Java面试参考指南（一）

2014/07/29 | 分类： [教程](http://www.importnew.com/cat/tutorial) | [0 条评论](http://www.importnew.com/12516.html#respond) | 标签： [面试](http://www.importnew.com/tag/%e9%9d%a2%e8%af%95)

分享到：**62**

本文由 [ImportNew](http://www.importnew.com/) - [darthy](http://www.importnew.com/author/zhufujin) 翻译自 [dzone](http://java.dzone.com/articles/java-interview-reference-guide" \t "_blank)。欢迎加入[翻译小组](http://group.jobbole.com/category/feedback/trans-team/" \t "_blank)。转载请见文末要求。

Java面向对象相关概念

Java是一种基于面向对象概念的编程语言，使用高度抽象化来解决现实世界的问题。    面向对象的方法将现实世界中的对象进行概念化，以便于在应用之间进行重用。例如：椅子、风扇、狗和电脑等。

Java里的类（Class）是一个蓝图、模板，或者称之为原型，它定义了同一类事物的相同属性和行为。实例（Instance）是某个类的一个具体实现，同一个类所有的实例拥有相同的属性。举例来说，你可以定义一个类叫做“房子（House）”，这个类拥有一个属性叫做“房间数（number of room）”，这样你就可以创建一个“房间数”为2的“房子”实例，你还可以创建一个“房间数”为3的“房子”实例，等等等等。

**优点：**

面向对象软件开发的若干优点在于：

* 模块化，维护成本低；
* 更好的代码重用，具备继承性，开发更为敏捷；
* 更好的代码可靠性和灵活性；
* 对现实世界进行建模，易于理解；
* 对象水平的抽象；
* 从一个开发阶段向另一个开发阶段过渡更为简便。

面向对象软件系统（OOPS）的四大主要特征为：

* 封装（Encapsulation）
* 继承（Inheritance）
* 多态（Polymorphism）
* 抽象（Abstraction）

 封装（Encapsulation）

封装机制在对象之间提供了一种隐藏域可见性的协议。Java中使用可见性修饰符private将方法和变量限制在类内部。Java提供的可见性修饰符包括public/ default/ protected/ private，用来在不同层面上隐藏变量、方法和类，但最终目的在于封装那些不需要进行修改的东西。实践表明，每个类应该只存在一种被修改的原因，而封装（Encapsulate）让这种“唯一原因”原则成为现实。

同时，最佳实践表明，封装意味着将会经常改变的东西隐藏起来，以防止对其他类造成破坏。

封装的优点

* 我们可以通过隐藏属性来保护对象的内部状态；
* 能够防止对象之间不恰当的相互作用，进而促进代码模块化；
* 增强可用性；
* 在特定对象之间维护互访协议；
* 封装以促进代码维护；
* 可以独立地进行代码修改

多态（Polymorphism）

多态是指使用相同的接口来应对不同的底层形式（例如数据类型）的能力。这就意味着同一个类可以使用一个共同的接口来实现多种不同的功能，并能通过传递特定的类引用来动态触发这些功能。

一个经典的多态的实例为“形状”类（Shape），以及所有继承Shape的类，如方形（square）、圆形（circle）、多面体（dodecahedron）、不规则多边形（irregular polygon）和长条（splat）等。在这个例子中，每个类中都拥有一个自己的Draw()函数，客户端程序代码可以简简单单地如下所示：

|  |  |
| --- | --- |
| 1 | Shape shape=new Square (); |

执行Shape.area() 可以得到任何形状的正确面积。

多态的美妙之处在于，不同类里的代码不需要知道自己所在的是哪个类，它们的使用方式都是一样的。

面向对象的编程语言在运行时所实现的多态过程叫做动态绑定。

**注：**多态是指根据调用函数的对象来选择更具针对性的方法的特性。当没有抽象类的时候就可以使用多态。

多态的优点

* 可用于创建可重用代码：一旦类被创建，实施和测试，就可以直接进行使用而不考虑具体的代码细节；
* 提供更为泛化和松耦合的代码；
* 编译时间更短，开发更为敏捷；
* 动态绑定；
* 可以使用同一个接口的不同方法来实现不同的功能；
* 可以使用相同的方法签名来代替完全实施。

方法覆盖实现多态：覆盖涉及到两个不同的方法，一个父类的方法，另一个则是子类中的方法，两个方法具有相同的函数名和方法签名。

覆盖可以以不同的方式对不同的对象类型定义相同的操作，例如：

|  |  |
| --- | --- |
| 1  2  3  4 | while(it.hasNext()) {      Shape s = (Shape) it.next();      totalArea += s.area(dim); //多态方法调用，将根据对象类型自动调用正确的方法  } |

方法重载、Ad-hoc多态性和静态多态

重载涉及的是同一个类内具有相同名称，但方法签名不同的多个方法。可以用不同的方法为不同的数据定义相同的操作。我们经常所说的静态多态实际上并不是真正意义上的多态。

方法重载实际上就是指两个方法使用相同的名称，但参数不同。这与继承和多态完全没有关系。重载方法不是覆盖方法。[Head First Java深入浅出]

Java中基于泛型的参数多态性

当进行类声明时，一个属性域名称可以与多种不同的数据类型相关联，一个方法也可以与不同的参数类型和返回类型相关联，Java支持使用泛型的参数多态性。例如，一个list对象可以通过泛型来接收它所包含的数据类型：

|  |  |
| --- | --- |
| 1 | List<String> list = new ArrayList<String>(); |

为什么在Java里我们不能覆盖静态（static）方法？

覆盖依赖于具体的类实例。多态的关键之处在于你可以继承一个类，而该类的子类所定义的实例对象又可以对父类中定义的方法进行了覆盖。而静态方法是不依赖与具体实例的，因此覆盖这一概念不适用于静态方法。

在Java设计早起有两点考虑直接导致了这一现象。第一是对性能方面的考虑：之前人们对Smalltalk语言（一种面向对象编程语言）运行太慢（垃圾回收和多态调用所致）的批评不绝于耳，Java的设计者决定回避这一弊端。第二是考虑到Java的预期受众主要是C++开发人员，而使静态方法能直接被调用刚好能迎合C++编程人员的开发习惯，同时由于不用上溯类层级结构来查找要调用的方法，而是直接调用指定类中的特定方法，这一设计使得代码运行非常快速。

继承（Inheritance）

继承是指派生类中包含了基类中的所有的行为（即方法）和状态（即变量），并能通过该派生类进行访问。继承的关键好处在于它提供了代码重用和避免重复的一遍机制。

继承类通过重用父类的方法并添加一些新的功能来扩展应用程序的功能。这回导致紧耦合，如果你想对父类进行修改，你必须清楚其所有子类的具体细节以防止阻断。

这是一种软件复用性，新类（子类）继承已有的父类，重用父类的特征并能添加一些新的功能。

因此，举例来说，如果你有一个Parent类和一个扩展（使用关键字extends）Parent类的Child类，那么Child类继承了Parent类所有特征。

优点

* 促进重用性；
* 建立逻辑“is a”关系，如：Dog is an animal.
* 使代码模块化；
* 避免冲突。

缺点

* 紧耦合：子类的实现依赖于父类，导致紧耦合。

抽象（Abstraction）

抽象意味着只需要开发类的接口和功能声明，而不需要实现具体的实施细节。抽象类呈现的是接口，而不需要具体实现，将对象的实施与行为或实现分离开来，进而通过隐藏无关的细节来减少复杂度。

优点

* 通过使用抽象，我们可以将不同类别的东西分离开来；
* 经常需要修改的属性和方法可以被分离出来形成一个单独的类别，而那些主要留下的部分就不需要进行修改了，进而增强面向对象的分析与设计（OOAD）原则，即“代码应该易于扩展，而不应该经常修改”；
* 简化领域模型的表征。

抽象和封装的区别：

封装作为一种策略，被用作广义抽象的一部分。封装是与对象状态相关的——对象将自己的状态封装起来并对外界不可见，类外部的用户只能通过该类的方法来与其进行交互，但不能直接改变其状态。因此，类可以将与状态相关的实施细节通过抽象隔离开来。

抽象是一个更泛化的概念，可以通过子类来实现具体的功能。例如：在Java标准库中，List是“一串事物”的抽象，ArrayList和LinkedList是List的两个具体的类型，作用于抽象List的代码同样抽象地不指明具体所使用的List类型。

如果没有通过封装隐藏底层状态，也就不可能进行抽象处理。也就是说，如果一个类的内部状态全部都是公开的，内部功能无法被更改，该类也就无法进行抽象。

什么是抽象类和抽象方法？

在程序设计过程中，你希望基类只是其派生类的一个接口，也就是说，你不希望任何人能实例化该基类。你只是想隐式（可以实现多态性）地提出它，以便可以使用它的接口。那么你可以使用abstract关键字来定义一个抽象类。

为该抽象类设定一些限制，所有使用该抽象类的子类都必须实现其中的抽象方法，并提供多态性。

抽象类中可以既包括抽象方法和具体方法，如果一个方法是抽象方法，其所在的类必须被声明成抽象类。反之不然，如果一个类是抽象类，其中不一定包括抽象方法。

如果一个方法只提供了方法签名，但没有被具体实现，则这个方法是一个抽象方法，该方法的具体实现是在扩展该抽象类的子类中进行的。

抽象方法不能被实例化，其他类只能扩展它。

什么时候使用抽象类？

抽象类定义了一些默认的行为，而将具体的功能留给子类来实现。例如：List是一个接口，而抽象类AbstractList提供了List的默认方法，这些默认方法可以被子类ArrayList继承或重新定义。

什么是接口？

Interface关键字使得接口相比于抽象类更进了一步，接口中不能定义实现的方法。实现（使用关键字implements）接口的非抽象类必须实现该接口的所有方法。接口是面向对象（OO）中的一个非常有用和常用的概念，它将接口和具体实现分离开来，并保证数据安全性：

接口是抽象类的延伸，java了保证数据安全是不能多重继承的，也就是说继承只能存在一个父类，但是接口不同，一个类可以同时实现多个接口，不管这些接口之间有没有关系，所以接口弥补了抽象类不能多重继承的缺陷，但是推荐继承和接口共同使用，因为这样既可以保证数据安全性又可以实现多重继承。

使用接口的优点

1. 多重继承；
2. 能对操作进行松耦合的抽象，可以分离实现任何功能， JDBC, JPA, JTA等等；
3. 开发接口而不需要具体实现；
4. 使用动态绑定的多态——揭示一个对象的编程接口，而无需展现其具体的实现细节；
5. 抽象层：分离问题。

接口和抽象类之间的区别

* 接口是一种协议，要实现接口的类需要根据接口中定义地来实现接口，它只是一个提供了方法声明的空壳；
* 抽象类定义了一些通用方法，其子类可以定义新的具体或特殊的方法；
* 抽象类中的方法和参数可以被定义成任何可见性的，而接口中的所有方法必须由public可见性修饰符定义；
* 继承一个抽象类，子类需要实现其中的抽象方法，然而接口可以扩展另一个接口而无需实现其中的方法；
* 子类只能继承单个抽象类，而一个接口或类可以实现多个接口；
* 继承抽象类的子类可以以相同或更低的可见性实现父类中的抽象方法，而实现接口的类只能以与原抽象方法相同的可见性实现接口中的方法；
* 接口没有构造函数，抽象类有；
* 接口中的变量都是final型的，而抽象类中可以包含非final型变量；
  + 接口中的成员默认是public类型的，但抽象类中的成员的访问类型可以是public，protected和默认类型。

合成

代码的重用性可以通过集成和合成来实现，但是用合成实现代码重用比继承居右更好的封装性，因为对后端代码的修改无需任何对仅依赖于前端类的代码的破坏。

合成石实现类之间“has-a”关系的设计技术，我们可以使用Java的继承或对象合成来实现代码重用。

合成表示的是对象之间的关系，以椅子chair为例，一把椅子chair有一个座位seat，一个靠背back，几条腿legs，词组“has a”表示一把椅子所包含，或所使用的其他实体，这种“has a”关系就是合成的基础。

优点:

* 控制可见性
* 运行时可以更换实现方法
* 松耦合，接口不依赖与具体实现。

合成与继承之间的区别？

|  |  |  |
| --- | --- | --- |
| No. | 合成Composition (has a) | 继承Inheritance (is a) |
| 1 | 主张多态和代码重用 | 主张多态和代码重用 |
| 2 | 运行时已经获得对象 | 编译时动态获得对象 |
| 3 | 运行时可以替换实施 | 编译时可以更换实施 |
| 4 | 松耦合，子类的存在并不依赖与父类（特别是接口驱动的情形下） | 紧耦合，子类严格依赖于父类 |
| 5 | 当House中有一个Bathroom时可以使用，不能说House是一种Bathroom | 继承是单向的，例如：House是一个Building，但Building不是一个House |

**注：**不要仅仅只为了代码重用而使用继承，如果类之间不存在“is a”关系，建议采用合成来实现代码重用。

对象关系中合成与聚合的区别：

聚合：聚合是一个类属于一个集合的关系。描述的是一种“部分与整体”的关系，“整体”不存在的情况下，“部分”是可以存在的，这是一种弱关系类型，没有循环依赖性。例如，订单和产品的关系。

合成：合成也是一个类隶属于一个集合的关系。描述的是一种只有“整体”存在，“部分”才能存在的“部分与整体”的关系。在这种关系下，如果“整体”被删除，则“部分”也就不复存在，体现的是一种强关系类型。例如，多边形和组成多边形的点，订单和订单明细。

Java面试参考指南（二）

2014/07/30 | 分类： [教程](http://www.importnew.com/cat/tutorial) | [1 条评论](http://www.importnew.com/12522.html#comments) | 标签： [面试](http://www.importnew.com/tag/%e9%9d%a2%e8%af%95)

分享到：**38**

本文由 [ImportNew](http://www.importnew.com/) - [jessenpan](http://www.importnew.com/author/jessenpan) 翻译自 [dzone](http://java.dzone.com/articles/java-interview-reference-guide-0" \t "_blank)。欢迎加入[翻译小组](http://group.jobbole.com/category/feedback/trans-team/" \t "_blank)。转载请见文末要求。

访问修饰符

对于基本的OOPS（面向对象）概念，请看[Java面试参考指南的第一部分](http://www.importnew.com/12516.html)。访问修饰符规定了一个类如何访问另一个类及它的成员（包括方法和变量）。

Java中有下列访问修饰符：

* private：私有变量和方法（非外部类）只能被声明它们的类的实例所使用。
* default：类中的数据、方法和它本身能够被声明为默认default。类中所有default成员都可以被本包中的其它类所访问。
* protected：相比default有更高的访问权限。只有成员变量和方法能够被声明为protected。父类的protected属性能被它的所有子类所共享。即使子类和父类不在同一包中，这种访问也是支持的。
* public：公共的类、变量和方法都可以毫无限制的被其它Java程序所使用。

**注意：**

* 对于非内部类就只存在public访问修饰符，不存在被protected或private修饰的顶层类。
* 一个特性至少拥有一个访问修饰符。如果一个特性没有访问修饰符，那么它的修饰符就是default模式。
* 方法重载的隐私：重载后的方法修饰符不能低于被重载的方法，比如父类是protected，子类中重载方法不能置为private。

下表描述了对各个成员修饰符所运行的访问权限：

|  |  |  |
| --- | --- | --- |
| 修饰符 | 被修饰者 | 描述 |
| public | 外部（Outer）类，接口，构造器，内部（Inner）类，方法和变量 | 可在包外部访问 |
| protected | 构造器，内部类，方法和变量 | 包内访问或包外的任何子类访问 |
| private | 构造器，内部类，方法和变量 | 只能够在它们被声明的内部访问 |
| 无描述符（default） | 外部类，内部类，接口，构造器，方法和变量 | 只能够在它们所在包中被访问 |

除了上述基本的访问符之外，下表描述了Java中的其它描述符。它们能够改变类和它的成员（如：方法、变量和内部类）的行为。

 Java修饰符

|  |  |  |  |
| --- | --- | --- | --- |
| 修饰符 | 内部类/类 | 方法 | 变量 |
| static | 一个静态内部类是另一个普通类的成员，不是该类的对象成员。 | 一个静态方法可以直接被类访问，并且被该类的对象所共享。 | 静态变量是类级变量且在jvm中只能够出现一次。 |
| abstract | 抽象类不能够被实例化，它只能被其它类所继承。 | 抽象方法只能被声明在抽象类中，它的实现代码只能被包含在子类中。 | 不适用 |
| synchronized | 不适用 | 它在类上加了一个监视器，使得访问该方法只能通过获得该方法所属对象的锁来进行。 |  |
| transient |  |  | 变量不能够被序列化。 |
| final | 类不能被继承 | 方法不能被重载。 | 变量值变成常量。 |

final

final修饰符只能够作用在类、方法和变量上。它的作用是所有被final修饰的内容不能被改变。如：

* final类不能被继承。
* final变量一旦被赋值就不能被改变。
* final方法不能被重载。

**注意：**如果一个final变量是对象的引用，这意味着该引用的值是一定不会改变的，但是对象的成员值可以改变。

static

static修饰符可以被应用在变量、方法、代码段和内部类中。如果需要Java实例变量被存储在单独的内存中，或需要变量对一个单独类的所有对象所通用，那么静态修饰符需要被应用在该变量的声明中。

* **静态方法：**Java的静态方法对类是通用的并且并不和Java实例相关联。尽管静态方法能够自由的访问类的静态数据和方法，但是它们并不能使用类的非静态功能。静态方法不能够被重载为非静态的。对于Java静态方法不能够使用this关键字。
* **静态块：**静态代码块在类加载时只执行一次。
* **静态变量：**Java的实例变量使用单独的空间进行存储。如果需要变量对一个类的所有对象都是通用的，那么这个变量就需要被声明为静态的。类的所有实例对象都可以修改该类的静态变量。此外，使用对象实例修改静态成员并不是唯一手段，也可以直接使用java类来修改。静态变量也可以被Java对象的方法所访问。当一个常量的值在编译器就已经知晓了，它就需要使用static关键字来声明为final类型。
* **静态内部类：**只有内部类能够使用static修饰符来声明，在Java中它被称为静态嵌套类。

外部类和内部类（或者嵌套类）

在Java中，并不是所有的类都必须单独定义。你可以把一个类定义在另一个类的内部中。这个在内部定义的类就叫做内部类。包围类则被称为外部类。因此，当你定义了一个内部类，它和包围类的其它变量、方法和构造器成员一样成为该类的成员。

当你访问外部类的私有数据成员时，JDK会在外部类中创建包级访问权限的（package-acess）成员函数以便内部类来访问外部类的私有成员。这种结果会导致一个安全漏洞。通常情况下，我们应当尽量避免使用内部类。

只有当内部类仅仅只和外部类的上下文相关或者内部类需要被设置为私有的且只能被外部类访问的这些情况下，我们才使用内部类。内部类主要被用来实现类似Iterators、Comparators这些辅助类。它们被使用在外部类的上下文中。

静态嵌套类和非静态嵌套类的区别

非静态嵌套类（即内部类）对它嵌套的外部类的成员拥有全部的访问权限。而静态嵌套类并不包含对外部类的实例引用，这就使得静态嵌套类不能够调用外部类实例的非静态方法或者访问外部类的非静态字段。

在声明成员字段和方法上，非静态嵌套类不能够定义静态字段和方法。但是，静态内部类则可以定义静态字段和方法也可以定义非静态字段和方法。

非静态内部类的实例是通过使用外部类的对象引用来创建的，也就在说在内部类中已经定义了外部类实例。但是静态嵌套类实例的创建过程中并不涉及到外部类的引用，这就是说它并不拥有包围类的实例。

**示例：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class OuterClass {        class InnerClass {          // static int x; not allowed here      }        static class StaticInnerClass {          static int x; // allowed here      }  }    class Test {      public static void main(String... str) {          OuterClass oc = new OuterClass();          OuterClass.InnerClass obj1 = oc.new InnerClass();// need of inclosing                                                              // instance          OuterClass.StaticInnerClass obj2 = new OuterClass.SIC();          // no need of reference of object of outer class      }  } |

native

native修饰符只能够修饰方法。native修饰符意味着方法的实现体是在JVM之外。

**注意：**抽象方法的实现体是在子类中，而native方法的实现体则完全不在Java的虚拟内存中，而是在其它程序库中。

transient

transient修饰符只能够应用在变量中，transient变量不做为它所属对象的持久状态的一部分进行存储。它并不需要被序列化，主要用作安全键（security key）或者连接（connection）等。

synchronized

synchronized修饰符在多线程程序中对临界区代码的访问进行控制。synchronized关键字是保持代码线程安全的工具之一。

对于某一对象的一个同步方法进行交叉访问是不可能的。当一个线程在执行某一对象的同步方法时，所有调用该同步方法的其它线程都将被阻塞直到第一个线程完成对该方法的调用。

其次，当一个同步方法执行结束时，对相同对象的后续同步方法调用，它会自动建立之前发生（happens-before）的顺序关系。这保证了该对象的状态更改对所有的线程都可见。

当你将一段代码块标记为同步化时，你需要使用对象作为该同步块的参数。当一个执行线程到达该代码块时，它首先需要等待在该对象的同步块上已经没有其它的执行线程。然而，一个线程可以进入不同对象锁定的同步方法块。

但对同一对象的非同步方法可以直接方法而无需锁检测。

如果你同步化的是静态方法，那么你同步时获取的是该方法类的锁而不是实例的。这就意味着当你同步的是静态方法时，整个类都将被阻塞。这样其它的静态同步方法也将被阻塞。

当一个线程进入同步化实例的方法时，其它线程则不能够再进入该实例的其它同步化方法。

当一个线程进入同步化的静态方法时，其它线程则不能够再进入该类的其它同步化静态方法。

**注意：**同步化的静态方法和同步化的非静态方法之间是没有多少联系的。例如：如果静态和非静态同步方法能够并发的执行，就需要将你的非静态方法显式的声明为在它自己的类上进行同步（如：同步MyClass.class{…}）。

volatile

只有变量能够被定义为volatile。这些变量可能会被异步修改，因此编译器需要对他们额外的关注。Volatile修饰符保证任何读取该字段的线程都能够获取它的最近修改值。

**volatile使用：**使用volatile的一种通用情况是将它用作boolean变量并作为判断线程终止的标志。

**volatile和synchronize之间的区别：**

因此，volatile关键字只用来在线程内存和主内存之间同步单个变量值，synchronized关键字用来同步线程内存和主内存之间的所有变量值以及如何锁定和释放一个监视器。清楚的是，synchronized比volatile有着更大的开销。

volatile变量不允许出现和当前主存中的数据值不同的的本地副本。

更准确地说，被声明为volatile的变量必须保证它的数据值在所有线程的中是同步的。也就是当你在任一线程中访问或者更新一个变量时，所有其它线程能够立即访问到相同的值。

abstract

abstract修饰符适用于类和方法。抽象类不能够被实例化，它必须被继承才能够访问。abstract不能够应用到类的成员变量。

**示例：**Animal类可以有travel方法同时它的各子类（蛇，狗，鸟）也有它们自己的travel方法。因此就不能够在超类中提供travel方法，而是需要将travel方法在超类中声明为抽象的。

**注意：**

* 如果类包含一个或多个抽象方法，编译器将强制该类必须声明为抽象的。
* abstract类和final类是相对的。final类不能够被继承，而抽象类必须被继承。

设计上，你希望基类对继承类仅仅只作为接口。这意味着你不希望任何人实例化该基类。你只想进行向上类型转换（隐式转换，它能够提供多态性），这样就只能使用该基类的接口了。这是通过使用abstract关键字来修饰该类实现的。

提供限制来不实例化抽象类，这样就需要任何使用该抽象类的人来实现它。并提供多态性。

**什么时候使用抽象类？**

抽象类让你可以定义一些默认行为并促使子类提供任意特殊化行为。

例如：Spring的依赖注入就使得代码实现了集合框架中的接口原则和抽象实现。

原文链接： [dzone](http://java.dzone.com/articles/java-interview-reference-guide-0) 翻译： [ImportNew.com](http://www.importnew.com/)- [jessenpan](http://www.importnew.com/author/jessenpan)  
译文链接： <http://www.importnew.com/12522.html>