

编写：[大半凉](#)；审核、校对：大半凉、[鸠鸠咕咕](#)

## 选择

1. C
2. D
3. D: 静态字段被“同名隐藏”。  
[C++继承：同名隐藏、覆盖，虚函数 C++类继承 同名虚函数-CSDN 博客](#)
4. A
5. A
6. D
7. D
8. A
9. A: 原始调用有编译警告。访问对象进行强制类型转换引发异常 [ClassCastException](#)。  
[ClassCastException\(类转换异常\)-CSDN 博客](#)
10. C: 双向链表适宜的迭代器是 foreach，每次查询均需遍历。
11. B
12. D: Map 是独立于 Collection 的存储形式。

## 判断

1. ✗: 严格来说是正确的。即便程序员不显式声明类的构造方法，编译器也会自动补全。
2. ✓: 比如 HashSet<T> 继承了 Set<T>, Cloneable 和 java.io.Serializable。
3. ✗: 路径覆盖规定每一条路径都必须到达，显然难以实现。
4. ✗: 简单工厂不是经典设计模式，它将对象创建的任务交给了方法，令传入参数匹配以返回对象。工厂模式是，且对象创建的任务位于每一个继承的子类中。
5. ✗: 复用的前提是可重组，类的负担过重意味着各种职责的耦合性强，代码复用性降低，不满足单一职责原则。
6. ✗: 工厂负责创建，显然是创建型模式。
7. ✗: 继承将两个类耦合在一起，自然增加了耦合性。
8. ✗: 同步块是必需的。wait()、notify() 等方法遵循现象级的通信机制，其在执行时必须拥有本方法的监视器锁，否则会抛出 [IllegalMonitorStateException](#) 异常。  
[锁和监视器之间的区别 - Java 并发 - Jiakeqiang - 博客园](#)  
[Java 中“IllegalMonitorStateException”揭秘：破解线程同步与死锁的奥秘 - 云原生实践](#)

9. ✓
10. ✗

## 填空

1. 面向对象特性：**封装、继承、多态**。
2. 策略模式角色：**抽象策略、具体策略、环境角色**。
3. 多继承的方法：**内部类、接口**。
4. I/O 处理方式：**字节流(InputStream/OutputStream)、字符流(Reader/Writer)**。
5. GC 是**隐式分配器**，只有在自定义可释放资源时才需定义终结器。其属于**守护线程**。  
[【JAVA 核心】Java GC 机制详解-CSDN 博客](#) (干货警告)
6. 通配符 **super** 代表父类；反之，**extends** 代表子类。
7. TCP/IP 基于**字节流**。TCP 和 UDP 都是基于字节的传输协议。
8. MVC 设计模式包含**模型、视图和控制器**三层架构。Model 处理数据，View 负责渲染，Controller 用于调度。

## 抽象类和接口异同

	属性	方法	继承	作用
抽象类	没有限制（甚至可以声明构造方法）	非抽象方法必须声明方法主体	子类仅允许继承本类，无法继承其他类	描述类的共性，将一类事物抽象并提供默认实现与共享状态
接口	仅允许声明公开 <code>public</code> 、静态 <code>static</code> 的 <code>final</code> 字段	不允许声明公开方法的主体，除被 <code>default</code> 或 <code>static</code> 关键字修饰；	在继承一个类之外，可以继承多个接口；	约定协议或定义能力
相同点	-	可声明带有主体的私有方法	必须声明接口中声明的方法（或抽象类的抽象方法）主体，除非继承类是抽象类	提高复用性（废话！）

成员：类中的所有被声明的东西。下文中 `ms365`、`test2()` 等等都是成员。

字段：类中没有访问器的（不带逻辑的）变量。下文中 `ms365` 就是字段。

属性：具有公开访问器的字段。下文中没有属性。

方法：类中声明的函数。匿名方法：使用 `lambda` 表达式声明的方法。

方法主体：方法下被大括号包含的区域。

```

3 ①↓ public interface ITester 1个用法 1个实现 新* 1个相关问题
4 {
5     public static final int ms365 = 666; 0个用法
6 ②↓     public abstract void test2(); 0个用法 新*
7
8     private void helper() { } 0个用法 新*
9     public default void test() { } 0个用法 新*
10    public static void test1() { } 0个用法 新*
11 }
12

```

```

3 ③↓ public abstract class Tester 1个用法 1个继承者 新* 1个相关问题
4 {
5     public static int ms365 = 666; 0个用法
6     private static int windows98 = 999; 0个用法
7
8     public int office2024 = 888; 0个用法
9     private int windowsxp = 111; 0个用法
10
11    public abstract void test2(); 0个用法 新*
12    public void test1() { } 0个用法 新*
13
14    private void helper() { } 0个用法 新*
15
16    public Tester() { } 0个用法 新* 1个相关问题
17 }
18

```

解答题见仁见智，默认情况下思路相同即可，无需咬文嚼字。某些答案由 ChatGPT 生成（由下划线标记）并已通过作者检查。

## 关键字 super 与 this 的异同

不同点：

`super` 作为标识父类部分的关键字，可访问父类的公开或受保护的（`protected`）属性、字段和方法（统一称为成员）（当然包含构造方法；构造方法不应称为成员）；

`this` 作为标识当前类实例的关键字，可访问本类的所有成员（静态字段和方法不允许实例调用）当传入参数与字段同名时，可用该关键字进行区分，以防同名隐蔽的发生。

相同点：

只能出现在类的内部，用来访问成员；（不能在静态方法或类外使用）

都用来区分作用域……；

在语法上都是指针型引用……（`super` 其实也是通过当前对象的引用去访问“父类视角”的成员。所以两者在 JVM 层面都属于“访问当前实例的引用”的操作，只是访问入口不同。）

## 集合与数组的区别

集合是抽象容器，可变长；但只能存储引用对象；按照迭代器索引。性能有折扣但灵活。有些支持线程安全。

数组是原始结构，定长；任何类型都可以存储（包含原始类型）；按照下标索引。性能极好但功能少。

## 常见的集合类型

单列集合

`List` 型：`LinkedList`、`ArrayList`、`Vector`（以及对应泛型类或接口）

`Set` 型：`(Linked)HashSet`、`TreeSet`

二列集合

`Map`：`Hashtable`、`HashMap`、`TreeMap`

## 生产者-消费者模式

(1) 关键字 `this` 指代该实例。

(2) 使用 `while` 以保证在被唤醒后重新验证条件。若改成 `if`：

- a) 不能防止伪唤醒。`wait()` 可能无原因地返回，`if` 只检查一次，线程会继续执行并可能在条件仍为真的情况下进入临界区。
- b) 竞态下会破坏容量约束。多个生产者被唤醒时，`if` 不会重新检查，可能同时 `add`，导致实际元素数超过逻辑上的 MAX (`ArrayList` 会自动扩容，程序不会抛异常但逻辑错乱)。
- c) 可能引起消费者和生产者之间的错误同步、数据不一致或长期等待。

(3) 优势：缓冲区用于平衡产生和消耗之间的速度不一致。一方面可以调整双方的并发数，提高任务处理效率。另一方面做到了异步编程：生产者只需关注缓冲区，同时执行 `put()` 的速度快，支持高并发；消费者也只需关注缓冲区，无需监视生产者。

[经典并发同步模式：生产者-消费者设计模式 - 知乎](#)

## 工厂模式设计汽车制造系统

优势：

符合开闭原则，具有稳健的代码扩展性。即若增加新的种类，只需新继承接口（抽象类），而无需更改原始代码；

产品创建被集中在工厂对象中，客户端不必关心其创建的过程；

将种类判断逻辑移向客户端，减少服务端的代码耦合。

劣势：

不适宜大量下游产品的情况，这会导致子类数量疯狂增加，理解和维护成本上升。

另外，如果各种产品相似，工厂方法特性：解耦创建与使用，就显得冗余。

UML 图略。

## 策略模式打车

(1) 策略模式是用来将各种具体的方法抽象出一个行为的方案。

好比商城的促销：原价、打折、买一送一等等，若将在原始代码中新建 if/else 语句体，会显得十分臃肿，并且不利于代码维护，也不符合设计原则。

业务逻辑经常变化、算法需要动态替换、可扩展性强的对战平台或模拟系统、条件判断容易膨胀的地方均适用于策略模式。这些场景均与一种设计类似，即可插拔，或即插即用 (Plug-and-Play)。

(2) 在本例中，打车策略的代码实现很容易，只需关注三种价格有关联即可使用重载轻松完成。

先继承：`public class PcStrategy implements PriceStrategy`

再重载抽象方法：`(public) double Computing(double originalPrice)`

方法主体：`{ return originalPrice / 2; }`

或者你也可以这样写：`{ return ZcStrategy.Computing(originalPrice) / 2; }`

第二种方法令你在下一问：询问多态中多一个材料可以添加。

其他策略类似，在此略。

(3) 表面上，多态发生在重载 Computing 方法中。实际上，当使用策略时，也会发生两次多态：

`PriceStrategy ps = new PcStrategy();` 一次多态：ps 的类型是接口。

`var price = ps.Computing(10);` 第二次多态，调用时，JVM 知晓到底是什么类型（动态绑定）。

策略的替换体现运行时行为的多态，其最大的价值并不在于方法不一样，而是对象可替换。

## 单例模式与反射攻击

(1) 饿汉式实现重点在于饿，它会不计代价地生成。相反，懒汉式实现由于懒，只有在使用时才会生成。

```
3  public class Singleton 2个用法
4  {
5      public static final Singleton Instance = new Singleton(10); 0个用法
6
7      private int Id; 4个用法
8      public int GetId() { return Id; } 0个用法
9      public void SetId(int id) { Id = id; } 0个用法
10
11     private Singleton(int id) { Id = id; } 1个用法
12
13     public void Display() { System.out.println("Singleton ID: " + Id); } 0个用法
14 }
15 |
```

请注意，加入 `final` 关键字，变量即锁定，任何反射都无法更改。

在本例中，若 `Id` 被 `final` 修饰，第三问中无法反射。

(2) 饿汉式简单、稳定、线程安全，其在类加载时就已经创建了唯一实例。但这也会带来困扰：可能会浪费资源、无法延迟初始化（根据运行时参数动态配置）、可能会引发连锁类加载等。

(3) 默认情况下，饿汉式实现无法抵抗反射。准确地说，私有构造方法无法抵抗反射。更准确地说，除被 `final` 修饰的成员均可被反射修改。

第一，设定构造方法：`constructor = objectClass.getDeclaredConstructor();`

第二，设置可见性：`constructor.setAccessible(true);`

第三，创建新实例：`newInstance = constructor.newInstance();`

你可以使用条件判断抵抗反射。反射编译通过，但会反馈运行时错误。

```
3  public class Singleton 2个用法 新*
4  {
5      public static final Singleton Instance = new Singleton(10); 1个用法
6
7      private final int Id; 3个用法
8      public int getId() { return Id; } 0个用法 新*
9
10     private Singleton(int id) 1个用法 新*
11     {
12         if (Instance != null) throw new IllegalStateException("别想拿反射搞我。");
13         Id = id;
14     }
15
16     public void Display() { System.out.println("Singleton ID: " + Id); } 0个用法 新*
17 }
18 |
```

## 附录

一些有意思的 Java 特性：由鸠鸠咕咕提供。

```
public class CharMagic {
    public static void main(String[] args) {
        char c = 'A';
        int i = 10;
        System.out.println(true ? c : i);
        System.out.println(true ? c : 66);
    }
}
```

答案：65; A。

对于 `(true ? c : 66)`, `c` 是 `char`, `66` 是 `int` 字面量。编译器会检查 `66` 是否在 `char` 的范围内。是，因此，整个表达式的类型被确定为 `char`。因为条件为 `true`, 所以表达式的值是 `c` (也就是'A')。因此调用 `println(char)`, 输出 A。

第一个表达式 `(true ? c : i)`, 因为 `i` 是变量, 编译器无法在编译期确定其值, 只能进行标准的数字类型提升, 将 `char` 提升为 `int`, 所以结果类型是 `int`, 输出 65。

```
public class TypePromotionPuzzle {
    public static void main(String[] args) {
        byte b1 = 10;
        byte b2 = 20;
        // byte b3 = b1 + b2; // 这行代码编译不通过
        final byte fb1 = 10;
        final byte fb2 = 20;
        byte fb3 = fb1 + fb2;

        short s = 1;
        // s = s + 1; // 这行代码编译不通过
        s += 1;
        System.out.println(fb3);
        System.out.println(s);
    }
}
```

在 Java 中, 对 `short`, `byte`, `char` 类型的算术运算, 其结果会被自动提升为 `int` 类型。所以 `b1 + b2` 的结果是 `int`, 不能直接赋给 `byte`。

`fb1` 和 `fb2` 都是 `final` 的编译期常量, 编译器会在编译时直接计算出  $10 + 20 = 30$ 。然后编译器会检查 30 是否在 `byte` 的表示范围 (-128 到 127) 内。因为在, 所以编译通过。