# 封装

答：封装的主要目的是隐藏对象的内部状态和实现细节，并对外界提供一个稳定的接口来与该对象进行交互。通过限制对对象内部结构的访问，可以提高代码的安全性并降低其复杂度。这意味着用户不需要了解对象内部是如何工作的就能使用该对象的功能，同时也使得对象的内部实现可以在不影响外部使用者的情况下进行更改。

# 构造器

答：在某些编程语言中，如Java，如果一个类没有显式地定义任何构造器，那么编译器会自动为这个类添加一个默认的无参数构造器。这个默认构造器是一个无参构造器，它没有任何参数，并且仅执行一些基本的初始化工作，比如调用父类的默认构造器（如果有的话），并可能对非静态字段设置默认值（例如，int类型的默认值为0，String类型的默认值为null等）。

# 封装考核

答：代码：

1. public class Test03 {

2.     public static void main(String[] args) {

3.     Student s = new Student("张三", 19, "男", "10000");

4.     System.out.println(s.getName()); //输出张三

5.     System.out.println(s.getAge()); //输出19

6.     s.setName("李四");

7.     System.out.println(s.getName()); //输出李四

8.     }

9. }

10. class Student {

11.     private String name;

12.     private int age;

13.     private String sex;

14.     private String id;

15.     public Student(){

16.     }

17.     public Student(String name,int age,String sex,String id){

18.         this.name=name;

19.         this.age=age;

20.         this.sex=sex;

21.         this.id=id;

22.     }

23.     public void setName(String name){

24.         this.name=name;

25.     }

26.     public String getName(){

27.         return name;

28.     }

29.     public int getAge(){

30.         return age;

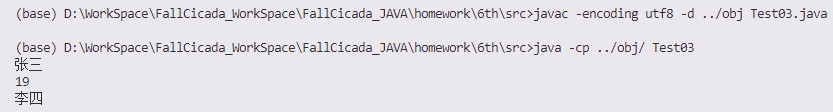
31.     }

32.

33. }

34.

运行结果：



# 面向对象考核

答：代码

1. // 定义一个人类

2. class Person {

3.     private String name; // 姓名

4.

5.     public Person(String name) {

6.         this.name = name;

7.     }

8.

9.     // 方法：让这个人开车

10.     public void drive(Car car, String destination) {

11.         System.out.println(name + " 开着 " + car.getModel() + " 去 " + destination);

12.     }

13. }

14.

15. // 定义一个交通工具基类

16. class Vehicle {

17.     private String model; // 模型

18.

19.     public Vehicle(String model) {

20.         this.model = model;

21.     }

22.

23.     public String getModel() {

24.         return model;

25.     }

26. }

27.

28. // 定义一个汽车类，继承自交通工具基类

29. class Car extends Vehicle {

30.     public Car(String model) {

31.         super(model);

32.     }

33. }

34.

35. public class Test04 {

36.     public static void main(String[] args) {

37.         // 创建一个叫阿臣的人

38.         Person zhangSan = new Person("阿臣");

39.

40.         // 创建一辆汽车

41.         Car car = new Car("小米 su7");

42.

43.         // 张三开着车去公司

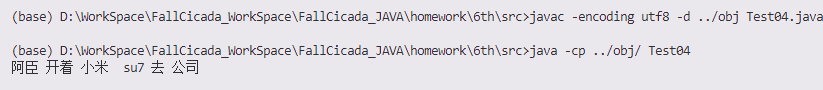
44.         zhangSan.drive(car, "公司");

45.     }

46. }

47.

运行结果：



# 构造器考核

需求分析：根据要求编写Duration(时长)类，编写完成后使用下面的测试类去测试，要求如下

Duration 类：

1. 定义3个整型属性：hours、minutes、seconds

2. 定义三参构造器，实现对hours、minutes、seconds进行初始化

3. 定义一参构造器，实现对hours、minutes、seconds进行初始化，参数为总 的seconds，例如将x秒，转为a小时b分钟c秒，3661秒，就是1小时1分钟1秒

4. 定义每个属性对应的get方法，实现获取属性值

5. 定义 public int getTotalSeconds(){} 方法，实现返回总的秒数值

6. 定义 public String toString() {} 方法，方法返回内容为时分秒信 息，格式为

代码：

1. public class Duration {

2.     private int hours;

3.     private int minutes;

4.     private int seconds;

5.

6.     // 三参构造器

7.     public Duration(int hours, int minutes, int seconds) {

8.         this.hours = hours;

9.         this.minutes = minutes;

10.         this.seconds = seconds;

11.         normalize();

12.     }

13.

14.     // 单参构造器，参数为总的秒数

15.     public Duration(int totalSeconds) {

16.         this.hours = totalSeconds / 3600;

17.         totalSeconds %= 3600;

18.         this.minutes = totalSeconds / 60;

19.         this.seconds = totalSeconds % 60;

20.     }

21.

22.     // 获取小时数

23.     public int getHours() {

24.         return hours;

25.     }

26.

27.     // 获取分钟数

28.     public int getMinutes() {

29.         return minutes;

30.     }

31.

32.     // 获取秒数

33.     public int getSeconds() {

34.         return seconds;

35.     }

36.

37.     // 返回总的秒数

38.     public int getTotalSeconds() {

39.         return hours \* 3600 + minutes \* 60 + seconds;

40.     }

41.

42.     // 将对象转换为字符串形式，格式为"X小时Y分钟Z秒"

43.     @Override

44.     public String toString() {

45.         return String.format("%d:%d:%d", hours, minutes, seconds);

46.     }

47.

48.     // 私有方法，用于标准化时间

49.     private void normalize() {

50.         while (seconds >= 60) {

51.             seconds -= 60;

52.             minutes++;

53.         }

54.         while (minutes >= 60) {

55.             minutes -= 60;

56.             hours++;

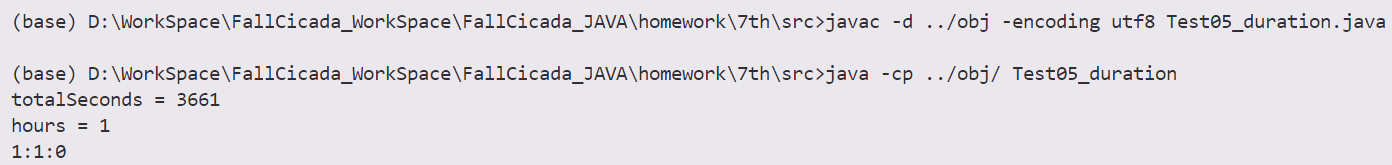
57.         }

58.     }

59. }

60.

运行结果：



# 程序分析

分析程序是否错误，如果错误，说明错误原因。如果正确，写出输出结果：

1. class Car{

2.     public Car(){

3.         System.out.println("car");

4.     }

5. }

6.     //奥迪类

7. class Audi extends Car{

8.     public Audi(){

9.         System.out.println("audi");

10.     }

11. }

12. public class Test06 {

13.     public static void main(String[] args){

14.         Audi a = new Audi();

15.         Car c = new Car();

16.     }

17. }

18.

答：

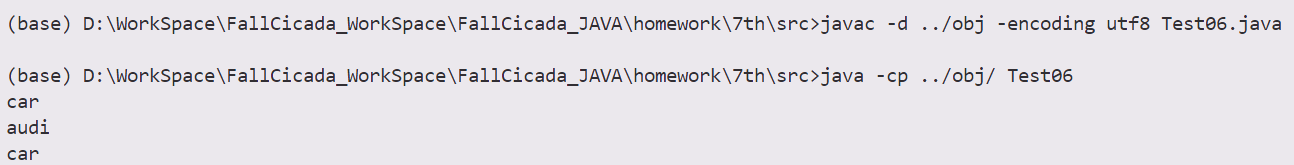
这段代码定义了两个类：Car 和 Audi，其中 Audi 是 Car 的子类。Test06 类中的 main 方法创建了一个 Audi 对象和一个 Car 对象。该代码的运行行为为：

1. 在 Car 类中，构造函数 Car() 打印 "car"。
2. 在 Audi 类中，构造函数 Audi() 打印 "audi" 并且继承了 Car 类，所以当创建 Audi 对象时，会先调用父类 Car 的构造函数再调用自身的构造函数。

故，在 main 方法中：

* 创建 Audi 对象 a 时，首先会调用 Car 的构造函数打印 "car"，然后调用 Audi 的构造函数打印 "audi"。
* 创建 Car 对象 c 时，直接调用 Car 的构造函数打印 "car"。

故，运行行为为：



# 一个类最多可以继承多少个类？

答：

在Java中，一个类只能直接继承一个父类，但可以实现多层继承，也可以实现多个接口。

# 封装和继承有什么区别？

答：

**封装**

封装是指隐藏对象的属性和实现细节，仅对外提供公共的访问方式。它的主要目的是增加抽象性和信息隐藏，使得数据安全，并且易于维护。封装的主要手段包括：

* 使用访问修饰符（如private、protected、public）来限制对类成员的访问。
* 提供get和set方法来访问和修改私有属性以及对私有属性的一系列操作。
* 使用final关键字来防止类被继承或者方法被重写。

通过封装，可以保护内部状态免受外部干扰，并且能够控制对象的访问方式，这样可以更容易地修改内部实现而不影响使用该类的其他部分代码。

**继承**

继承是指允许创建一个新的类去继承现有类的数据结构和方法。它支持层次分类的概念，并且可以实现代码的复用。继承的主要特征包括：

* 子类可以继承父类的方法和字段（但私有的除外）。
* 子类可以覆盖父类的方法以改变其行为（重写）。
* 子类可以添加新的方法和字段，也可以声明父类中不存在的新特性。

继承提供了重用现有类的能力，并允许创建具有共同特性的类的集合。这样可以减少重复代码，使得软件系统更加模块化。

**区别**

* **目的**：封装是为了隐藏实现细节和保护数据；而继承是为了重用代码和捕捉类之间的“is-a”关系。
* **机制**：封装主要依靠访问控制符和方法；继承则依靠extends关键字以及super关键字来引用父类的方法。
* **适用场景**：封装适用于任何需要保护数据完整性的场合；继承通常用于创建具有相似特性的类。

两者结合使用可以构建出灵活、可扩展且健壮的软件系统。在实际开发过程中，合理地利用封装和继承可以使代码更加清晰、易于理解和维护。

# 继承考核

需求分析：

编写程序，实现汽车租赁公司汽车出租方案：

1. 所有车辆（Automobile）都具有品牌（brand）和车牌号（plateNumber）信 息和可以计算租金（getRent）的功能

2. 所有车主要分为卡车（Truck）和巴士（Bus）2种类型

3. 卡车租金方案：



答：

代码：

1. abstract class Automobile {

2.     private String brand;

3.     private String plateNumber;

4.     public Automobile(String brand, String plateNumber) {

5.         this.brand = brand;

6.         this.plateNumber = plateNumber;

7.     }

8.

9.     public abstract int getRent(int days);

10.

11.     public String getBrand() {

12.         return brand;

13.     }

14.

15.     public String getPlateNumber() {

16.         return plateNumber;

17.     }

18. }

19.

20. class Truck extends Automobile {

21.     private String size;

22.

23.     public Truck(String brand, String plateNumber, String size) {

24.         super(brand, plateNumber);

25.         this.size = size;

26.     }

27.

28.     @Override

29.     public int getRent(int days) {

30.         int rent = 0;

31.         switch (size) {

32.             case "小型":

33.                 rent = 300 \* days;

34.                 break;

35.             case "中型":

36.                 rent = 350 \* days;

37.                 break;

38.             case "大型":

39.                 rent = 500 \* days;

40.                 break;

41.             default:

42.                 throw new IllegalArgumentException("Invalid truck size");

43.         }

44.         return rent;

45.     }

46. }

47.

48. class Bus extends Automobile {

49.     private int seats;

50.

51.     public Bus(String brand, String plateNumber, int seats) {

52.         super(brand, plateNumber);

53.         this.seats = seats;

54.     }

55.

56.     @Override

57.     public int getRent(int days) {

58.         int rent = 0;

59.         if (seats <= 16) {

60.             rent = 400 \* days;

61.         } else {

62.             rent = 600 \* days;

63.         }

64.         return rent;

65.     }

66. }

67.

68. public class Test09\_automobile {

69.     public static void main(String[] args) {

70.         Truck t = new Truck("北汽", "苏U12345", "中型");

71.         int rent = t.getRent(3);

72.         System.out.println(t.getBrand() + "\t" +

73.                            t.getPlateNumber() + " 租金为：" + rent);

74.         // 输出 北汽 苏U12345 租金为：1050

75.         Automobile b = new Bus("宇通", "苏A11111", 30);

76.         int busRent = b.getRent(10);

77.         System.out.println(b.getBrand() + "\t" +

78.                            b.getPlateNumber() + " 租金为：" + busRent);

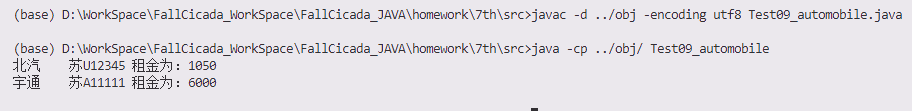
79.         // 输出 宇通 苏A11111 租金为：6000

80.     }

81. }

82.

运行结果：



# 需求分析： 编写代码表示多态概念，并说明什么是多态？

**需求分析：**

编写代码表示多态概念，并说明什么是多态？

**答：**

多态是面向对象编程的一个核心概念，它允许子类对象替换父类对象。简单来说，多态意味着一个类的实例可以像另一个类的对象一样行为，只要这两个类之间存在继承关系，并且子类重写了（或重载了）父类的方法。在Java中，多态通常通过方法重写（overriding）来实现，即子类中覆盖了父类的方法。

**代码：**

1. abstract class Ball {

2.     public abstract void play();

3. }

4. class Basketball extends Ball {

5.     @Override

6.     public void play() {

7.         System.out.println("Playing basketball.");

8.     }

9. }

10. class Football extends Ball {

11.     @Override

12.     public void play() {

13.         System.out.println("Playing football.");

14.     }

15. }

16. public class Test10\_ball {

17.     public static void playBall(Ball ball) {

18.         ball.play();

19.     }

20.     public static void main(String[] args) {

21.         Ball b = new Ball() {

22.             @Override

23.             public void play() {

24.                 System.out.println("Playing an undefined type of ball.");

25.             }

26.         };

27.         Ball b2 = new Basketball();

28.         Ball b3 = new Football();

29.

30.         playBall(b);   // 输出 Playing an undefined type of ball.

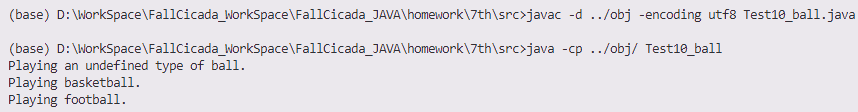
31.         playBall(b2);  // 输出 Playing basketball.

32.         playBall(b3);  // 输出 Playing football.

33.     }

34. }

**运行结果：**

****

# 重载与重写

**需求分析：**

描述方法重载和方法重写的语法要求

**答：**

方法重载

**定义**： 方法重载是指在一个类中可以定义多个同名的方法，但是这些方法必须具有不同的参数列表（参数的数量、类型或顺序不同）。

**语法要求**：

方法名必须相同。

参数列表必须不同（参数的数量、类型或顺序不同）。

返回类型可以相同也可以不同。

访问修饰符可以不同。

**方法重写**

**定义**： 方法重写是指子类重新定义父类中的方法，使得子类能够拥有不同于父类的方法实现。

**语法要求**：

子类和父类中方法的签名必须完全相同（包括方法名、参数列表以及抛出的异常）。

子类方法的访问级别不能比父类方法的访问级别更严格。

如果父类方法是 final 或 private 的，则不能被重写。

如果父类方法是 abstract 的，那么子类必须实现这个方法，除非子类也是 abstract 的。

如果父类方法声明了 throws 某些异常，则子类重写的方法可以声明相同或其子类的异常，但不能声明父类异常。

# 程序分析

**需求分析：**

分析程序是否错误，如果错误，说明错误原因。如果正确，写出输出结果

1. class Super{

2.     protected int test(){

3.     return 1;

4.     }

5. }

6. class Sub extends Super{

7.     public long test(){

8.         return 0L;

9.     }

10. }

11. public class Test12 {

12.     public static void main(String[] args) {

13.         Super s = new Sub();

14.         System.out.println(s.test());

15.     }

16. }

17.

**1. 分析程序是否错误**

上述代码存在类型不匹配的问题。具体来说，在 Sub 类中重写了 test() 方法，并将其返回类型从 int 改为了 long，这与父类 Super 中声明的 test() 方法的返回类型不符。Java 不允许子类方法的返回类型比父类方法的返回类型精度更高，除非它是相同类型或者是父类型的子类型。在这个例子中，long 并不是 int 的子类型，因此会导致编译错误。

**2. 错误原因说明**

**编译错误**：Sub 类中的 test() 方法返回类型为 long，而 Super 类中的 test() 方法返回类型为 int，这种情况下，子类不能改变父类方法的返回类型，否则会引发编译错误。

**3.修改后代码**

1. class Super{

2.     protected int test(){

3.     return 1;

4.     }

5. }

6.

7. class Sub extends Super{

8.

9.     @Override

10.     public int test(){

11.         return 0;

12.     }

13. }

14. public class Test12 {

15.     public static void main(String[] args) {

16.         Super s = new Sub();

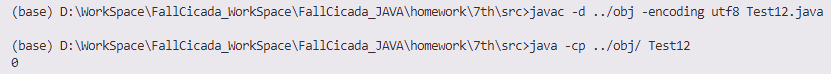
17.         System.out.println(s.test());

18.     }

19. }

20.

**运行结果：**

****

# 程序设计

**需求分析：**

假设要为某个公司编写雇员工资支付程序：

1. 工人（Worker）按每月工作的天数计算工资

2. 销售人员（Salesman）在基本工资基础上每月还有销售提成

3. 经理（Manager） 每月按固定工资支付

4. 临时工（Floater ）按每小时50元支付

5. 所有员工都有共同特性（如姓名，性别，出生日期，员工类别）

**代码：**

1. public class Test13\_employee {

2.     public static void main(String[] args) {

3.         // 21表示工作天数

4.         Employee worker = new Worker("张三", "男", "2001.01.01", 21);

5.         // 3000 表示基本工资 2000表示销售提成

6.         Employee salesman = new Salesman("赵六", "男", "2000.03.07", 3000, 2000);

7.         // 6000 表示基本工资

8.         Employee manager = new Manager("李四", "女", "2003.02.09", 6000);

9.         // 168表示工作小时

10.         Employee floater = new Floater("王五", "女", "2002.10.23", 168);

11.

12.         System.out.println(worker.toString() + " 本月工资为 " + worker.computeSalary() + " 元。");

13.         System.out.println(salesman.toString() + " 本月工资为 " + salesman.computeSalary() + " 元。");

14.         System.out.println(manager.toString() + " 本月工资为 " + manager.computeSalary() + " 元。");

15.         System.out.println(floater.toString() + " 本月工资为 " + floater.computeSalary() + " 元。");

16.     }

17. }

18.

19. /\*

20.  \* 抽象类 Employee 定义了所有员工共有的属性和方法

21.  \*/

22. abstract class Employee {

23.     private String name; // 姓名

24.     private String gender; // 性别

25.     private String birthDate; // 出生日期

26.

27.     public Employee(String name, String gender, String birthDate) {

28.         this.name = name;

29.         this.gender = gender;

30.         this.birthDate = birthDate;

31.     }

32.

33.     /\*\*

34.      \* 计算工资

35.      \* @return 工资金额

36.      \*/

37.     public abstract double computeSalary();

38.

39.     @Override

40.     public String toString() {

41.         return this.name + " " + this.gender + " " + this.birthDate;

42.     }

43. }

44.

45. /\*\*

46.  \* 工人类

47.  \*/

48. class Worker extends Employee {

49.     private int workDays; // 工作天数

50.     private static final double DAILY\_WAGE = 200; // 每天工资

51.

52.     public Worker(String name, String gender, String birthDate, int workDays) {

53.         super(name, gender, birthDate);

54.         this.workDays = workDays;

55.     }

56.

57.     @Override

58.     public double computeSalary() {

59.         return workDays \* DAILY\_WAGE;

60.     }

61. }

62.

63. /\*\*

64.  \* 销售人员类

65.  \*/

66. class Salesman extends Employee {

67.     private double baseSalary; // 基本工资

68.     private double commission; // 销售提成

69.

70.     public Salesman(String name, String gender, String birthDate, double baseSalary, double commission) {

71.         super(name, gender, birthDate);

72.         this.baseSalary = baseSalary;

73.         this.commission = commission;

74.     }

75.

76.     @Override

77.     public double computeSalary() {

78.         return baseSalary + commission;

79.     }

80. }

81.

82. /\*\*

83.  \* 经理类

84.  \*/

85. class Manager extends Employee {

86.     private double baseSalary; // 基本工资

87.

88.     public Manager(String name, String gender, String birthDate, double baseSalary) {

89.         super(name, gender, birthDate);

90.         this.baseSalary = baseSalary;

91.     }

92.

93.     @Override

94.     public double computeSalary() {

95.         return baseSalary;

96.     }

97. }

98.

99. /\*\*

100.  \* 临时工类

101.  \*/

102. class Floater extends Employee {

103.     private int workHours; // 工作小时数

104.     private static final double HOURLY\_WAGE = 50; // 每小时工资

105.

106.     public Floater(String name, String gender, String birthDate, int workHours) {

107.         super(name, gender, birthDate);

108.         this.workHours = workHours;

109.     }

110.

111.     @Override

112.     public double computeSalary() {

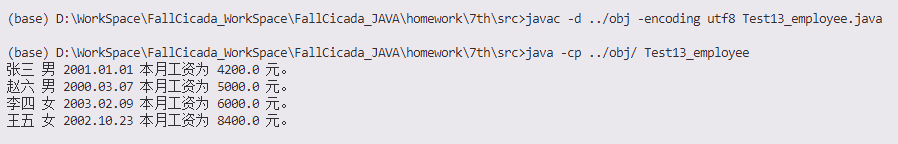
113.         return workHours \* HOURLY\_WAGE;

114.     }

115. }

116.

运行结果：



# 图形类

1. 定义一个圆类 Circle

属性：半径radius

功能：1.计算表面积、2.计算周长

重写功能：1.toString方法会输出对象信息

2.定义圆的子类：圆柱体 Cylinder

属性：高 height

重写功能：1.计算表面积、2.toString方法会输出对象信息

新增功能：计算体积

当Circle类型的引用指向Cylinder类型的对象时，能否调用到它的计算体积的方法？如果能，如何编写代码？

**代码：**

# 程序分析

**需求分析：**

分析程序是否错误，如果错误，说明错误原因。如果正确，写出输出结果

1. class Father {

2.     void show() {

3.         System.out.println("A");

4.     }

5. }

6. class Son extends Father {

7.     void show() {

8.         super.show();

9.         System.out.println("B");

10.     }

11. }

12. public class Test15 {

13.     public static void main(String[] args) {

14.         Father f = new Son();

15.         f.show();

16.     }

17. }

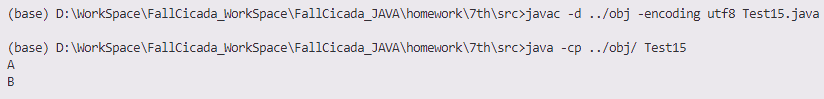
18.

答：

**程序分析**

1. **定义父类 Father**：
   * Father 类有一个 void show() 方法，该方法打印字符 "A"。
2. **定义子类 Son 继承自 Father**：
   * Son 类重写了 show() 方法，在其中首先调用父类的 show() 方法（即 super.show()），然后打印字符 "B"。
3. **主类 Test15**：
   * 在 main 方法中，声明了一个 Father 类型的引用 f 并将其指向 Son 类的一个实例。
   * 接着调用了 f.show() 方法。
4. **运行过程**
   * 当 f.show() 被调用时，虽然 f 是一个 Father 类型的引用，但它实际上指向的是 Son 类的一个实例。因此，Java 解释器会在运行时查找 Son 类中是否有重写的 show() 方法。找到后，它会执行 Son 类中的 show() 方法。
   * Son 类中的 show() 方法首先调用父类的 show() 方法，这会导致字符 "A" 被打印出来。接着，Son 类中的 show() 方法继续执行并打印字符 "B"。

**运行解果：**

****

# 程序分析

**需求分析：**

分析程序是否错误，如果错误，说明错误原因。如果正确，写出输出结果

答：

**程序分析**

1. **定义父类 A**：
   * A 类有一个实例变量 int x = 10 和一个方法 void show()，该方法打印 "A: " 加上 x 的值。
2. **定义子类 B 继承自 A**：
   * B 类也有一个实例变量 int x = 20，并且重写了 show() 方法，该方法打印 "B: " 加上 x 的值。
3. **主类 Test16**：
   * 在 main 方法中，声明了一个 A 类型的引用 a 并将其指向 B 类的一个实例。
   * 接着调用了 a.show() 方法。
   * 打印 a.x 的值。
   * 使用强制类型转换将 a 转换为 B 类型，并打印 ((B)a).x 的值。

**程序执行过程**

**第一步：调用 a.show()**

* a 是一个 A 类型的引用，但实际上指向的是 B 类的一个实例。
* 当调用 a.show() 时，Java 解释器会在运行时查找 B 类中是否有重写的 show() 方法。找到后，它会执行 B 类中的 show() 方法，输出 "B: " 加上 B 类中的 x 值，即 "B: 20"。

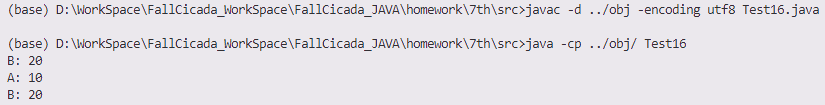
**第二步：打印 a.x**

* a 是一个 A 类型的引用，所以 a.x 访问的是 A 类中的 x 变量。
* 因此，这里输出的将是 A 类中的 x 值，即 "A: 10"。

**第三步：打印 ((B)a).x**

* 这里先将 a 强制转换为 B 类型，然后访问 B 类中的 x 变量。
* 因此，这里输出的将是 B 类中的 x 值，即 "B: 20"。

**运行结果：**



# 程序分析

**需求分析：**

分析程序是否错误，如果错误，说明错误原因。如果正确，写出输出结果

**答：**

**程序分析**

1. 定义父类 Animal：
   * Animal 类有一个方法 speak()，该方法打印 "I am an animal."。
2. 定义子类 Dog 继承自 Animal：
   * Dog 类重写了 speak() 方法，该方法打印 "I am a dog."。
   * Dog 类还有一个方法 wagTail()，该方法打印 "I am wagging my tail."。
3. 主类 Test17：
   * 在 main 方法中，声明了一个 Animal 类型的引用 a 并将其指向 Dog 类的一个实例。
   * 接着调用了 a.speak() 方法。
   * 尝试调用 a.wagTail() 方法。

**程序执行过程**

第一步：调用 a.speak()

* a 是一个 Animal 类型的引用，但实际上指向的是 Dog 类的一个实例。
* 当调用 a.speak() 时，Java 解释器会在运行时查找 Dog 类中是否有重写的 speak() 方法。找到后，它会执行 Dog 类中的 speak() 方法，输出 "I am a dog."。

第二步：尝试调用 a.wagTail()

* a 是一个 Animal 类型的引用，虽然它指向的是 Dog 类的一个实例，但是 wagTail() 方法并没有在 Animal 类中定义。
* 在 Java 中，编译器在编译阶段会检查方法签名是否存在于引用类型中。因为 wagTail() 方法不存在于 Animal 类中，所以在编译阶段就会报错。

**程序报错:**

编译器会抛出一个错误，指出 Animal 类中没有定义 wagTail() 方法。

