# 第 1 章 对象导论

1. 将类的一个对象置于某个新的类中，称为“创建一个成员对象”。
2. 使用现有的类合成新的类，称为“组合”（composition），如果组合是动态发生的，通常被称为“聚合”（aggregation）。组合经常被视为“has-a”（拥有）关系，如“汽车拥有引擎”。
3. **在 Java 中，动态绑定是默认行为，不需要添加额外的关键字来实现多态。(C++中通过继承和虚函数[virtual 关键字]实现多态)**
4. 在 Java 中，所有的类最终都继承自单一的基类，这个终极基类就是 Object。
5. 向上转型（派生类赋值给基类）是安全的，如 Circle 是一种 Shape 类型；但是不知道某个 Object 是 Circle 还是 Shape，所以除非确切知道所要处理的对象的类型，否则向下转型（基类赋值给派生类）几乎是不安全的。

# 第 2 章 一切都是对象

1. 在 Java 里，一切都被视为对象。尽管一切都**看作**对象，但操纵的标识符实际上是对象的一个“引用”。引用可独立存在，即你拥有一个引用，并不一定需要有一个对象与它关联。例：如果想操纵一个词或句子，则可以创建一个 String 引用：String s;但这里所创建的**只是**引用，并不是对象。
2. 对象的存储位置：
（1）**寄存器**。这是**最快**的存储区。
（2）**栈**。位于通用 RAM 中，这是一种**快速有效**的分配存储方法，**仅次于**寄存器。
（3）**堆**。一种通用的内存池（也位于 RAM 区），用于存放**所有**的 Java 对象。
（栈比堆要高效，考虑：栈指针保存在寄存器中，堆分配空间时要做一系列查找分配工作。）
（4）**常量存储**。常量值通常直接存放在程序代码的内部，这样做是安全的，因为它们永远不会被改变。
（5）**非 RAM 存储**。如磁盘等。
3. 基本类型不用 new 来创建变量，而是创建一个并非是引用的“自动”变量。这个变量直接存储“值”，并置于栈中，因此更加高效。如 boolean、char(16-bit)、byte(8bits)、short(16bits)、int(32bits)、long(64bits)、float(32bits)、double(64bits)、void。
4. 在 Java 中，每种基本类型所占存储空间的大小并不像其它大多数语言那样随机器硬件架构的变化而变化，这是其更具可移植性的原因之一。
5. Java **只**支持有符号数。
6. Java 确保数组会被初始化，而且不能在它的范围之外被访问。这种范围检查，是以每个数组上少量的内存开销及运行时的下标检查为代价的。
7. 尽管以下代码在 C 和 C++中是合法的，但在 Java 中却不能这样书写：

```
{
    int x = 12;
    {
        int x = 96; //illegal
    }
```

}
编译器将会报告变量 x 已经定义过。
8. Java 对象不具备和基本类型一样的生命周期。当用 new 创建一个 Java 对象时，它可以存活于作用域之外。如下代码：

```
{
    String s = new String("a string");
}// end of scope
```

引用 s 在作用域终点就消失了。然而，s 指向的 String 对象仍继续占据内存空间。
9. 若**类的某个成员**是基本数据类型，即使没有进行初始化，Java 也会确保它获得一个默认值。(char 类型被初始化为一个空格)注意：当变量作为**类的成员**使用时，Java 才确保给定其默认值，以确保那些是基本类型的成员变量得到初始化（C++没有此功能），防止产生程序错误。然而确保初始化的方法并不适用于“局部”变量（即并非某个类的字段）。如果在某个方法定义中有 int x; 那么变量 x 得到的可能是任意值（与 C 和 C++中一样），而不会被自动初始化为零。实际上，如果在某个方法中像这样只定义 int x, Java 在编译时便会报错，提示该变量未被初始化。
10. 方法的参数列表中必须指定每个所传递对象的类型及名字。像 Java 中任何传递对象的场合一样，这里传递的实际上也是引用。对于前面所提到的特殊数据类型 boolean 等来说是一个例外。通常，**尽管传递的是对象，而实际上传递的是对象的引用**。
11. 一个 static 字段对每个类来说都只有一份存储空间，而非 static 字段则是对每个对象有一个存储空间。
补充：通常，static 方法不能直接调用非 static 成员变量和方法。对于一般的非 static 成员变量和方法来说，需要有一个对象的实例才能调用，所以要先生成对象的实例，它们才会实际的分配内存空间。而对于 static 的对象和方法，在程序载入时便已经分配了内存空间，它只和特定的类相关联，无需实例化。如果要在 static 方法中调用非 static 成员变量和方法，须先实例化，即：在 static 方法中调用实例对象的非 static 成员变量和方法。
12. java.lang 是默认导入到每个 Java 文件中的，所以它的所有类都可以被直接使用。（编写的代码中，java.lang 不会显式出现）。

# 第 3 章 操作符

1. 对象“赋值”：对一个对象进行操作时，我们真正操作的是对对象的引用。所以倘若“将一个对象赋值给另一个对象”，实际是将“引用”从一个地方复制到另一个地方。（引用与对象之间存在关联，但这种关联可以被改变。）
2. ==和!=比较的是对象的引用。equals()方法的默认行为是比较引用，如果定义类的对象中对 equals()方法进行重写，则可以实现比较对象的实际内容是否相等的效果。（int 类型与 Integer 进行“==”比较时，Integer 会自动拆箱成 int 类型再进行比较。）
3. “与”(&&)、“或”(||)、“非”(!)操作只可应用于布尔值。与在 C 和 C++中不同的是：不可将一个非布尔值当作布尔值在逻辑表达式中使用。注意，如果在应该使用 String 值的地方使用了布尔值，布尔值会自动转换成适当的文本形式。
4. 如果对 char、byte 或者 short 类型的数值进行移位处理，那么在移位进行之前，它们会被转换为 int 类型，并且得到的结果也是一个 int 类型的值。
5. 直接将 float 或 double 转型为整数值时，总是对该数字执行截尾。如果想要得到四舍五入

的结果，需要使用 java.lang.Math 中的 round()方法。
6. 只要类型比 int 小（即 char、byte 或者 short），那么在运算前，这些值会自动转换成 int。通常，表达式中出现的最大的数据类型决定了表达式最终结果的数据类型。float*double=double,int*long=long.
7. Java 没有 sizeof，因为所有数据类型在所有机器中的大小都是相同的。

# 第 4 章 控制执行流程

1. Java 编译器生成它自己的“汇编代码”，但是这个代码是运行在 Java 虚拟机上的，而不是直接运行在 CPU 硬件上。
2. switch 语句要求使用一个选择因子，并且必须是 int 或 char 那样的整数值。假若将一个字符串或者浮点数作为选择因子使用，那么它们在 switch 语句里是不会工作的。

# 第 5 章 初始化与清理

1. 每个**重载**的方法都必须有**独一无二的参数类型列表**。（参数顺序的不同也足以区分两个方法，但不建议这样做，会使代码难以维护。）
2. 方法重载时，如果可以重载的方法间只是参数类型不同，传入的数据类型（实际参数类型）**小于**方法中声明的形式参数类型，实际数据类型就会被提升至该方法所接受的类型。char 型略有不同，如果无法找到恰好接受 char 参数的方法，就会把 char 直接提升至 int 型。（P81）如果传入的实际参数**较大**，就得通过类型转换来执行窄化转换。如果不这样做，编译器就会报错。即先类型转换，后传入参数。
3. 要是你没有提供任何构造器，编译器会认为“你需要一个构造器，让我给你制造一个吧”；但假如你已写了一个构造器，编译器就会认为“啊，你已写了一个构造器，所以你知道你在做什么；你是刻意省略了默认构造器。”即编译器此时是不会为你制造一个默认构造器的。
4. this 关键字只能在方法内部使用，表示对“调用方法的那个对象”的引用。
5. 可能为一个类写了多个构造器，有时可能想**在一个构造器中调用另一个构造器**，以避免重复代码，可用 **this** 关键字做到。此时，在构造器中，如果为 this 添加了参数列表，将产生对符合此参数列表的某个构造器的明确调用。如：

```
public class A{
    A(String s){
    }
    A(String s,int i){
    this(s);//  相当于 A(s);
    }
}
```

然而，需要遵守如下规则：
（1）尽管可以用 this 调用一个构造器，但却不能在一个构造器中调用两个构造器，即在一个构造器中最多只能调用一个构造器；

（2）必须将构造器调用置于最起始处，否则编译器会报错；
（3）除构造器之外，编译器禁止在其他任何方法中调用构造器。
6. static 方法就是没有 this 的方法。在 static 方法的内部不能调用非静态方法，当然，这不是完全不可能：如果你传递一个对象的引用到静态方法里，然后通过这个引用，你就可以调用非静态方法和访问非静态数据成员了。
7. Java 中垃圾回收遵守的原则：
（1）对象可能不被垃圾回收；
（2）垃圾回收并不等于“析构”；
（3）垃圾回收只与内存有关。
如果 JVM 并未面临内存耗尽的情形，它是不会浪费时间去执行垃圾回收以恢复内存的。
8. 在 C++中可以创建一个局部对象（也就是在栈上创建，这在 Java 中行不通），在 Java 中不允许创建局部对象，必须使用 new 创建对象。（**Java 对象都在堆上创建，不能在栈上创建**）
9. 在类的内部，变量定义的先后顺序决定了初始化的顺序。即使变量定义散布于方法定义之间，它们仍旧会在任何方法（包括构造器）被调用之前得到初始化。
10. 静态对象的初始化先于非静态对象，静态对象只被初始化一次。
11. 在声明数组时，编译器不允许指定数组的大小。即这样：int a[10];编译器会报错。数组元素中的基本数据类型值会自动被初始化。
12. switch 与 enum 是绝佳的组合。

# 第 6 章 访问权限控制

1. 访问权限控制的等级，从最大权限到最小权限依次为：**public、protected、包访问权限（没有关键字）和 private**。
2. 如果不提供任何访问权限修饰词，则意味着它是“**包访问权限**”，即当前的包中的所有其他类对那个成员都有访问权限，但对于这个包之外的所有类，这个成员却是 private。
3. 使用关键字 **public**，就意味着 public 之后紧跟着的成员声明自己对每个人都是可用的。
4. 关键字 **private** 的意思是，除了**包含该成员的类**之外，其他任何类都无法访问这个成员。（注意：**C++**中声明为 private，只能是类本身，以及友元函数和友元类访问；**类的对象实例是不能访问 private 类成员的**。而 Java 中 private 属性的权限扩大到了包含该成员的整个类范围。）
5. **protected（继承访问权限）**：基类的创建者会希望有某个特定成员，把**对它的访问权限赋予派生类而不是所有类**。这就需要 protected 来完成这一工作。protected 也提供包访问权限，即相同包内的其他类可以访问 protected 元素。（毕竟 protected 大于包访问权限）
6. 类既不可以是 private 的，也不可以是 protected 的（事实上，一个内部类可以是 private 或 protected 的但那是特例）。所以对于类的访问权限，仅有两个选择：包访问权限或 public。
7. 相同目录下的所有不具有明确 package 声明的文件，都被视作是该目录下默认包的一部分。

# 第 7 章 复用类

1. 每一个非基本类型的对象都有一个 toString()方法，而且当编译器需要一个 String 而你却只有一个对象时，该方法便会被调用。
2. 当创建一个类时，总是在继承，因此，除非已明确指出要从其他类中继承，否则就是在隐式地从 Java 的标准根类 Object 进行继承。
3. 当创建一个导出类的**对象**时，对象所包含的基类的子对象被包装在导出类对象的内部。Java 会自动在导出类的构造器中插入对基类构造器的调用。
4. @Override 注解表明覆盖某个方法，可以防止在不想重载时而意外地进行了重载。
5. “is-a”的关系是用继承来表达的，而”has-a”的关系则是用组合来表达的。
6. 新类是现有类的一种类型。由导出类转型成基类，在向上转型的过程中，类接口中唯一可能发生的事情是丢失方法，而不是获取它们。
7. 一个既是 static 又是 final 的 field（个人理解：字段，变量）只占据一段不能改变的存储空间。
8. 当对**对象引用**运用 final 时，引用恒定不变。一旦引用被初始化指向一个对象，就无法再把它改为指向另一个对象。然而，对象其自身却是可以被修改的。
9. 类中所有的 private 方法都隐式地指定为是 final 的。由于无法取用 private 方法，所以也就无法覆盖它。可以对 private 方法添加 final 修饰词，但并不能为该方法增加任何额外的意义。
10. 当将某个类的整体定义为 final 时，表明你不打算继承该类，而且也不允许别人这样做。换句话说，出于某种考虑，你对该类的设计永不需要做任何变动，或者出于安全考虑，你不希望它有子类。注意：**final 类的 field** 可以根据个人意愿**选择为是或不是 final**，但由于 final 类禁止继承，所以 **final 类中所有的方法都隐式指定为 final** 的，因为无法覆盖它们。在 final 类中可以为方法添加 final 修饰词，但无任何额外意义。
11. Java 中允许生成“空白 final”，所谓空白 final 是指被声明为 final 但又未给定初值的 field。无论什么情况，编译器都确保**空白 final 在使用前必须被初始化**。它使得一个类中的 final field 可以做到根据对象而有所不同，但又保持其恒定不变的特性。（通常在类中定义 final field，在不同的构造器中给予不同的初值。）
12. 类的代码在初次使用时才加载。通常是指加载发生于创建类的第一个对象之时，但是当访问 static 域或 static 方法时，也会发生加载（**构造器也是 static 方法**，尽管 static 关键字并没有显示给出，因此更准确地讲，**类是在其任何 static 成员被访问时加载的**）。初次使用之处也是 static 初始化之处。

假设有一 public 类 A，运行 A.java 代码的过程：第一件事情是试图访问 A.main()（一个 static 方法），于是加载器开始启动并找出 A 类的编译代码（A.class）。在对它进行加载的过程中，如果它有一个基类，于是继续进行加载。如果该基类还有其自身的基类，那么第二个基类就会被加载，如此类推。接下来，根基类中的 static 初始化即被执行，然后是下一个导出类，以此类推，必要的类都加载完后，对象就可以被创建了。

**说明**：不能误认为所有执行都是从 main()开始的，典型的例子是：public 类里有 static field 需要被初始化，而这个 field 在初始化过程中有输出的话，它会优于 main()中的输出，其实想想，如果 static field 没有先被初始化，在 main()中万一用到它，就来不及了。所以类的加载必须保证先对 static field 进行初始化（如果定义处确实需要初始化）。详见 P146。

**总结**：运行 Java 代码虽是从 main()进入的，但在真正执行前，有必要看当前 public 类有木有基类（因为往往导出类会与基类有关联），一直上溯。到最顶层时，应该看 static field 是否

明确要求初始化，若有，必须先初始化，一直向下递推。等所有准备工作都做好了，对象才可以被创建。（这其中体现了一种依赖的思想：**如果 A 的发生是在 B 已经发生的前提下进行的，那么要使得 A 发生，必须确保 B 已经发生。**类的继承、static field（属于类的）即是此。）

13. 尽管面向对象编程对继承极力强调，但在开始一个设计时，一般应优先选择使用组合（或者可能是代理[代理使得该类中的方法不必完全暴露在使用它的类中，而是可以选择性地调用它的方法。即代理类本身需要使用其他类的方法，可以选择继承该其他类。但若继承，则该其他类的所有方法均在需要使用它的类中暴露。如果使用代理类，可以创建其他类的对象（如创建 private 的其他类对象），通过代理方法选择性地访问其他类的方法。详见 P131]），只在确实必要时才使用继承。

# 第 8 章 多态

1. Java 中除了 static 方法和 final 方法（**private 方法属于 final 方法**）之外，其他所有的方法都是后期绑定。这意味着通常情况下，我们不必判定是否应该进行后期绑定—它会自动发生。

2. 只有非 private 方法才可以被覆盖，在导出类中，对于基类中的 private 方法，最好采用不同的名字。

3. Java 中不具有多态性的情况：

（1）static、final（private）方法；

（2）在编译期进行解析的 field，通常为类定义中已初始化的基本数据类型。

4. 对象初始化顺序：

（1）在其他任何事物发生之前，将分配给对象的存储空间初始化为二进制的零；

（2）调用基类构造器。这个步骤会不断地反复递归下去，首先是构造这种层次结构的根，然后是下一层导出类，等等，直到最低层的导出类；（当然，如果基类中有成员对象，先对成员对象进行初始化，这一点和 C++一致，也是体现依赖的思想。）

（3）按声明顺序调用成员的初始化方法；（指最低层的导出类中的成员）

（4）调用导出类构造器的主体。

即：基类成员对象构造器、基类构造器、导出类成员对象构造器、导出类构造器。

5. **不能降低从基类继承的方法的可见性。**因为，每个子类的实例都应该是一个基类的有效实例，如果降低了方法的可见性，那么就相当于子类失去了一个父类的方法，这个时候，父类将不是一个有效的基类。

# 第 9 章 接口

0. 抽象类和接口都是为了**继承**而存在的，所以 field 和 method 定义为 public 才是合理的！

1. 抽象方法（相当于 C++中的纯虚函数）声明：abstract void f(); 包含**抽象方法**的类叫做**抽象类**。如果一个类包含一个或多个抽象方法，该类必须被限定为抽象的。否则，编译器就会报错。（如果从一个抽象类继承，并想创建该新类的对象，那么就必须为基类中的所有抽象方法提供方法定义。如果不这样做，那么导出类便也是抽象类，且编译器将会强制我们用 abstract 关键字来限定这个类。）**创建抽象类的对象没有意义。**

2. abstract 强调**可以有**没有实现的方法；
interface 强调**根本没有**实现了的方法；接口抽象程度更高！
3. 接口中可以包含 field，但是这些 field 隐式地是 static 和 final 的。（抽象类则不然）
4. 可以选择在**接口中**显式地将**方法**声明为 public 的，但即使你不这么做，它们也**是 public 的**。
5. **抽象类和接口的区别**：
（1）语法层面上的区别
1) 接口中不能有构造方法，而抽象类可以有构造方法；
2) 抽象类可以提供成员方法的实现细节，而接口中只能存在 public abstract 方法；
3) 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 public static final 类型的；
4) 接口中不能含有静态代码块以及静态方法，而抽象类可以有静态代码块和静态方法；
5) 一个类只能继承一个抽象类，而一个类却可以实现多个接口。
（2）设计层面上的区别
1) **抽象类是对一种事物的抽象，即对类抽象**，而**接口是对行为的抽象**。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。（继承是一个"是不是"的关系，而接口实现则是"有没有"的关系。如果一个类继承了某个抽象类，则子类必定是抽象类的种类，而接口实现则是有没有、具备不具备的关系。）
2) 对于抽象类，如果需要添加新的方法，可以直接在抽象类中添加具体的实现，子类可以不进行变更；而对于接口则不行，如果接口进行了变更，则所有实现这个接口的类都必须进行相应的改动。
6. toString()方法是根类 Object 的一部分；
7. **使用接口**的核心**原因**是：为了能够**向上转型为多个基类型**（以及由此而带来的灵活性）。第二个原因与使用抽象基类相同：**防止**客户端程序员**创建该类的对象**，并确保这仅仅是建立一个接口。如果要创建**不带任何方法定义和成员变量**的基类，那么就应该选择接口而不是抽象类。事实上，如果知道某事物应该成为一个基类，那么第一选择应该是使它成为一个接口。
8. 一般情况下，只可以将 extends 用于单一类，但是可以引用多个基类接口，只需用逗号将接口名一一分隔开即可。（**Java 不支持多重继承，但可以用接口实现**）

# 第 10 章 内部类

1. 当生成一个内部类的对象时，此对象与制造它的外围对象之间就有了一种联系，所以它能访问其外围对象的所有成员，而不需要任何特殊条件。此外，内部类还拥有其外围类的所有元素的访问权。
2. 内部类的对象只能在与其外围类的对象相关联的情况下才能被创建（在内部类是非 static 类时）。构建内部类对象时，需要一个指向其外围类对象的引用。如果编译器访问不到这个引用就会报错。**在拥有外部类对象之前是不可能创建内部类对象的**。这是因为内部类对象会暗暗地连接到创建它的外部类对象上。但是，如果你创建的是嵌套类（静态内部类），那么它就不需要对外部类对象的引用。
例：

```
public class DotNew{
    public class Inner{}
    public static void main(String[] args){
        DotNew dn = new DotNew();
        DotNew.Inner dni = dn.new Inner();
    }
}
```

3. 匿名内部类：

```
//: innerclasses/Contents.java
public interface Contents {
  int value();
} ///:~//: innerclasses/Parcel7.java
// Returning an instance of an anonymous inner class.
public class Parcel7 {
  public Contents contents() {
    return new Contents() { // Insert a class definition
      private int i = 11;
      public int value() { return i; }
    }; // Semicolon required in this case
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Contents c = p.contents();
  }
} ///:~
```

在匿名内部类末尾的分号，并不是用来标记此内部类结束的。它标记的是表达式的结束，只不过这个表达式正巧包含了匿名内部类罢了。

在匿名类中定义字段时，还能够对其执行初始化操作。如果定义一个匿名内部类，并且希望它使用一个在其外部定义的对象，那么编译器会要求其参数引用是 final 的。

**匿名内部类**与正规的继承相比有些受限，因为匿名内部类既可以扩展类，也可以实现接口，但是不能两者兼备。而且如果是**实现接口，也只能实现一个接口**。

4. 如果不需要内部类对象与其外围类对象之间有联系，可以将内部类声明为 static，这通常称为嵌套类。它意味着：

1) 要创建嵌套类的对象，并不需要其外围类的对象。
2) 不能从嵌套类的对象中访问非静态的外围类对象。

5. 正常情况下，不能在接口内部放置任何代码，但嵌套类可以作为接口的一部分。你放到接口中的任何类都自动地是 public 和 static 的。

6. 有一个外围类 A，含有一个内部类 InnerClass。当有另外一个类 B 去继承这个外围类 A，并且这个派生类 B 也有一个与它基类同名的内部类 InnerClass 时，覆盖并不会发生。这两个内部类是完全独立的两个实体，各自在自己的命名空间内。

7. 局部内部类：在一个方法体的里面创建内部类。局部内部类不能有访问说明符（public、protected 等），因为它不是外围类的一部分；但是它可以访问当前代码块内的常量，以及此外围类的所有成员。

8. 每个类都会产生一个.class 文件，其中包含了如何创建该类型的对象的全部信息（此信息

产生一个“meta-class”，叫做 Class 对象）。内部类生成一个.class 文件，命名为：外围类的名字加上“$”，再加上内部类的名字。

9. 为什么需要内部类？内部类会对类的加载顺序有什么影响？

# 第 11 章 持有对象

1. **泛型**允许我们为集合提供一个可以容纳的对象类型，因此，如果你添加其它类型的任何元素，它会在**编译时**报错。这避免了在运行时出现 ClassCastException，因为你将会在编译时得到报错信息。
2. 容器类的基本类型包括 **List**、**Set**、**Queue** 和 **Map**。这些对象类型也称为**集合类**。**Set** 对于每个值都只保存一个对象，**Map** 是允许你将某些对象与其他一些对象关联起来的关联数组。
3. 如果一个类没有显式地声明继承自哪个类，那么它自动地继承自 **Object**。（这其实隐含着它可以调用 Object 所拥有的方法）
4. 如果像这样 ArrayList arraylist = new ArrayList(); 那么 ArrayList 保存的是 Object，因此你可以通过 add()方法将任何对象添加进去。这意味着当你通过 get()方法取出来添加进去的对象时，你得到的只是 Object 引用。
5. 当你指定了某个类型作为泛型参数时，并不仅限于只能将该确切类型的对象放置到容器中，**向上转型**（该确切类型的子类型的对象也可以放置到该容器中）也可以像作用于其他类型一样作用于泛型。
6. Autoboxing：
7. HashSet 是最快的获取元素方式；
   TreeSet 按照比较结果的升序保存对象；
   LinkedHashSet 按照被添加的顺序保存对象；
   Map 对于每一个键，只接受存储一次；它会自动地调整尺寸；
   HashMap 也提供了最快的查找技术，也没有按照任何明显地顺序来保存其元素；
   TreeMap 按照比较结果的升序保存键；
   LinkedHashMap 按照插入顺序保存键，同时还保留了 HashMap 的查询速度。