InputStream in = client.getInputStream();
        OutputStream out = client.getOutputStream();
        // 读写数据
    }).start();
}
```

方式

- **继承 Thread** 或 **实现 Runnable/Callable** 都可
- 每个客户端对应一个独立线程

UDP（无连接）多线程处理

特点

- 无连接，每个数据包独立
- 通常一个线程 **循环接收数据**
- 可使用线程池或单线程循环处理，也可多线程处理不同数据包

示例思路

```
DatagramSocket socket = new DatagramSocket(8888);

while(true){
    DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);
    socket.receive(packet); // 阻塞等待数据包

    // 使用线程池处理接收到的数据
    new Thread(() -> {
        // 处理 packet 数据
        String msg = new String(packet.getData(), 0, packet.getLength());
        System.out.println("收到消息: " + msg);
    }).start();
}
```

方式

- UDP 数据包处理可以 **线程池 + Runnable**
- 不需要为每个客户端建立独立线程（因为没有连接）

特性	TCP	UDP
连接方式	面向连接	无连接
线程创建	每个客户端独立线程	可单线程循环或线程池处理数据包
数据顺序	有序	无序，每个包独立
可靠性	高（ACK/重传机制）	低，不保证到达
使用方式	Thread / Runnable / Callable	Runnable + 线程池更常用

- TCP = **每个客户端独立线程**
- UDP = **循环接收 + 线程池处理数据包我帮你把** Java 集合类**整理成一个** 清晰对照表**，方便快速记忆顺序、特点和常用类。

集合类

Collection 接口（顶层接口）

- **特点**：操作一组对象（元素）
- **主要子接口**：`List`、`Set`、`Queue`

List（有序可重复）

类	特点	说明
`ArrayList`	底层数组	查询快，增删慢
`LinkedList`	双向链表	增删快，查询慢
`Vector`	线程安全数组	类似 ArrayList，但同步开销大

特性：

- **有序**（插入顺序）
- **可重复**（允许重复元素）
- **索引访问**（通过 `get(index)`）

Set（无序不可重复）

类	特点	说明
`HashSet`	无序	基于哈希表，元素唯一

类	特点	说明
`TreeSet`	有序	基于红黑树，自动排序
`LinkedHashSet`	有序	保留插入顺序

特性：

- **无重复元素**
- **HashSet** → 无序
- **TreeSet** → 按自然顺序或自定义 Comparator 排序
- **LinkedHashSet** → 插入顺序

Queue（队列）

- 用于 **FIFO（先进先出）**
- 常用类：
  - `LinkedList`（双向链表实现 Queue）
  - `PriorityQueue`（优先级队列）

总结对比

接口	有序	可重复	常用实现类
List	✔	✔	ArrayList, LinkedList, Vector
Set	✘ / ✔	✘	HashSet, TreeSet, LinkedHashSet
Queue	✔（逻辑顺序）	✔ / ✘	LinkedList, PriorityQueue

💡 **记忆技巧**：

1. **List = 有序 + 可重复** → 索引访问
2. **Set = 无重复** → HashSet（无序）、TreeSet（排序）、LinkedHashSet（插入顺序）
3. **Queue = 队列逻辑** → FIFO 或 优先级
