Java 基础 第 2 阶段：面向对象编程——尚硅谷学习笔记（含面试题） 2023 年

第 6 章 面向对象——基础

6.1 面向过程 vs 面向对象

- 不管是面向过程、面向对象，都是程序设计的思路。
- 面向过程（Process Oriented Programming，简称 POP）：以函数为基本单位，适合解决简单问题。

- 面向对象（ Object Oriented Programming），简称 OOP）：以类为基本单位，适合解决复杂问题。

6.2 类、对象

6.2.1 类

类：具有相同特征的事物的抽象描述，是抽象的、概念上的定义。

6.2.2 对象

对象：实际存在的该类事物的每个个体，是具体的，因而也称为实例。

面向对象完成具体功能的操作的三步流程：

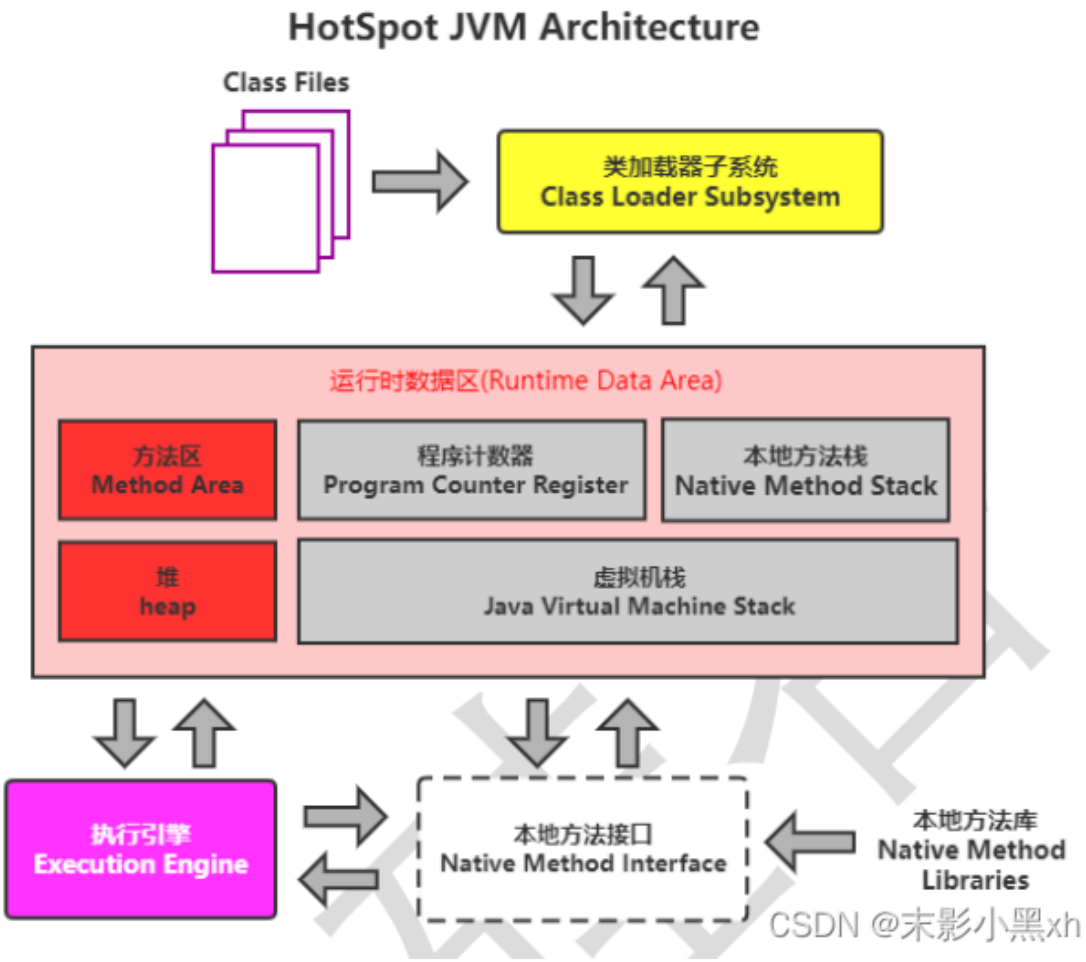
- 步骤 1：创建类，并设计类的内部成员（属性、方法）。
- 步骤 2：创建类的对象。如：Phone p1 = new Phone()。
- 步骤 3：通过对象，调用其内部声明的属性或方法，完成相关的功能。

匿名对象 （anonymous object）：

- 我们也可以不定义对象的句柄，而直接调用这个对象的方法。这样的对象叫做匿名对象。
- 如：new Person().shout();
- 使用情况：
    - 如果一个对象只需要进行一次方法调用，那么就可以使用匿名对象。
    - 我们经常将匿名对象作为实参传递给一个方法调用。

6.2.3 对象的内存解析

- Java 中内存结构划分为：虚拟机栈、堆、方法区；程序计数器、本地方法栈。
- 虚拟机栈：以栈帧为基本单位，有入栈和出栈操作；每个栈帧入栈操作对应一个方法的执行；方法内的局部变量会存储在栈帧中。
- 堆空间：new 出来的结构（数组、对象）：
  ① 数组，数组的元素在堆中
  ② 对象的成员变量在堆中。
- 方法区：加载的类的模板结构。



6.3 类的成员之一：属性（成员变量 field）

6.3.1 声明成员变量

语法格式：

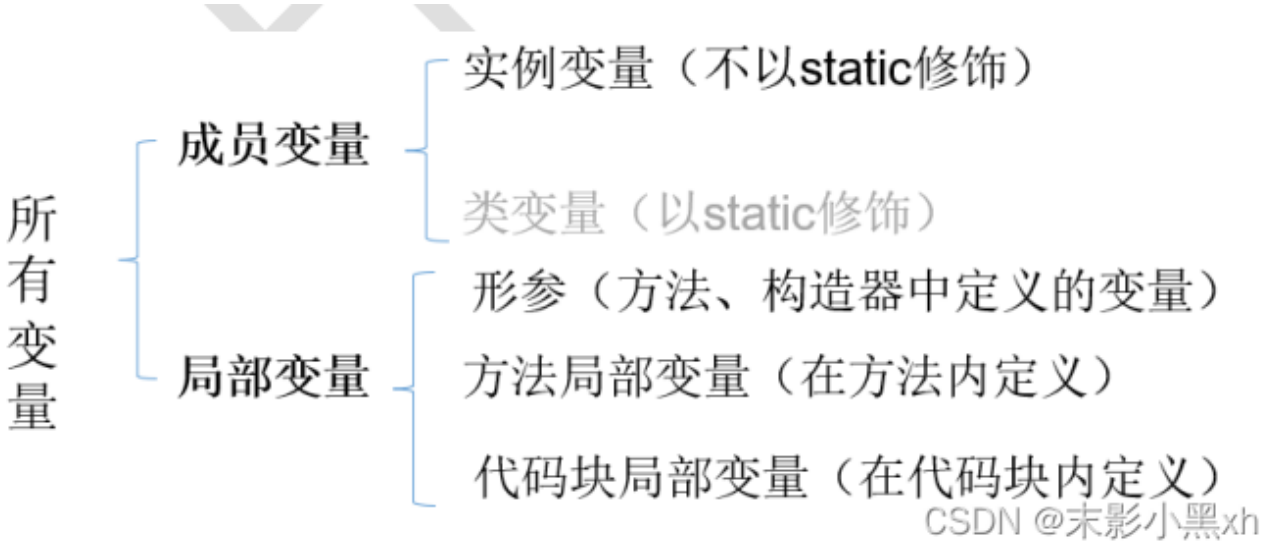
```
修饰符 class 类名
{
    修饰符 数据类型 成员变量名 = 初始化值;
}
```

代码示例：

```
public class Person
{
    private int age;     // 声明 private 变量 age
    public String name = “Lila”;     // 声明 public 变量 name
}
```

6.3.2 成员变量 vs 局部变量

- 在方法体外，类体内声明的变量称为成员变量。
- 在方法体内部等位置声明的变量称为局部变量。



- 不同点：
    - 内存中存储的位置
      ① 成员变量：堆 。
      ② 局部变量：栈。
    - 生命周期
      ① 成员变量：和对象的生命周期一样，随着对象的创建而存在，随着对象被 GC 回收而消亡， 而且每一个对象的实例变量是独立的。
      ② 局部变量：和方法调用的生命周期一样，每一次方法被调用而在存在，随着方法执行的结束而消亡， 而且每一次方法调用都是独立。
    - 作用域
      ① 实例变量：通过对象就可以使用，本类中直接调用，其他类中“对象.实例变量” 。
      ② 局部变量：出了作用域就不能使用。

6.4 类的成员之二：方法（method）

方法(method、函数)的理解：

- 方法是类或对象行为特征的抽象，用来完成某个功能操作。在某些语言中也称为函数或过
  程。
- 将功能封装为方法的目的是，可以实现代码重用，减少冗余，简化代码。
- Java 里的方法不能独立存在，所有的方法必须定义在类里。

6.4.1 方法的声明

语法格式：

```
修饰符 返回值类型 方法名(形参列表) throws 异常列表
{
    方法体的功能代码
}
```

代码示例：

```java
public class Person
{
    private String name;
    private int age;

    // get 方法用于获取 name 属性值
    public String getName()
    {
        return name;
    }

    // set 方法用于设置 name 属性值
    public void setName(String name)
    {
        this.name = name;
    }

    // get 方法用于获取 age 属性值
    public int getAge()
    {
        return age;
    }

    // set 方法用于设置 age 属性值
    public void setAge(int age)
    {
        this.age = age;
    }
}
```

6.4.2 方法的重载(overload)

- 方法的重载的要求：“两同一不同”，方法名、返回值类型相同，形参列表不同。
- 调用方法时，如何确定调用的是某个指定的方法呢?
  ① 方法名
  ② 形参列表

6.4.3 可变个数形参的方法

语法格式:

```
void method (int ... args)
{

}
```

int ... arg 相当于 int[] arg

6.4.4 方法的参数传递机制：值传递

如果形参是基本数据类型的变量，则将实参保存的数据值赋给形参。

如果形参是引用数据类型的变量，则将实参保存的地址值赋给形参。

6.4.5 递归方法

递归方法构成了隐式的循环。

对比：相较于循环结构，递归方法效率稍低，内存占用偏高。

6.5.5 方法调用内存分析

方法没有被调用的时候，都在方法区中的字节码文件(.class)中存储。

方法被调用的时候，需要进入到栈内存中运行。方法每调用一次就会在栈中有一个入栈动作，即给当前方法开辟一块独立的内存区域，用于存储当前方法的局部变量的值。

当方法执行结束后，会释放该内存，称为出栈，如果方法有返回值，就会把结果返回调用处，如果没有返回值，就直接结束，回到调用处继续执行下一条指令。

栈结构：先进后出，后进先出。

6.5 对象数组

- 数组的元素可以是基本数据类型，也可以是引用数据类型。当元素是引用类型中的类时，我们称为对象数组。
- String[ ];
- Person[ ];
- Customer[ ];

6.6 package、import 关键字

- package：指明声明的类所属的包。
  语法格式：

```
package 顶层包名.子包名 ;
```

- import：当前类中，如果使用其它包下的类（除 java.lang 包），原则上就需要导入。
  语法格式：

```
import 包名.类名;
```

6.7 面向对象的特征之一：封装性

Java 规定了 4 种权限修饰，分别是：private、缺省、protected、public。

我们可以使用 4 种权限修饰来修饰类及类的内部成员。当这些成员被调用时，体现可见性的大小。

修饰符	本类内部	本包内	其他包的子类	其他包非子类
private	√	×	×	×
缺省	√	√	×	×
protected	√	√	√	×
public	√	√	√	√

6.7.1 封装性的体现

- 场景 1：私有化(private)类的属性，提供公共(public)的 get 和 set 方法，对此属性进行获取或修改。
- 场景 2：将类中不需要对外暴露的方法，设置为 private。

- 场景 3：单例模式中构造器 private 的了，避免在类的外部创建实例。

6.7.2 封装性的作用

- 高内聚：类的内部数据操作细节自己完成，不允许外部干涉；
  （Java 程序通常以类的形态呈现，相关的功能封装到方法中。）
- 低耦合：仅暴露少量的方法给外部使用，尽量方便外部调用。
  （给相关的类、方法设置权限，把该隐藏的隐藏起来，该暴露的暴露出去。）

6.8 类的成员之三：构造器（Constructor）

语法格式：

```
修饰符 class 类名
{
    修饰符 构造器名()
    {
        // 实例初始化代码
    }
    修饰符 构造器名(参数列表)
    {
        // 实例初始化代码
    }
}
```

代码示例：

```
public class Person
{
    private String name;
    private int age;

    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

构造器的作用：

① 搭配上 new，用来创建对象 。
② 初始化对象的成员变量。

6.9 类的实例变量的赋值过程

- 在类的属性中，可以有哪些位置给属性赋值?
  ① 默认初始化;
  ② 显式初始化;
  ③ 构造器中初始化;
  ④ 通过"对象.方法"的方式赋值;
  ⑤ 通过"对象.属性"的方式赋值;
- 这些位置执行的先后顺序是怎样?
  ① - ② - ③ - ④/⑤

6.10 JavaBean

JavaBean 是一种 Java 语言写成的可重用组件。

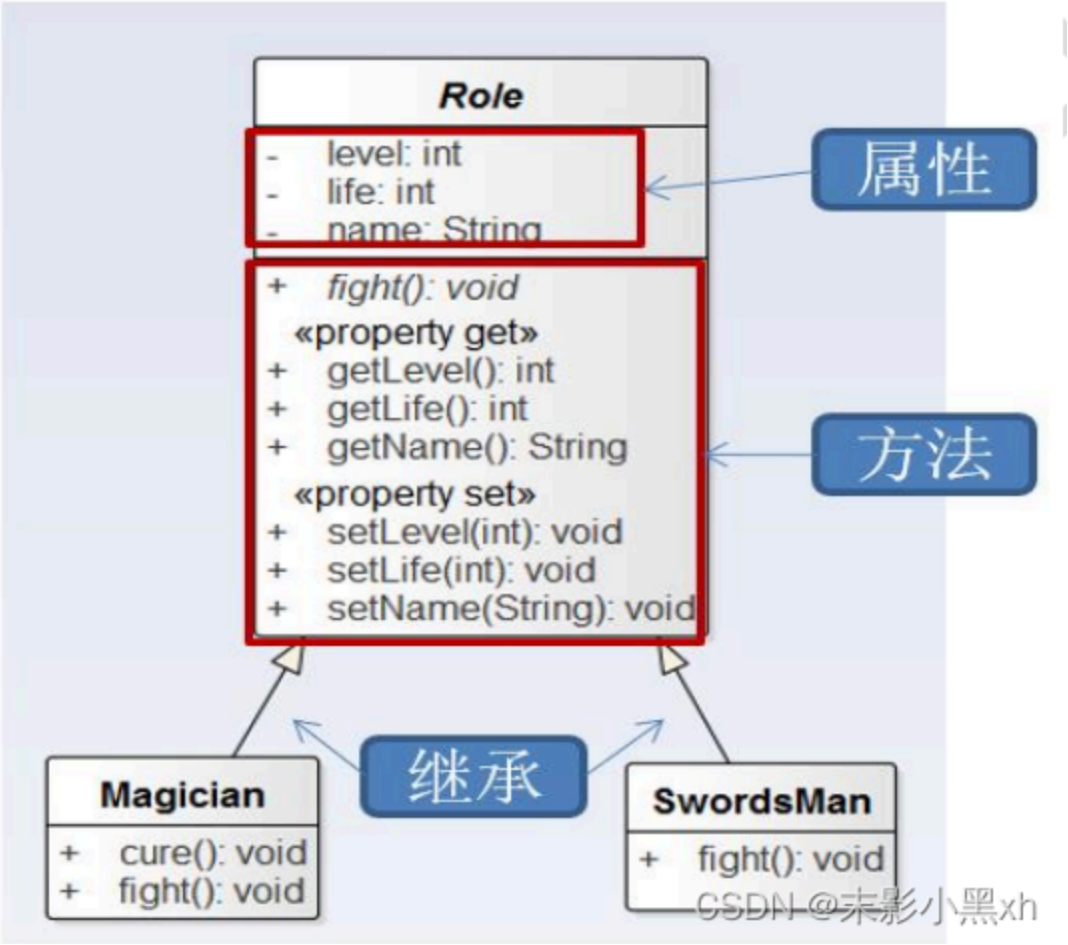
所谓 JavaBean，是指符合如下标准的 Java 类:

- 类是公共的。
- 有一个无参的公共的构造器。
- 有属性，且有对应的 get、set 方法。

6.11 UML 类图

UML（Unified Modeling Language，统一建模语言），用来描述软件模型和架构的图形化语言。

在软件开发中，使用 UML 类图可以更加直观地描述类内部结构（类的属性和操作）以及类之间的关系（如关联、依赖、聚合等）。



6.12 企业真题

1. 面向对象，面向过程的理解?
   答:
   面向对象和面向过程是两种不同的编程思想。
   面向过程编程是一种按照一定的流程和步骤来完成任务的编程方式。它将程序看作是一系列的步骤或者函数，每个函数都有一定的输入和输出。面向过程的编程思想强调的是程序的执行过程，程序的主要任务是按照一定的流程和步骤来完成任务。
   面向对象编程是一种将程序看作一组对象的编程方式，它将程序看作是一组相互协作的对象，每个对象都有一定的属性和方法。面向对象的编程思想强调的是程序的设计和架构，程序的主要任务是设计和实现对象之间的关系和交互。
   总的来说，面向过程的编程思想适合于简单的程序，而面向对象的编程思想适合于复杂的程序。在实际的编程中，可以根据需要选择不同的编程思想来完成任务。
2. Java 的引用类型有哪几种?
   答:

类、数组、接口、枚举、注解、记录。

3. 类和对象的区别?

   答:

   类是一种抽象的概念，描述了一类具有相同属性和行为的对象的集合。对象则是类的实例，具有独立的状态和行为。

   类是一种模板或蓝图，用于创建对象。它定义了对象的属性和方法，但并不实际存在于程序中。对象则是类的具体实现，它们是程序中实际存在的实体。

   类和对象之间的关系是一种“是一种”（is-a）的关系。例如，一个狗类可以派生出吉娃娃犬类，吉娃娃犬就是狗类的一种具体实现。同样，一个具体的吉娃娃犬对象也是狗类的一个实例。

   总之，类是一种抽象的概念，描述了一类对象的共同特征；对象则是类的具体实现，具有独立的状态和行为。

4. 面向对象，你解释一下，项目中哪些地方用到面向对象?

   答:

   “万事万物皆对象”。

   面向对象是一种编程思想，它将程序中的数据和操作封装成对象，并通过对象之间的交互来完成任务。在面向对象的编程中，重点是对象，而不是函数或过程。

   在项目中，面向对象通常用于设计和实现程序的核心功能模块。例如，一个电商网站的订单管理系统可以设计为一个 Order 类，它包含订单信息和操作方法，如创建订单、更新订单状态等。另外，面向对象还可以用于设计和实现用户界面、数据访问、日志记录等模块。

   具体地说，项目中哪些地方用到面向对象取决于具体的业务需求和技术架构，但通常会涉及以下方面:

   ① 类的设计和实现：根据业务需求，设计和实现类，包括类的属性、方法、构造函数、析构函数等。

   ② 对象的创建和使用：在程序运行时，根据需要创建对象，调用对象的方法来完成任务。

   ③ 继承和多态：利用继承和多态的特性，实现代码的复用和灵活性。

   ④ 接口和抽象类：通过接口和抽象类，定义程序的公共接口，提高代码的可维护性和可扩展性。

   总之，面向对象是一种强大的编程思想，可以提高代码的可读性、可维护性和可扩展性，是现代软件开发中不可或缺的一部分。

5. Java 虚拟机中内存划分为哪些区域，详细介绍一下。

   答:

   Java 虚拟机中内存划分为以下几个区域:

① 程序计数器区域（Program Counter Register）：程序计数器是一块较小的内存区域，它可以看作是当前线程所执行的字节码指令的行号指示器。每个线程都有一个独立的程序计数器，用于记录线程执行的位置。当线程执行 Java 方法时，程序计数器记录的是正在执行的虚拟机字节码指令的地址；当线程执行 native 方法时，程序计数器的值为 undefined。
② Java 虚拟机栈区域（Java Virtual Machine Stacks）：每个线程在创建时都会分配一个 Java 虚拟机栈，用于存储方法调用的局部变量、操作数、返回值等信息。Java 虚拟机栈是一种线程私有的内存区域，它的生命周期与线程的生命周期相同。当线程调用一个方法时，Java 虚拟机会为该方法分配一个栈帧，并将该栈帧推入当前线程的 Java 虚拟机栈中。当方法执行结束时，对应的栈帧会被弹出。
③ 本地方法栈区域（Native Method Stacks）：与 Java 虚拟机栈类似，本地方法栈也是一种线程私有的内存区域，用于存储 native 方法的局部变量、操作数、返回值等信息。与 Java 虚拟机栈不同的是，本地方法栈为 native 方法服务。
④ 堆区域（Heap）：堆是 Java 虚拟机中最大的一块内存区域，用于存储对象实例和数组。堆是所有线程共享的内存区域，它的大小可以通过-Xmx 和-Xms 参数进行调整。Java 虚拟机的垃圾回收器会定期对堆中的无用对象进行回收。
⑤ 方法区域（Method Area）：方法区是一种线程共享的内存区域，用于存储类的结构信息、常量池、静态变量、即时编译器编译后的代码等。方法区的大小可以通过-XX:MaxMetaspaceSize 参数进行调整。在 JDK8 之前，方法区被称为持久代（Permanent Generation），但在 JDK8 之后，持久代被移除，方法区被移到了本地内存中，称为元空间（Metaspace）。
⑥ 运行时常量池区域（Runtime Constant Pool）：运行时常量池是方法区的一部分，用于存储编译期间生成的字面量和符号引用。与 Class 文件中的常量池不同，运行时常量池可以动态地添加、删除、修改常量池中的内容。在 JDK7 之前，运行时常量池也属于方法区，但在 JDK7 之后，运行时常量池被移到了堆中。

6. 对象存在 Java 内存的哪块区域里面?
   答:
   堆空间。
7. private 、缺省、protected、public 的作用区域?
   答:
   下表是 private、缺省、protected、public 的作用区域:

修饰符	类内部	同包	继承子类	其他包
private	✓			

修饰符	类内部	同包	继承子类	其他包
缺省	✓	✓		
protected	✓	✓	✓	
public	✓	✓	✓	✓

解释：
类内部：在类的内部可以随意访问该成员。
同包：在同一个包中的其他类可以访问该成员。
继承子类：在继承该类的子类中可以访问该成员。
其他包：在其他包中的类可以访问该成员。

8. main 方法的 public 能不能换成 private？为什么？
   答：
   能。但是更改以后就不能作为程序的入口了，就只是一个普通的方法。
9. 构造方法和普通方法的区别？
   答：
   编写代码的角度：没有共同点。声明格式、作用都不同。
   字节码文件的角度：构造器会以()方法的形态呈现，用以初始化对象。
10. 构造器 Constructor 是否可被 overload？
    答：
    可以。
11. 无参构造器和有参构造器的的作用和应用？
    答：
    无参构造器和有参构造器是 Java 中的两种构造方法。
    无参构造器是指不需要传入参数的构造器，它是默认的构造器，如果一个类没有定义任何构造器，那么编译器会自动为该类生成一个无参构造器。无参构造器的作用主要是用来初始化对象的成员变量，为对象赋初值，也可以在其中进行一些初始化操作。
    有参构造器是指需要传入参数的构造器，它可以根据传入的参数不同来创建不同的对象。有参构造器的作用主要是用来实现对象的初始化，可以在其中设置对象的属性，进行一些初始化操作，以及传递参数给父类的构造器。
    无参构造器和有参构造器的应用场景：
    ① 无参构造器适用于只需要对对象进行简单的初始化的情况。
    ② 有参构造器适用于需要传递参数，进行复杂初始化操作的情况。

③ 如果一个类需要继承父类，那么需要在子类的有参构造器中调用父类的有参构造器，以便完成父类的初始化操作。

④ 在实现某些设计模式时，如工厂模式、建造者模式等，需要使用有参构造器来创建对象。

12. 成员变量与局部变量的区别?

答:

① 定义位置不同：成员变量定义在类中，局部变量定义在方法、代码块或者语句中。

② 生命周期不同：成员变量的生命周期与对象相同，而局部变量在方法结束后就会被销毁。

③ 访问方式不同：成员变量可以被类中的所有方法访问，而局部变量只能在定义它的方法中被访问。

④ 默认值不同：成员变量有默认值，而局部变量没有。成员变量的默认值是基本类型为 0 或者 false，引用类型为 null。

⑤ 内存分配不同：成员变量在对象创建时会被分配内存空间，而局部变量在方法调用时才会被分配内存空间。

⑥ 作用范围不同：成员变量的作用范围是整个类，而局部变量的作用范围只在定义它的方法内部。

13. 变量赋值和构造方法加载的优先级问题?

答:

通过字节码文件，变量显式赋值先于构造器中的赋值。

从字节码文件的角度来看，变量的显式赋值是在类的初始化阶段执行的，而构造方法中的赋值是在对象实例化阶段执行的。

在类的初始化阶段，字节码解释器会按照顺序执行类中所有静态变量的显式赋值语句，将值存储在静态变量表中。然后再执行静态代码块和其他静态方法。这个过程只会执行一次，即在类被加载到内存中时执行。

在对象实例化阶段，字节码解释器会先分配对象所需的内存空间，然后会按照顺序执行实例变量的显式赋值语句，将值存储在实例变量表中。接着执行构造方法中的代码，其中可能包含对实例变量的赋值操作。这个过程对每个对象都会执行一次。

因此，从字节码文件的角度来看，变量的显式赋值优先于构造方法中的赋值，因为它们在不同的阶段执行。

假设有以下 Java 类:

```
public class MyClass
{
    private int x = 1;

    public MyClass(int x)
    {
        this.x = x;
    }
}
```

对应的字节码文件如下（省略了一些细节）：

```
// MyClass.class

// 类的头部
public class MyClass
{
    // 实例变量表
    private int x;  // 默认值为0

    // 构造方法表
    public MyClass(int x)
    {
        // 调用父类构造方法
        super();
        // 对象实例化阶段，先执行实例变量的显式赋值
        this.x = 1;
        // 构造方法中的赋值
        this.x = x;

        // 构造方法结束
        return;
    }

    // 静态变量表
    static int y;  // 默认值为0

    // 静态代码块
    static
    {
        // 类的初始化阶段，先执行静态变量的显式赋值
        y = 2;
        // 再执行静态代码块和其他静态方法
        // ...

        return;
    }
}
```

从字节码文件可以看出，在类的初始化阶段，静态变量的显式赋值先于静态代码块和其他静态方法执行；而在对象实例化阶段，实例变量的显式赋值先于构造方法中的赋值执行。

以下是使用 javap 命令生成的 MyClass 类的字节码汇编代码：

```
public class MyClass
{
  private int x;

  public MyClass(int);
    Code:
        0: aload_0
        1: invokespecial #1                  // Method java/lang/Object."<init>":()V
        4: aload_0
        5: iconst_1
        6: putfield      #2                  // Field x:I
        9: aload_0
        10: iload_1
        11: putfield      #2                 // Field x:I
        14: return

  static {};
    Code:
        0: iconst_2
        1: putstatic     #3                  // Field y:I
        4: return
}
```

其中，构造方法的字节码指令如下：

```
0: aload_0           // 将this引用入栈
1: invokespecial #1  // 调用父类构造方法
4: aload_0           // 将this引用入栈
5: iconst_1          // 将常量1入栈
6: putfield #2       // 将栈顶值（常量1）赋值给实例变量x
9: aload_0           // 将this引用入栈
10: iload_1          // 将参数x入栈
11: putfield #2      // 将栈顶值（参数x）赋值给实例变量x
14: return           // 返回
```

静态代码块的字节码指令如下：

```
0: iconst_2          // 将常量2入栈
1: putstatic #3      // 将栈顶值（常量2）赋值给静态变量y
4: return            // 返回
```

可以看出，字节码汇编代码中也体现了变量的显式赋值先于构造方法中的赋值。

第 7 章：面向对象——进阶

7.1 this 关键字

- this 调用的结构：属性、方法、构造器。
- this 调用属性或方法时，理解为当前对象或当前正在创建的对象。
  代码示例：

```
public class Person
{
    private String name;
    private int age;

    // 无参构造
    public Person()
    {

    }
    // 有参构造
    public Person(String name)
    {
        this();     // 调用本类无参构造器
        this.name = name;
    }
    // 有参构造
    public Person(String name,int age)
    {
        this(name);     // 调用本类中有一个 String 参数的构造器
        this.age = age;
    }

    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        // 当属性名和形参名同名时，必须使用this来区分
        this.name = name;
    }
    public int getAge()
    {
        return age;
    }
    public void setAge(int age)
    {
        this.age = age;
    }
}
```

- this(形参列表)的方式，表示调用当前类中其他的重载的构造器。

7.2 面向对象的特征二：继承性

语法格式：

```
修饰符 class A
{

}

修饰符 class B extends A
{

}
```

子类获取了父类中声明的全部的属性、方法。但是可能受封装性的影响，不能直接调用。

7.2.1 继承性的好处

- 减少了代码的冗余，提高了复用性。
- 提高了扩展性。
- 为多态的实现，提供了前提。

7.2.2 Java 中继承性的特点

- 局限性：类的单继承性。通过类实现接口的方式，解决单继承的局限性。
- 支持多层继承，一个父类可以声明多个子类。

7.3 方法的重写（override 、overwrite）

方法的重载与重写的区别？

- 方法的重载：“两同一不同”，方法名、返回值类型相同，形参列表不同。
- 方法的重写：
    - 前提：类的继承关系。
    - 子类对父类中同名同参数方法的覆盖、覆写。

7.4 super 关键字

super 可以调用的结构：属性、方法、构造器。

super 调用父类的属性、方法：

- 如果子父类中出现了同名的属性，此时使用 super.的方式，表明调用的是父类中声明的属性。
- 子类重写了父类的方法。如果子类的任何一个方法中需要调用父类被重写的方法时，需要使用 super.

super 调用构造器：

- 在子类的构造器中，首行使用了"this(形参列表）“，或者使用了"super(形参列表)”。

7.5 子类对象实例化的全过程

- 结果上：体现为继承性。
- 过程上：子类调用构造器创建对象时，一定会直接或间接的调用其父类的构造器，以及父类的父类的构造器，直到调用到 Object()的构造器。

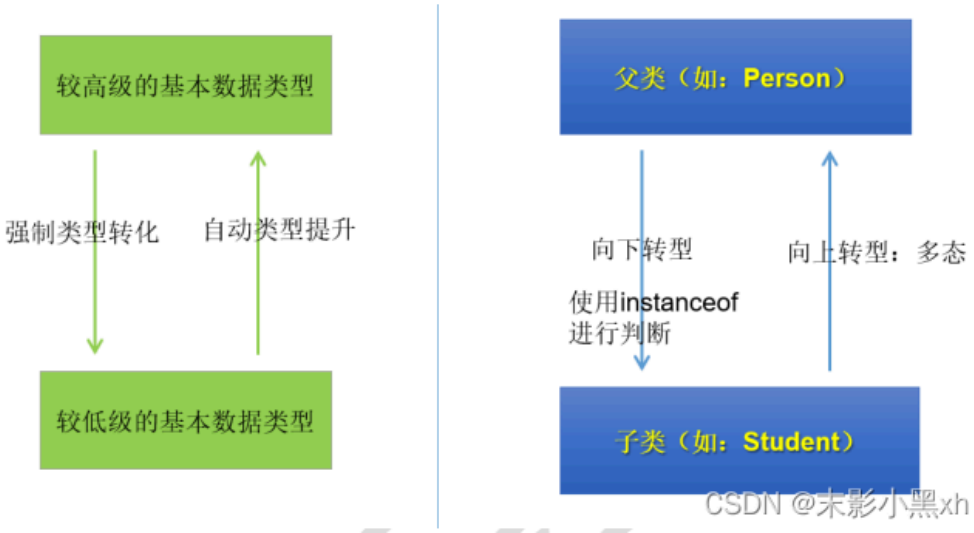
7.6 面向对象的特征三：多态性

- 广义上的理解：子类对象的多态性、方法的重写；方法的重载。
- 狭义上的理解：子类对象的多态性，即父类的引用指向子类的对象。
  语法格式：

```
Object obj = new String(“HelloWorld!”);     // 父类的引用指向子类的对象。
```

- 多态的好处：减少了大量的重载的方法的定义；对扩展开放，对修改关闭原则。
- 举例：public boolean equals(Object obj)
- 多态的使用：虚方法调用（动态链接或晚期绑定）。“编译看左边，运行看右边”。
- 多态的逆过程：向下转型，使用强转符()。
    - 为了避免出现强转时的 ClassCastException，建议()之前使用

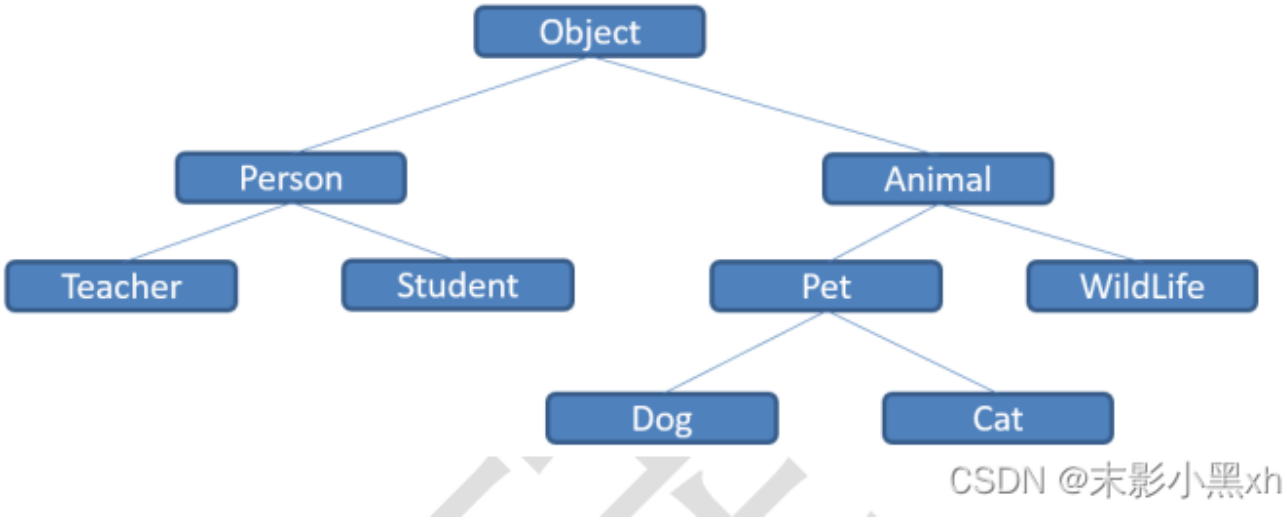
instanceOf 进行判断。



7.7 Object 类

7.7.1 理解根父类

类 java.lang.Object 是类层次结构的根类，即所有其它类的父类。每个类都使用 Object 作为超类。



7.7.2 Object 类的方法

- equals()
    - == ：

        - 基本类型比较值：只要两个变量的值相等，即为 true。
        - 引用类型比较引用(是否指向同一个对象)：只有指向同一个对象时，==才返回 true。
    - equals()：所有类都继承了 Object，也就获得了 equals()方法，可以重写。
        - 只能比较引用类型，Object 类源码中 equals()的作用与“==”相同：比较是否指向同一个对象。
- toString()
    - Object 中 toString()调用后，返回当前对象所属的类和地址值。
    - 开发中常常重写 toString()，用于返回当前对象的属性信息。
- clone()
    - clone() 方法是 Object 类中的一个方法，它用于创建并返回当前对象的一个副本。
    - 该方法是 protected 访问修饰符，只能被子类重写或调用。
    - 在调用 clone() 方法时，如果类没有实现 Cloneable 接口，会抛出 CloneNotSupportedException 异常。因此，要使用 clone() 方法，需要满足两个条件：
        - 类必须实现 Cloneable 接口。
        - 在类中重写 clone() 方法，并将其访问修饰符改为 public。
    - clone() 方法的使用可以避免一些对象拷贝的问题，例如浅拷贝和深拷贝。浅拷贝只复制对象的引用，而不是对象本身，因此当拷贝对象的某个属性发生变化时，原对象和拷贝对象的该属性都会发生变化。深拷贝则是完全复制一个对象，包括其所有属性，因此拷贝对象的属性变化不会影响原对象的属性。
- finalize()
    - 当对象被回收时，系统自动调用该对象的 finalize() 方法。（不是垃圾回收器调用的，
      是本类对象调用的。）
- getClass()
    - public final Class<?> getClass()：获取对象的运行时类型。
- hashCode()
    - public int hashCode()：返回每个对象的 hash 值。

7.2.3 native 关键字

使用 native 关键字说明这个方法是原生函数，也就是这个方法是用 C/C++等非 Java 语言实现的，并且被编译成了 DLL，由 Java 去调用。

7.8 企业真题

1. 父类哪些成员可以被继承，属性可以被继承吗？请举下例子。
   答：
   子类可以继承父类的以下成员：
   ① 非私有的属性和方法（包括静态和实例成员）。
   ② 父类的构造方法。
   子类不能继承父类的私有成员。例如：

```
class Person
{
    public String name;    // 非私有的属性

    public void say()
    {
        // 非私有的方法
        System.out.println("Hello!");
    }

    private int age;    // 私有的属性

    private void walk()
    {
        // 私有的方法
        System.out.println("Walking...");
    }
}

class Student extends Person
{
    // 可以继承父类的非私有属性和方法
    public void study()
    {
        System.out.println("Studying...");
    }

    // 无法继承父类的私有属性和方法
    /*
    public int getAge()
    {
        return age;
    }
    public void go()
    {
        walk();
    }
    */
}
```

在上面的例子中，子类 Student 继承了父类 Person 的 name 属性和 say()方法，但无法继承 age 属性和 walk()方法，因为它们都是私有的。

需要注意的是，子类虽然可以继承父类的属性，但是它们并不属于子类自己的属

性，而是从父类继承而来的。子类可以通过继承来获得父类的属性和方法，但是不能直接访问父类的私有成员。

2. 什么是 Override，与 Overload 的区别?

   答:

   Override 是指在子类中重写父类的方法，使得子类的方法覆盖了父类的方法，具有相同的方法名、参数列表和返回值类型。

   Overload 是指在同一类中定义多个方法，它们具有相同的方法名，但参数列表不同（包括参数类型、个数或顺序），从而实现不同的功能。

   区别在于 Override 是继承关系中子类对父类方法的重写，Overload 是在同一类中定义多个方法，参数列表不同，目的是提供多个方法实现不同的功能。

3. Overload 的方法是否可以改变返回值的类型?

   答:

   Overload 的方法不能仅仅通过返回值类型的改变来进行重载。方法的重载必须要求参数列表不同，而返回值类型不是方法签名的一部分。如果只是改变返回值类型而不改变参数列表，编译器将会报错。

4. 构造器 Constructor 是否可被 override?

   答:

   构造器 Constructor 不能被 override，因为构造器的名字必须与类名相同，并且没有返回值类型。如果子类需要调用父类的构造器，可以使用 super 关键字来实现。但是子类可以提供自己的构造器，可以使用 super 关键字在子类的构造器中调用父类的构造器。

5. 为什么要有重载，我随便命名一个别的函数名不行吗？谈谈你是怎么理解的。

   答:

   Java 中，重载是指在同一个类中，可以定义多个同名但参数类型、个数或顺序不同的方法，这些方法可以根据不同的参数类型和个数进行区分，从而实现不同的功能。

   重载的作用是提高代码的可读性和可维护性。通过重载，可以使用相同的方法名来表示不同的操作，使代码更加简洁明了。同时，重载还可以提高代码的灵活性，可以根据不同的情况选择不同的方法进行调用，从而实现更加精细的控制。

   如果随便命名一个别的函数名，可能会导致命名冲突，增加代码的维护难度，同时也会降低代码的可读性和可维护性。因此，在 Java 中，重载是一种非常重要的语言特性，可以提高代码的可读性和可维护性。

6. super 和 this 的区别?

   答:

   super 和 this 是 Java 中两个关键字，它们的作用不同。

super 表示父类，可以用来访问父类中的属性和方法。在子类中，如果要调用父类中的方法或者属性，可以使用 super 关键字。例如，如果子类中有一个和父类同名的方法或者属性，可以使用 super 关键字来区分调用父类的方法或者属性。super 关键字只能用在子类中。

this 表示当前对象，可以用来访问当前对象的属性和方法。在一个类中，如果要调用当前对象的属性或者方法，可以使用 this 关键字。例如，如果一个类中有一个成员变量和一个局部变量同名，可以使用 this 关键字来区分访问成员变量或者局部变量。this 关键字可以在任何地方使用。

总的来说，super 和 this 的作用不同，super 用于访问父类的属性和方法，this 用于访问当前对象的属性和方法。

7. this、super 关键字分别代表什么?以及他们各自的使用场景和作用。

   答：

   this 和 super 是 Java 中的关键字，分别代表当前对象和父类对象。

   this 关键字代表当前对象，可以用来引用当前对象的属性和方法。this 关键字主要用于以下场景：

   ① 当局部变量和成员变量同名时，使用 this 关键字可以区分二者，如 this.name 表示当前对象的 name 属性，name 表示局部变量。

   ② 在构造器中调用另一个构造器时，使用 this 关键字可以调用同一个类中的其他构造器，如 this(name)，表示调用该类中的带有一个参数的构造器。

   ③ 在方法链式调用时，使用 this 关键字可以返回当前对象，如 return this。

   super 关键字代表父类对象，可以用来调用父类的属性和方法。super 关键字主要用于以下场景：

   ① 在子类中调用父类的构造方法时，使用 super 关键字可以调用父类的构造方法，如 super()，表示调用父类的无参构造器。

   ② 当子类和父类有同名属性或方法时，使用 super 关键字可以调用父类的属性或方法，如 super.name 表示调用父类的 name 属性。

   总的来说，this 关键字用于当前对象中，super 关键字用于父类对象中。它们的使用场景和作用不同，但都可以方便地引用当前对象或父类对象的属性和方法。

8. 谈谈你对多态的理解。

   答：

   多态是面向对象程序设计中的重要概念，它是指同一个方法可以根据不同的对象调用出不同的行为。具体来说，多态包括两种类型：静态多态和动态多态。

   静态多态是指在编译阶段就确定了调用的方法，主要通过方法的重载和参数的多态来实现。例如，同一个类中可以有多个同名方法，但是参数类型或个数不同，编译器在编译时会根据参数类型和个数来自动匹配调用哪个方法。

动态多态是指在运行时根据实际对象的类型来确定调用的方法，主要通过方法的重写和父类引用指向子类对象来实现。例如，一个父类引用可以指向其子类对象，当调用该引用的方法时，实际调用的是子类的方法。

多态的优点在于可以增加程序的灵活性和可扩展性，使代码更加简洁、可读性更高。它可以减少代码的重复，使程序更易于维护和修改。同时，多态也是面向对象程序设计的核心思想之一，它有助于提高代码的可重用性和可维护性，增强了程序的可扩展性和可靠性。

9. 多态 new 出来的对象跟不多态 new 出来的对象区别在哪?

   答:

   多态 new 出来的对象是基于父类或接口创建的，可以根据实际情况指向不同的子类对象，具有更强的灵活性和可扩展性；而不多态 new 出来的对象是直接创建的具体子类对象，无法在运行时进行动态绑定，缺乏灵活性和可扩展性。

10. 说说你认为多态在代码中的体现。

    答:

    多态是指同一种操作作用于不同的对象上面，可以产生不同的执行结果。在 Java 中，多态主要通过以下三种方式体现:

    ① 方法重载：方法重载是指在同一个类中，有多个方法名相同但参数类型或个数不同的方法。这种方法的调用会根据传入的参数类型或个数的不同，自动匹配调用相应的方法，实现了多态。

    ② 方法重写：方法重写是指子类重写了父类的方法，当调用这个方法时，实际上会根据对象的实际类型调用对应的方法。这种方法实现了运行时多态。

    ③ 接口实现：接口是一种规范，定义了一组方法的签名，实现接口的类必须实现这些方法。当一个类实现了多个接口时，可以根据需要选择调用不同的方法，实现了多态。

11. == 与 equals() 的区别?

    答:

    在 Java 中，== 是一个操作符，用于比较两个对象的内存地址是否相同，即它们是否是同一个对象。而 equals()是一个方法，用于比较两个对象的内容是否相同，即它们是否具有相同的属性值。

    在 Java 中，所有的类都继承自 Object 类，Object 类中的 equals()方法默认使用==比较两个对象的内存地址。因此，如果我们想要比较两个对象的内容是否相同，需要重写 equals()方法来实现比较对象属性值的操作。

    例如，我们可以在自定义类中重写 equals()方法，比较对象的属性值是否相同，如下所示:

```java
public class Person
{
    private String name;
    private int age;

    // 构造方法和其他方法省略
    @Override
    public boolean equals(Object o)
    {
        if (this == o)
          return true;
        if (!(o instanceof Person))
          return false;

        Person person = (Person) o;

        return age == person.age &&
                Objects.equals(name, person.name);
    }
}
```

在这个例子中，我们重写了 Person 类的 equals()方法，首先使用==比较两个对象的内存地址，如果相同则返回 true，否则判断 o 是否是 Person 类的实例，如果不是则返回 false，最后比较两个对象的属性值是否相同，如果相同则返回 true，否则返回 false。

在使用 equals()方法比较两个对象时，需要注意以下几点：

① equals()方法必须满足自反性、对称性、传递性和一致性等特性；

② equals()方法的参数必须是 Object 类型，需要进行类型转换；

③ equals()方法比较两个对象的属性值时，需要使用 Objects.equals()方法，避免空指针异常。

12. 重写 equals()方法要注意什么？

答：

重写 equals()方法是为了比较两个对象是否相等，因此需要注意以下几点：

① 对象比较要使用 equals()方法，而不是“==”运算符。

② 重写 equals()方法时，需要重写 hashCode()方法，保证相等的对象具有相同的哈希码。

③ equals()方法必须具有自反性、对称性、传递性和一致性。

④ equals()方法的参数必须是 Object 类型，需要进行类型检查和类型转换。

⑤ 在比较对象的属性时，需要逐个比较所有属性，而不是只比较其中的几个属性。

⑥ 如果子类中增加了新的属性，需要在 equals()方法中同时比较新的属性。
⑦ 如果父类已经实现了 equals()方法，子类可以选择继承父类的 equals()方法，也可以重写 equals()方法。如果重写了 equals()方法，需要调用父类的 equals()方法进行比较。

13. Java 中所有类的父类是什么？他都有什么方法?
    答：
    Java 中所有类的父类是 Object 类。
    Object 类中常用的方法有：
    ① toString()：返回对象的字符串表示形式。
    ② equals(Object obj)：判断两个对象是否相等。
    ③ hashCode()：返回对象的哈希码值。
    ④ getClass()：返回对象的类。
    ⑤ wait()：使当前线程等待。
    ⑥ notify()：唤醒正在等待该对象的线程。
    ⑦ notifyAll()：唤醒正在等待该对象的所有线程。
    ⑧ finalize()：在对象被垃圾回收器回收之前调用。

第 8 章 面向对象——高级

8.1 static 关键字

- static 使用范围：
    - 在 Java 类中，可用 static 修饰属性、方法、代码块、内部类。
- 语法格式：

```
修饰符 class 类
{
    修饰符 static 数据类型 变量名;

    修饰符 static 返回值类型 方法名(形参列表)
    {
      方法体
    }
}
```

- static 修饰后的成员具备以下特点：

    - 静态的，随着类的加载而加载、执行。
    - 随着类的加载而加载。
    - 优先于对象存在。
    - 修饰的成员，被所有对象所共享。
    - 访问权限允许时，可不创建对象，直接被类调用。
- 类变量：类的生命周期内，只有一个。被类的多个实例共享。

8.2 单例模式

经典的设计模式有 23 种。每个设计模式均是特定环境下特定问题的处理方法。

> 对软件设计模式的研究造就了一本可能是面向对象设计方面最有影响的书籍：《设计模式》：《Design Patterns: Elements of Reusable Object-Oriented Software》，由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 合著（Addison-Wesley，1995）。这几位作者常被称为"四人组（Gang of Four）"，而这本书也就被称为"四人组（或 GoF）"书。

单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。

实现方式：饿汉式、懒汉式、枚举类等。

饿汉式代码示例：

```
lass Singleton
{
    // 1.私有化构造器
    private Singleton()
    {

    }

    // 2.内部提供一个当前类的实例
    // 4.此实例也必须静态化
    private static Singleton single = new Singleton();

    // 3.提供公共的静态的方法，返回当前类的对象
    public static Singleton getInstance()
    {
        return single;
    }
  }
```

懒汉式代码示例：

```
class Singleton
{
  // 1.私有化构造器
  private Singleton()
  {

  }

  // 2.内部提供一个当前类的实例
  // 4.此实例也必须静态化
  private static Singleton single;

  // 3.提供公共的静态的方法，返回当前类的对象
  public static Singleton getInstance()
  {
      if(single == null)
      {
        single = new Singleton();
      }

    return single;
  }
}
```

饿汉式和懒汉式的区别：

- 饿汉式： “立即加载”，线程安全的。
- 懒汉式： “延迟加载”，线程不安全。

8.3 理解 main()方法的语法

```
public static void main(String[] args)
{

}
```

由于 JVM 需要调用类的 main()方法，所以该方法的访问权限必须是 public。

又因为 JVM 在执行 main()方法时不必创建对象，所以该方法必须是 static 的，该方法接收一个 String 类型的数组参数，该数组中保存执行 Java 命令时传递给所运行的类的参数。

又因为 main() 方法是静态的，我们不能直接访问该类中的非静态成员，必须创建该类的一个实例对象后，才能通过这个对象去访问类中的非静态成员。

8.4 类的成员之四：代码块

- 分类：静态代码块、非静态代码块。
- 语法格式：

```
修饰符 class 类
{
    static
    {
        静态代码块
    }
}
```

- 静态代码块：随着类的加载而执行。
- 非静态代码块：随着对象的创建而执行。
- 总结

  对象的实例变量可以赋值顺序：

8.5 final 关键字

final：最终的，不可更改的。

用来修饰：类、方法、变量（成员变量、局部变量）。

- 类：不能被继承。
- 方法：不能被重写。
- 变量：是一个“常量”，一旦赋值不能修改。

8.6 abstract 关键字

随着继承层次中一个个新子类的定义，类变得越来越具体，而父类则更一般，更通用。类的设计应该保证父类和子类能够共享特征。有时将一个父类设计得非常抽象，以至于它没有具体的实例，这样的类叫做抽象类。

abstract：抽象的。

抽象类的语法格式：

```
修饰符 abstract class 类名
{

}

修饰符 abstract class 类名 extends 父类
{

}
```

抽象方法的语法格式：

```
修饰符 abstract 返回值类型 方法名([形参列表]);
```

用来修饰：类、方法。

- 类：抽象类：不能实例化。
- 方法：抽象方法：没有方法体，必须由子类实现此方法。

模板方法模式（TemplateMethod）：

- 抽象类体现的就是一种模板模式的设计，抽象类作为多个子类的通用模板，子类在抽象类的基础上进行扩展、改造，但子类总体上会保留抽象类的行为方式。

8.7 interface 关键字

Java 的软件系统会有很多模块组成，那么各个模块之间也应该采用这种面向接口的低耦合，为系统提供更好的可扩展性和可维护性。

interface：接口，用来定义一组规范、一种标准。

接口声明的语法格式：

```
修饰符 interface 接口名
{
    接口的成员列表
    公共的静态常量
    公共的抽象方法

    公共的默认方法（JDK1.8 以上）
    公共的静态方法（JDK1.8 以上）
    私有方法（JDK1.9 以上）
}
```

在 JDK8.0 之前，接口中只允许声明：
① 公共的静态的常量。
② 公共的抽象的方法。

- 在 JDK8.0 时，接口中允许声明默认方法和静态方法。

类实现接口（implements）：

- 接口不能创建对象，但是可以被类实现（implements ，类似于被继承）。
- 类与接口的关系为实现关系，即类实现接口，该类可以称为接口的实现类。

接口实现的语法格式：

```
修饰符 class 实现类 implements 接口
{
    重写接口中抽象方法
    重写接口中默认方法
}

修饰符 class 实现类 extends 父类 implements 接口
{
    重写接口中抽象方法
    重写接口中默认方法
}
```

接口可以多继承、多实现。

接口与抽象类之间的对比：

No.	区别点	抽象类	接口
1	定义	可以包含抽象方法的类	主要是抽象方法和全局常量的集合
2	组成	构造方法、抽象方法、普通方法、常量、变量	常量、抽象方法、(jdk8.0:默认方法、静态方法)
3	使用	子类继承抽象类(extends)	子类实现接口(implements)
4	关系	抽象类可以实现多个接口	接口不能继承抽象类，但允许继承多个接口
5	常见设计模式	模板方法	简单工厂、工厂方法、代理模式
6	对象	都通过对象的多态性产生实例化对象	
7	局限	抽象类有单继承的局限	接口没有此局限
8	实际	作为一个模板	是作为一个标准或是表示一种能力
9	选择	如果抽象类和接口都可以使用的话，优先使用接口，因为避免单继承的局限	

8.8 类的成员之五：内部类

将一个类 A 定义在另一个类 B 里面，里面的那个类 A 就称为内部类，类 B 则称为外部类。

- 成员内部类：
  成员内部类的语法格式：

```
修饰符 class 外部类
{
    修饰符 static class 内部类
    {

    }
}
```

- 非匿名局部内部类：
  非匿名局部内部类的语法格式：

```
修饰符 class 外部类
{
    修饰符 返回值类型 方法名(形参列表)
    {
        final/abstract class 内部类
        {

        }
    }
}
```

- 匿名内部类：
  匿名内部类的语法格式：

```
new 父接口()
{
    重写方法
}

new 父类(实参列表)
{
    重写方法
}
```

8.9 枚举类：enum

枚举类型本质上也是一种类，只不过是这个类的对象是有限的、固定的几个，不能让用户随意创建。

使用 enum 关键字定义枚举类。

语法格式：

```
修饰符 enum 枚举类名
{
    常量对象列表
}

修饰符 enum 枚举类名
{
    常量对象列表;

    对象的实例变量列表;
}
```

8.10 注解：Annotation

注解（Annotation）是从 JDK5.0 开始引入，以“@注解名”在代码中存在。

Annotation 可以像修饰符一样被使用，可用于修饰包、类、构造器、方法、成员变量、参数、局部变量的声明。还可以添加一些参数值，这些信息被保存在 Annotation 的“name=value” 对中。

注解可以在类编译、运行时进行加载，体现不同的功能。

常用注解：

- @Override: 限定重写父类方法，该注解只能用于方法。
- @Deprecated: 用于表示所修饰的元素(类，方法等)已过时。通常是因为所修饰的结构危险或存在更好的选择。
- @SuppressWarnings: 抑制编译器警告。

元注解：对现有的注解进行解释说明。

- @Target：表明可以用来修饰的结构。
- @Retation：表明生命周期。
- @Documented：表明这个注解应该被 javadoc 工具记录。
- @Inherited：允许子类继承父类中的注解如何自定义注解。

自定义注解的使用：

- 一个完整的注解应该包含三个部分： ① 声明 ② 使用 ③ 读取
- 声明自定义注解 ：
  语法格式：

```
元注解
修饰符 @interface 注解名
{
    成员列表
}
```

> 框架 = 注解 + 反射 + 设计模式

8.11 JUnit 单元测试

- 测试分类：
  - 黑盒测试：不需要写代码，给输入值，看程序是否能够输出期望的值。
  - 白盒测试：需要写代码的。关注程序具体的执行流程。
- JUnit 是由 Erich Gamma 和 Kent Beck 编写的一个测试框架（regression testing framework），供 Java 开发人员编写单元测试之用。
- 编写和运行@Test 单元测试方法：
- JUnit4 版本，要求@Test 标记的方法必须满足如下要求：
  - 所在的类必须是 public 的，非抽象的，包含唯一的无参构造器。
  - @Test 标记的方法本身必须是 public，非抽象的，非静态的，void 无返回值，无参数的。

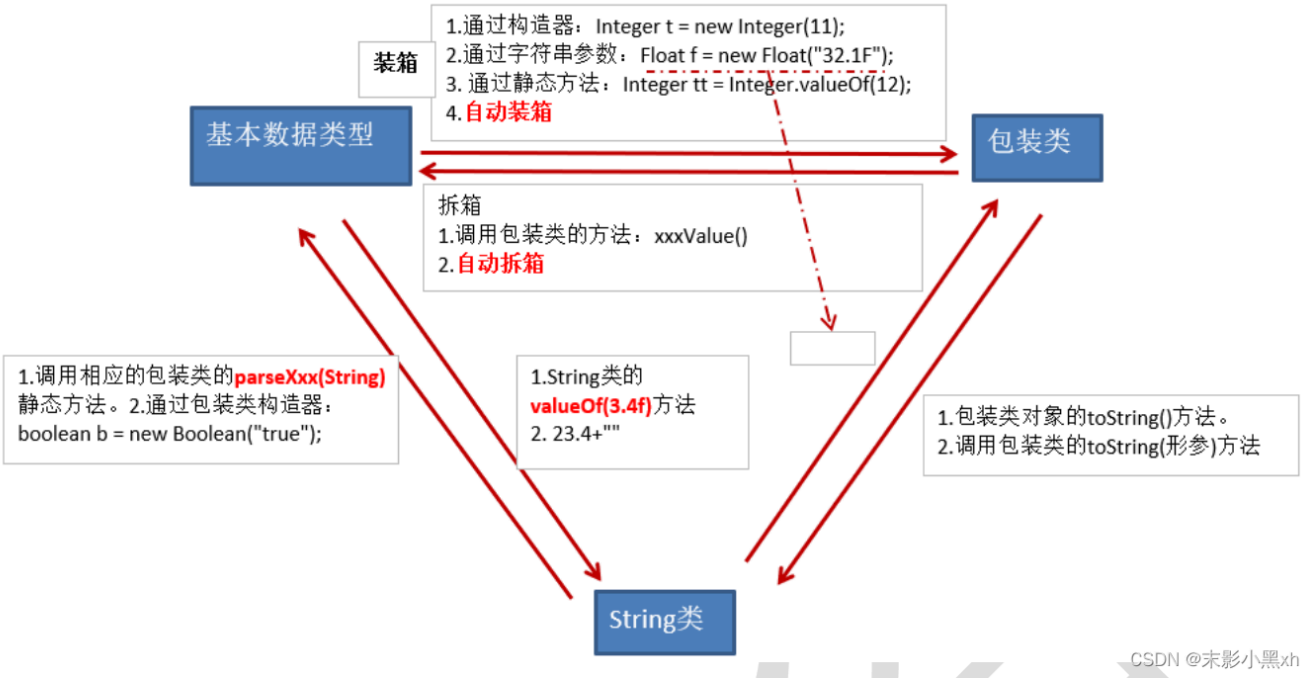
8.12 包装类

基本数据类型对应的包装类都有哪些?

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

父类:Number

基本数据类型、包装类、String 三者之间的转换：



基本数据类型 <——> 包装类：自动装箱、自动拆箱。

String 类的 valueOf(xxx)方法。

包装类的 parseXxx(String str)方法。

8.13 IDEA 的 Debug

程序在执行过程中如果出现错误，该如何查找或定位错误呢？简单的代码直接就可以看出来，但如果代码比较复杂，就需要借助程序调试工具（Debug）来查找错误了。

Debug(调试)程序步骤如下：
① 添加断点。
② 启动调试。
③ 单步执行。
④ 观察变量和执行流程，找到并解决问题。

8.14 企业真题

1. 静态变量和实例变量的区别？
   答：
   静态变量是属于类的变量，只有一个副本，被该类的所有实例共享，可以通过类名访问。实例变量是属于对象的变量，每个对象都有自己的一份副本，互相独立，只能通过实例名访问。静态变量在类加载时初始化，实例变量在实例化对象时初始化。静态变量一般用于存储常量或类级别的变量，实例变量一般用于存储对象的状态信息。
2. 静态属性和静态方法是否可以被继承？是否可以被重写？以及原因？
   答：
   静态属性和静态方法可以被继承，但不能被重写。
   静态属性和静态方法是属于类的，而不是属于实例的。子类继承父类时，会继承父类的静态属性和静态方法，但是子类无法重写父类的静态属性和静态方法，因为静态属性和静态方法是属于类的，而不是属于对象的。在子类中可以使用父类的静态属性和静态方法，也可以定义自己的静态属性和静态方法，但是不能重写父类的静态属性和静态方法。
3. 是否可以从一个 static 方法内部发出对非 static 方法的调用？
   答：
   可以从一个 static 方法内部发出对非 static 方法的调用，但前提是必须先创建一个对象实例，然后通过该实例来调用非 static 方法。因为非 static 方法是属于对象的，需要通过对象实例才能调用，而 static 方法是属于类的，可以直接通过类名调用。如果在 static 方法中要调用非 static 方法，必须先创建一个对象实例，然后通过该实例来调用非 static 方法。例如：

```
public class MyClass
{
    private int num;

    public void nonStaticMethod()
    {
        System.out.println("这是一个非静态方法。");
    }

    public static void staticMethod()
    {
        MyClass obj = new MyClass();
        obj.nonStaticMethod();
        System.out.println("这是一个静态方法。");
    }
}
```

在上面的代码中，staticMethod() 中创建了一个 MyClass 的对象实例 obj，并通过该实例来调用 nonStaticMethod()。

4. 被 static 修饰的成员(类、方法、成员变量)能否再使用 private 进行修饰?
   答:
   可以。被 static 修饰的成员可以再使用 private 进行修饰，表示该成员只能在当前类中被访问，不能被其他类访问。这样可以保证该成员的访问权限更加严格，增加程序的安全性。
5. 知道哪些设计模式?
   答:
   我知道很多设计模式，包括： - 单例模式 - 工厂模式 - 抽象工厂模式 - 建造者模式 - 原型模式 - 适配器模式 - 桥接模式 - 组合模式 - 装饰器模式 - 外观模式 - 享元模式 - 代理模式 - 责任链模式 - 命令模式 - 解释器模式 - 迭代器模式 - 中介者模式 - 备忘录模式 - 观察者模式 - 状态模式 - 策略模式 - 模板方法模式 - 访问者模式
   当然，这只是其中的一部分，设计模式还有很多种，每种模式都有自己的应用场景和优缺点。
6. 开发中都用到了那些设计模式?用在什么场合?
   答:
   在开发中常用的设计模式包括:
   ① 单例模式：保证一个类只有一个实例，并提供全局访问点。
   ② 工厂模式：将对象的创建和使用分离，通过工厂类来创建对象。
   ③ 观察者模式：一对多的依赖关系，当一个对象的状态发生改变时，所有依赖它的

对象都会得到通知。

④ 建造者模式：将一个复杂对象的构建过程分解为多个简单对象的构建过程，使得构建过程灵活性增强，且更易于扩展。

⑤ 适配器模式：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的类可以一起工作。

⑥ 装饰器模式：动态地给一个对象添加一些额外的职责，而不需要修改原始类的代码。

⑦ 策略模式：定义一系列算法，将它们封装起来，并且使它们可以互相替换，使得算法的变化独立于使用算法的客户。

这些设计模式都有各自的使用场合，如单例模式可以用于创建全局唯一的对象；工厂模式可以用于创建一系列相关对象；观察者模式可以用于实现事件驱动的程序设计；建造者模式可以用于构建复杂对象；适配器模式可以用于不同接口之间的转换；装饰器模式可以用于扩展对象的功能；策略模式可以用于解耦算法的实现和使用。

7. main()方法的 public 能不能换成 private，为什么?

   答:

   不可以。因为 main()方法是程序的入口，如果将其访问修饰符改为 private，那么程序在启动时就无法访问该方法，无法执行程序。因此，main()方法必须是 public 修饰，以便程序能够访问执行。

8. main()方法中是否可以调用非静态方法?

   答:

   可以调用非静态方法，但需要先创建该方法所在类的对象，然后通过对象来调用非静态方法。

9. 类的组成和属性赋值执行顺序?

   答:

   类的组成包括类名、属性、方法和构造函数。属性赋值执行顺序是在创建类的实例时执行的，先执行构造函数，然后按照属性的定义顺序依次执行属性的赋值操作。如果属性有默认值，则先执行默认值的赋值操作，然后再执行构造函数中的赋值操作。在属性赋值时，如果属性有 setter 方法，则会调用 setter 方法来完成赋值操作。

10. 静态代码块，普通代码块，构造方法，从类加载开始的执行顺序?

    答:

    ① 静态代码块：在类加载时执行，只执行一次；

    ② 普通代码块：在创建对象时执行，每次创建对象都会执行一次；

    ③ 构造方法：在创建对象时执行，每次创建对象都会执行一次。

执行顺序为：静态代码块 → 普通代码块 → 构造方法。

11. 描述一下对 final 理解。

    答：

    final 是 Java 中的一个关键字，可以用来修饰类、方法和变量。final 关键字的主要作用是：

    ① final 修饰的类不能被继承，即该类为最终类。

    ② final 修饰的方法不能被子类重写，即该方法为最终方法。

    ③ final 修饰的变量为常量，即该变量的值不能被修改，一旦被赋值后就不能再改变。

    final 关键字的使用可以提高程序的安全性和效率，因为 final 修饰的类、方法和变量在程序运行时无法被修改，从而防止了程序中的错误和不必要的开销。同时，final 还可以用来定义常量，方便程序中的使用和维护。

12. 使用 final 修饰一个变量时，是引用不能改变，引用指向的对象可以改变?

    答：

    如果 final 修饰的是一个引用变量，则该引用变量所指向的对象不能被改变，但该对象的属性值可以被改变。如果 final 修饰的是一个基本数据类型变量，则该变量的值不能被改变。

13. final 不能用于修饰构造方法?

    答：

    是的，final 关键字不能用于修饰构造方法。final 关键字可以用于修饰类、成员变量和方法，但不能用于构造方法。这是因为构造方法的主要作用是初始化对象的状态，而 final 关键字表示不可变性，这与构造方法的目的不符。

14. final 或 static final 修饰成员变量，能不能进行++操作?

    答：

    无法进行++操作。final 修饰的成员变量是常量，不可被修改；而 static final 修饰的成员变量是静态常量，也不可被修改。因此，++操作是不允许的。

15. 什么是抽象类？如何识别一个抽象类?

    答：

    抽象类是一种不能被实例化的类，其目的是为了被其他类继承而设计的。抽象类中可以包含抽象方法和非抽象方法，抽象方法是指没有实现的方法，需要在子类中被实现。非抽象方法是指已经实现的方法，可以直接在抽象类中调用。

    抽象类可以通过关键字"abstract"来定义。如果一个类中包含至少一个抽象方法，那么该类必须被定义为抽象类。抽象类不能被实例化，只能被继承，并且子类必须实现所有抽象方法，否则子类也必须被定义为抽象类。

    以下是一个抽象类的例子：

```java
public abstract class Animal
{
    public abstract void makeSound();
    public void eat()
    {
        System.out.println("我在吃东西。");
    }
}
```

在上面的例子中，Animal 是一个抽象类，其中包含一个抽象方法 makeSound()和一个非抽象方法 eat()。makeSound()方法必须在子类中被实现，而 eat()方法已经被实现，可以直接在 Animal 类中调用。

16. 为什么不能用 abstract 修饰属性、私有方法、构造器、静态方法、final 的方法?
    答:
    ① 属性：抽象属性没有实现，无法在子类中重写，因此没有意义。
    ② 私有方法：私有方法只能在本类中被调用，无法在子类中重写，因此没有意义。
    ③ 构造器：构造器用于创建对象，不能被重写，因此没有意义。
    ④ 静态方法：静态方法属于类，不属于实例，无法被重写，因此没有意义。
    ⑤ final 方法：final 方法表示该方法不能被重写，因此使用 abstract 修饰没有意义。
17. 接口与抽象类的区别?
    答:
    ① 定义方式不同：接口使用 interface 关键字定义，抽象类使用 abstract 关键字定义。
    ② 实现方式不同：类可以实现多个接口，但只能继承一个抽象类。
    ③ 方法实现方式不同：接口中的方法都是抽象方法，没有方法体，实现类必须重写所有接口中的方法；抽象类中可以包含抽象方法和非抽象方法，实现类需要重写抽象方法，可以选择性地重写非抽象方法。
    ④ 成员变量不同：接口中只能定义常量，而抽象类可以定义普通成员变量。
    ⑤ 构造方法不同：接口中不能定义构造方法，而抽象类可以定义构造方法。
    ⑥ 目的不同：接口用于定义一组规范，而抽象类用于被继承。
18. 接口是否可继承接口？抽象类是否可实现（implements）接口？抽象类是否可继承实现类（concrete class）？
    答:
    接口是可以继承接口的，这个过程和类之间的继承类似，子接口继承父接口的方法和常量，并可以在子接口中添加新的方法和常量。

抽象类可以通过实现接口来实现接口的方法，这个过程和普通类实现接口的过程一样。

抽象类也可以继承实现类，但是这种做法不太常见，因为实现类已经实现了接口中的方法，而抽象类的目的是为了让子类来实现，这样就会导致代码的重复和混淆。

19. 接口可以有自己属性吗?

    答:

    在面向对象编程中，接口是一个纯粹的抽象概念，不应该包含实现代码或属性。接口只定义了类应该具有的方法和属性，但不提供它们的实现。因此，接口本身不应该有任何属性。

20. 访问接口的默认方法如何使用?

    答:

    访问接口的默认方法可以通过实现该接口的类来调用。默认方法在接口中被定义，但是可以在实现类中被重写或者调用。如果实现类没有重写接口中的默认方法，那么默认方法将被继承并使用。

    以下是访问接口默认方法的示例代码:

```
interface MyInterface
{
    default void myMethod()
    {
        System.out.println("这是一个默认方法。");
    }
}

    class MyClass implements MyInterface
    {
        // 此类不会重写接口中的默认方法。
    }

    public class Main
    {
        public static void main(String[] args)
        {
            MyClass obj = new MyClass();
            obj.myMethod();     //输出：这是一个默认方法。
        }
    }
```

    在上面的代码中，我们定义了一个接口 MyInterface，其中包含了一个默认方法 myMethod()。然后，我们通过实现该接口的类 MyClass 来调用默认方法。在

main()方法中，我们创建了一个 MyClass 对象并调用了 myMethod()方法，输出了默认方法的内容。

21. 内部类有哪几种?

    答:

    内部类分为四种:

    ① 成员内部类（也称为普通内部类）

    ② 静态内部类

    ③ 局部内部类

    ④ 匿名内部类

22. 内部类的特点说一下。

    答:

    ① 内部类是定义在另一个类内部的类，在外部类的范围内声明，但是不能独立存在。

    ② 内部类可以访问外部类的所有成员，包括私有成员。

    ③ 内部类可以使用外部类的引用，通过 this 关键字来访问外部类的成员。

    ④ 内部类可以被 private、protected、public 和 static 修饰，可以作为外部类的成员或局部变量。

    ⑤ 内部类可以继承其他类或实现接口，可以被其他类继承或实现。

    ⑥ 内部类可以访问外部类的私有构造方法，可以用来实现单例模式。

    ⑦ 内部类可以被用来实现回调机制，通过实现接口来实现回调函数。

23. 枚举可以继承吗?

    答:

    枚举类型不能被继承，因为它们已经是最终的类型。枚举类型是一种特殊的值类型，它们的值是固定的且不可修改，因此没有必要对其进行继承。

24. Java 基本类型与包装类的区别?

    答:

    Java 基本类型是指 Java 语言中最基本的数据类型，包括整数类型、浮点数类型、字符类型和布尔类型，其值是直接存储在内存中的。而包装类是一种特殊的类，用于将基本类型转换为对象，以便在面向对象的环境中使用。包装类提供了一些方法，使得基本类型可以像对象一样进行操作。

    Java 基本类型与包装类的区别主要有以下几点:

    ① 基本类型的变量直接存储值，而包装类的对象存储的是值的引用。

    ② 基本类型的变量在内存中占用的空间比包装类的对象小。

    ③ 基本类型的变量不能为 null，而包装类的对象可以为 null。

    ④ 基本类型的变量不能调用方法，而包装类的对象可以调用方法。

⑤ 基本类型的变量可以直接进行算术运算，而包装类的对象需要通过方法进行运算。

⑥ 基本类型的变量可以直接赋值，而包装类的对象需要使用构造方法或 valueOf() 方法进行赋值。

⑦ 基本类型的变量在方法中传递时是按值传递，而包装类的对象在方法中传递时是按引用传递。

25. 谈谈你对面向对象的理解?

答:

面向对象是一种编程范式，它将现实世界中的事物抽象成对象，并将对象之间的关系封装成类，从而实现程序的模块化、复用和扩展。面向对象的编程主要基于三个核心概念：封装、继承和多态。

封装是指将数据和行为封装在一个对象中，对外部只暴露必要的接口，隐藏内部实现细节，从而保证数据的安全性和可靠性。

继承是指一个类可以继承另一个类的属性和方法，从而减少代码的重复和冗余，提高代码的复用性和可维护性。

多态是指同一种行为具有不同的表现形式，不同的对象可以对同一消息作出不同的响应，从而增强代码的灵活性和可扩展性。

面向对象的编程思想可以使程序更加易于理解、扩展和维护，同时也可以提高代码的可复用性和可测试性，因此在现代软件开发中得到了广泛的应用。

26. 面向对象的特征有哪些方面?

答:

面向对象的特征有以下几个方面:

① 封装性：将数据和操作数据的方法封装在一起，对外部隐藏其内部实现细节，只提供公共接口，以保证数据的安全性和完整性。

② 继承性：通过继承机制，子类可以继承父类的属性和方法，从而减少重复代码，提高代码的复用性和可维护性。

③ 多态性：同一种类的对象在不同的情况下表现出不同的行为，即一个方法可以有多个不同的实现方式，提高了代码的灵活性和可扩展性。

④ 抽象性：通过抽象类和接口，将对象的共性抽象出来，从而使得代码更加简洁、易于理解和维护。

⑤ 组合性：通过将多个对象组合在一起，形成一个更加复杂的对象，从而提高了代码的可拓展性和可复用性。