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。
（3）新增：
* Java8 接口中可以定义实现了的方法 关键字：default
* 实现该接口的类仍然可以覆盖该实现了方法
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。

Java 编程思想 - Java 类型信息

1. 在 Java 中，所有的类型转换都是在运行时进行正确性检查的。这也是 RTTI 名字的含义：**在运行时，识别一个对象的类型**；
2. 当使用.class 来创建对对象的引用时，**不会自动地初始化该 Class 对象**（即还未构造出一个对象）；但是，为了产生 Class 引用，Class.forName()**立即就进行初始化**。

3. 如果一个 static final 值是“编译期常量”（在编译期已经能确定其值，如 static final int a = 1;），那么这个值不需要对类进行初始化就可以被读取（必须确保既是 static 的又是 final 的）。但是，如果只是将一个 field 设置为 static final 的，还不足以确保这种行为，例如，当该 field 是通过随机方法生成的(如 static final int b = new Random().nextInt(1000);)，那么将强制进行类的初始化，因为它不是一个编译期常量。如果一个 static field 不是 final 的，那么在对它访问时，总是要求在它被读取之前，要先进行链接（为这个 field 分配存储空间）和初始化（初始化该存储空间）
4. Integer Class 对象不是 Number Class 对象的子类？？？（猜测原因：因为擦除的缘故）
5. 通配符是 Java 泛型的一部分，**通配符就是“?”，表示“任何事物”**。通配符**与“extends”关键字配合使用，创建一个范围**，如 Class<? extends Number> class = int.class;

6. A super B 表示 A 是 B 的父类或者祖先

   A extends B 表示 A 是 B 的子类或者子孙

7. 14.3 例子程序实现与理解？？？
8. instanceof 保持了类型的概念，它指的是“你是这个类吗？或者你是这个类的派生类吗？”，等同于 isInstance();而如果用==比较实际的 Class 对象，就没有考虑继承------它或者是这个确切的类型，或者不是。
9. 当通过反射与一个未知类型的对象打交道时，JVM 只是简单地检查这个对象，看它属于哪个特定的类（就像 RTTI 那样）。在用它做其他事情之前必须先加载那个类的 Class 对象。因此，那个类的.class 文件对于 JVM 来说必须是可获取的：要么在本地机器上，要么可以通过网络取得。所以**RTTI 和反射之间真正的区别只在于，对 RTTI 来说，编译器在编译时打开和检查.class 文件。（换句话说，我们可以用“普通”方式调用对象的所有方法。）而对于反射机制来说，.class 文件在编译时是不可获取的，所以是在运行时打开和检查.class 文件。**
10. 14.8 以后未读

Java 编程思想 - 泛型

1. Java 泛型的核心概念是：告诉编译器想使用什么类型，然后编译器帮你处理一切细节。
2. 在泛型代码内部，无法获得任何有关泛型参数类型的信息；
3. 泛型类型只有在静态类型检查（ Java 使用静态类型检查在编译期间分析程序，确保没有类型错误。详见：http://www.cnblogs.com/chenpi/p/5503997.html）期间才出现，在此之后，程序中的所有泛型类型都将被擦除，替换为它们的非泛型上界。例如，

诸如 List<T>这样的类型注解将被擦除为 List，而普通的类型变量在未指定边界的情况下将被擦除为 Object。

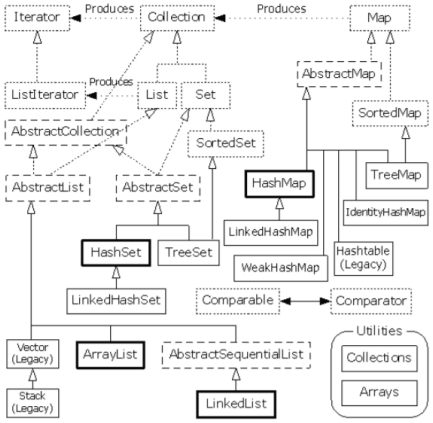
4. 通过允许非泛型代码与泛型代码共存，擦除使得这种向着泛型的迁移成为可能。擦除的主要正当理由是从非泛化代码到泛化代码的转变过程，以及在不破坏现有类库的情况下，将泛型融入 Java 语言。擦除使得现有的非泛型客户端代码能够在不改变的情况下继续使用，直至客户端准备好用泛型重写这些代码。
5. 在泛型中的所有动作都发生在边界处---对传递进来的值进行额外的编译期检查，并插入对传递出去的值的转型。这有助于澄清对擦除的混淆，记住：**“边界就是发生动作的地方”**。（理解此处的边界处行为可以结合常规的 set()和 get()方法：set() 接收传递进来的值，要进行编译期检查；get() 将值传递出去，要做类型检查）
6. **不能创建泛型数组，如：T[] array = new T[SIZE]；是不能通过编译器检查的；**一般的解决方案是在任何想要创建泛型数组的地方都使用 ArrayList；因为有了擦除，数组的运行时类型就只能是 Object[]。
7. 15.9 以后未仔细读

Java 编程思想 - 数组

1. 在 Java 中，**数组是一种效率最高的存储和随机访问对象引用序列的方式**。数组就是一个简单的线性序列，这使得元素访问非常快速。但是为这种速度所付出的代价是**数组对象的大小被固定，并且在其生命周期中不可改变**。
2. 对象数组保存的是引用，基本类型数组直接保存基本类型的值。
3. 不能实例化具有参数化类型的数组：Peel<Banana>[] peels = new Peel<Banana>[10]; // Illegal 擦除会移除参数类型信息，而数组必须知道它们所持有的确切类型，以强制保证类型安全。但是可以参数化数组本身的类型。
4. 尽管不能创建实际的持有泛型的数组对象，但是你可以创建非泛型的数组，然后将其转型：List<String>[] ls; List[] la = new List[10]; ls = (List<String>[])la; // "Unchecked warning"
5. System.arraycopy()：如果复制对象数组，那么只是复制了对象的引用，而不是对象本身的拷贝（这被称作：浅拷贝）。并且该方法不会执行自动包装和自动拆包。

Java 编程思想 - 容器深入研究

1. a more complete diagram of the collections library, including abstract classes and legacy components:



Full Container Taxonomy

2. Collection 中的常用操作比较：
   - boolean **addAll**(Collection<? extends T>): Adds all the elements in the argument. Returns true if **any** elements were added. ("Optional.")
   - Boolean **retainAll**(Collection<?>) : Retains only elements that are contained in the argument (an "**intersection**," from set theory). Returns true if any changes occurred. ("Optional.")
3. Arrays.asList()返回**固定尺寸**的 List，它基于一个固定大小的数组，仅支持那些不会改变数组大小的操作。任何会引起对底层数据结构的尺寸进行修改的方法都会产生一个 UnsupportedOperationException 异常。但是可以修改指定的元素。
4. Collections.unmodifiableList()产生不可修改的列表，其结果在任何情况下都应该不是可修改的。
5. 关于 Set 的一些定义：
   - **Set**：存入 Set 的每个**元素**都必须是**唯一**的，因为 **Set 不保存重复的元素**。加入 Set 的元素**必须定义 equals()**方法以确保对象的唯一性。Set 与 Collection 有完全一样的接口。Set 接口不保证维护元素的次序；
   - **HashSet**：为**快速查找**而设计的 Set。存入 HashSet 的元素**必须定义 hashCode()**；
   - **TreeSet**：**保持次序**的 Set，底层为树结构。使用它可以从 Set 中提取有序的序列。元素**必须实现 Comparable 接口**；
   - **LinkedHashSet**：具有 HashSet 的查询速度，且内部使用链表维护元素的顺序（插入的次序）。于是在使用迭代器遍历 Set 时，结果会按元素插入的次序显示。元素也必须定义 hashCode()方法；

6. Map 中：**任何键都必须具有一个 equals()方法**；如果键被用于散列 Map，那么它必须还具有恰当的 hashCode()方法；如果键被用于 TreeMap，那么它必须实现 **Comparable**
7. **Object 的 hashCode()方法默认是使用对象的地址计算散列码**
8. 正确的 equals()方法必须满足下面 5 个条件：
    1. 自反性。对任意 x， x.equals(x)一定返回 true；
    2. 对称性。对任意 x，y，如果 y.equals(x)返回 true，那么 x.equals(y)也返回 true；
    3. 传递性。对任意 x，y，z，如果有 x.equals(y)返回 true，y.equals(z)返回 true， 则 x.equals(z)一定返回 true；
    4. 一致性。对任意 x 和 y，如果对象中用于等价比较的信息没有改变，那么无论调用 x.equals(y)多少次，返回的结果应该保持一致，要么一直为 true，要么一直为 false；
    5. 对任何不是 null 的 x， x.equals(null)一定返回 false。
9. **默认的 Object.equals()只是比较对象的地址**
10. 如果 instanceof 左边的参数为 null，它会返回 false
11. Joshua Bloch gives a basic recipe for generating a decent hashCode( ):
    0. Store some constant nonzero value, say 17, in an int variable called result.
    1. For each significant field f in your object (that is, each field taken into account by the equals( ) method), calculate an int hash code c for the field:

Field type	Calculation
boolean	c = (f ? 0 : 1)
byte, char, short, or int	c = (int)f
long	c = (int)(f ^ (f>>>32))
float	c = Float.floatToIntBits(f);
double	long l = Double.doubleToLongBits(f); c = (int)(l ^ (l>>>32))
Object, where equals( ) calls equals( ) for this field	c = f.hashCode( )
Array	Apply above rules to each element

    2. Combine the hash code(s) computed above: **result = 37 * result + c**;
    3. Return result.
    4. Look at the resulting hashCode( ) and make sure that equal instances have equal hash codes.
12. ArrayList 源码分析：底层使用**数组**实现
    0. 数据存储在 Object[] 数组里，通过 Arrays.copyOf(Object[] arg, int newLength)方法动态扩充数组容量；
    1. 每次调用 add()时，会去检查当前数组长度是否满足新元素的插入，需要扩充容量时才会动态扩充数组容量，否则，直接将新元素插入当前数组即可；
13. LinkedList 源码分析：底层使用**双向链表**实现
14. @Todo HashSet 源码分析：
15. @Todo HashMap 源码分析：
16. 对 List 的选择：将 ArayList 作为默认首选，只有你需要使用额外的功能，或者当程序的性能因为经常从表中间进行插入和删除而变差时，才去选择 LinkedList。

17. 对 Set 的选择：HashSet 的性能基本上总是比 TreeSet 好，特别是在添加和查询元素时，而这两个操作也是最重要的操作。TreeSet 存在的唯一原因是它可以维持元素的排序状态；所以，只有当需要一个排好序的 Set 时，才应该使用 TreeSet。用 TreeSet 迭代通常比用 HashSet 要快。
18. 对 Map 的选择：当使用 Map 时，第一选择应该是 HashMap，只有在你要求 Map 始终保持有序时，才需要使用 TreeMap。LinkedHashMap 在插入时比 HashMap 慢一点，因为它维护散列数据结构的同时还要维护链表（以保持插入顺序）。正是由于这个列表，使得其迭代速度更快。
19. HashMap 的性能因子：
    - 容量：表中的桶位数；
    - 初始容量：表在创建时所拥有的桶位数。HashMap 和 HashSet 都具有允许你指定初始容量的构造器；
    - 尺寸：表中当前存储的项数；
    - 负载因子：尺寸/容量。空表的负载因子是 0，而半满表的负载因子是 0.5，依此类推。负载轻的表产生冲突的可能性小，因此对于插入和查找都是最理想的（但是会减慢使用迭代器进行遍历的过程）。HashMap 和 HashSet 都具有允许你指定负载因子的构造器，**表示当负载情况达到该负载因子的水平时，容器将自动增加其容量（桶位数），实现方式是使容量大致加倍，并重新将现有对象分布到新的桶位集中（这被称为再散列）。**
    - **HashMap 使用的默认负载因子是 0.75（只有当表 达到 3/4 满时，才进行再散列），这个因子在时间和空间代价之间达到了平衡。**更高的负载因子可以降低表所需的空间，但是会增加查找代价，这很重要，因为查找是我们子啊大多数时间里所做的操作。
    - 如果你知道将要在 HashMap 中存储多少项，那么创建一个具有恰当大小的初始容量将可以避免自动再散列的开销。
20. Java 容器类类库采用**快速报错（fail-fast）**机制。它会探查容器上的任何除了你的进程所进行的操作以为的所有变化，一旦它发现其他进程修改了容器，就会立刻抛出 ConcurrentModificationException 异常。这就是“快速报错”的意思--即，不是使用复杂的算法在事后来检查问题。

Java 编程思想 - Java I/O

1. 通过继承，任何自 **InputStream 或 Reader** 派生而来的类都含有名为 read()的基本方法，用于读取单个字节或者字节数组。同样，任何自 **OutputStream 或 Writer** 派生而来的类都含有名为 write()的基本方法，用于写单个字节或者字节数组。
2. InputStream 的作用是用来表示那些从不同数据源产生输入的类。这些数据源包括：
    - 字节数组；**ByteArrayInputStream**
    - String 对象；**StringBufferInputStream**
    - 文件；**FileInputStream**

- “管道”，工作方式与实际管道相似，即，从一端输入，从另一端输出。**PipedInputStream**
- 一个由其他种类的流组成的序列，以便我们可以将它们收集合并到一个流内。**SequenceInputStream**
- 其他数据源，如 Internet 连接等。

3. 设计 Reader 和 Writer 继承层次结构主要是为了国际化。老的 I/O 流继承层次（InputStream/OutputStream）仅支持 8 位字节流，并且不能很好地处理 16 位的 Unicode 字符。由于 Unicode 用于字符国际化（Java 本身的 char 也是 16 位的 Unicode），所以添加 Reader 和 Writer 继承层次结构就是为了在所有的 I/O 操作中都支持 Unicode。另外，新类库的设计使得它的操作比旧类库更快。**InputStreamReader 可以把 InputStream 转换成 Reader，而 OutputStreamWriter 可以把 OutputStream 抓换成 Writer。**
4. 尽量**尝试**使用 Reader 和 Writer，一旦程序代码无法成功编译，就会发现不得不使用面向字节的类库。
5. 无论我们何时使用 readLine()，都不应该使用 DataInputStream（这会遭到编译器的强烈反对），而应该使用 BufferedReader。除此之外，DataInputStream 仍是 I/O 类库的首选成员。
6. Java 的 System 类提供了一些简单的静态方法调用，以允许我们对标准输入、输出和错误 I/O 流进行重定向。 **setIn(InputStream) setOut(PrintStream) setErr(PrintStream) I/O 重定向操纵的是字节流，而不是字符流**；因此我们使用的是 InputStream 和 OutputStream，而不是 Reader 和 Writer。
7. 文件加锁对其他的操作系统进程是可见的，因为 Java 的文件加锁直接映射到了本地操作系统的加锁工具。
8. tryLock()是非阻塞式的，它设法获取锁，但是如果不能获得（当其他一些进程已经持有相同的锁，并且不共享时），它将直接从方法调用返回。lock()是阻塞式的，它要阻塞进程直至锁可以获得，或调用 lock()的线程中断或通道关闭。
9. It is also possible to lock a part of the file by using **tryLock(long position, long size, boolean shared)** or **lock(long position, long size, boolean shared)** which locks the region (size - position). The third argument specifies whether this lock is shared.
10. 尽管无参数的加锁方法将根据文件尺寸的变化而变化，但是具有固定尺寸的锁不随文件尺寸的变化而变化。如果你获得了某一区域（从 position 到 position+size）上的锁，当文件增大超出 position+size 时，那么在 position+size 之外的部分不会被锁定。无参数的加锁方法会对整个文件进行加锁，甚至文件变大后也是如此。
11. 对独占锁或者共享锁的支持必须由底层的操作系统提供。如果操作系统不支持共享锁并为每一个请求都创建一个锁，那么它就会使用独占锁。锁的类型（共享或独占）可以通过 FileLock.isShared()进行查询。
12. 当你创建对象时，只要你需要，他就会一直存在。但是在程序终止时，无论如何他都不会继续存在。如果对象能够在程序不运行时仍能保存其信息，那将非常有用。这样，在下次运行程序时，该对象将被重建并且拥有的信息与程序上次运行时她所拥有的信息相同。当然，你也可以将对象信

息也如文件或者数据库来实现相同的效果，但是在万物都是对象的精神中，如果能够将一个对象声明为持久化的，并为我们处理掉所有细节，那将会显得十分方便。

Java 的对象序列化将那些实现了 Serializable 接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象。“持久性”就会意味着对象的生存周期并不取决于程序是否在执行。

通过将一个序列化对象写入磁盘，然后再重新调用程序时恢复该对象，就能够实现持久性的效果。必须在程序中显式地序列化（serialze）和反序列化还原（deserialize）。如果需要一个更严格的持久性机制，可以考虑 Hibernate 之类的工具。

序列化主要是为了支持两种特性：一是 Java 的远程方法调用（RMI）；二是 Java Beans。

13. 在对一个 Serializable 对象进行还原的过程中，没有调用任何构造器，包括默认的构造器。整个对象都是通过从 InputStream 中取得数据恢复而来的。
14. 对于 Serializable 对象，对象完全以它存储的二进制位为基础来构造，而不调用构造器。而对于一个 Externalizable 对象，所有普通的默认构造器都会被调用（包括在字段定义时的初始化），然后调用 readExternal()。必须注意这一点------所有默认的构造器都会被调用，才能使 Externalizable 对象产生正确的行为。
15. 由于 Externalizable 对象在默认情况下不保存它们的任何字段，所以 **transient 关键字只能和 Serializable 对象一起使用**。
16. 静态成员是不能被序列化的，因为静态成员是随着类的加载而加载的，与类共存亡，并且静态成员的默认初始值都是 0；

Java 编程思想 - 并发

1. 静态方法 Thread.yield() ：对该方法的调用**是对 线程调度器** （Java 线程机制的一部分，可以将 CPU 从一个线程转移给另一个线程）**的一种建议**，它在声明：“我已经执行完生命周期中最重要的部分了，此刻正是切换给其他任务执行一段时间的大好时机”。这完全是选择性的，并**没有任何机制保证它会被采纳**。
2. 一个后台（Daemon）线程创建的任何线程将被自动地设置成后台线程。当最后一个非后台线程终止时，后台线程会“突然”终止（如果有 finally，也依然来不及执行了），JVM 会立即关闭所有后台线程。
3. 执行的任务与驱动它的线程：对 Thread 类实际没有任何控制权，**创建任务，并通过某种方式将一个线程附着到任务上，以使得这个线程可以驱动任务。**
4. 如果某个线程 a 在另一个线程 t 上调用 t.join()，此线程 a 将被挂起，直到目标线程 t 结束才恢复。（即 t.isAlive() 返回 false）可以这样理解：在某线程 x 上调用 join()方法，相当于让该线程 x 现在 join 进来，CPU 先去调度线程 x。

5. 用 volatile 修饰的变量：
    -
        - 使用 volatile 关键字会强制将修改的值立即写入主存；
        - 使用 volatile 关键字的话，当线程 2 进行修改时，会导致线程 1 的工作内存中缓存变量的缓存行无效(反映到硬件层的话，就是 CPU 的 L1 或者 L2 缓存中对应的缓存行无效）；
        - 由于线程 1 的工作内存中缓存变量的缓存行无效，所以线程 1 再次读取变量的值时会去主存读取。
    - 缓存一致性协议。最出名的就是 Intel 的 MESI 协议，MESI 协议保证了每个缓存中使用的共享变量的副本是一致的。它核心的思想是：当 CPU 写数据时，如果发现操作的变量是共享变量，即在其他 CPU 中也存在该变量的副本，会发出信号通知其他 CPU 将该变量的缓存行置为无效状态，因此当其他 CPU 需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。
6. To control access to a shared resource, you first put it inside an object. Then any method that uses the resource can be made synchronized. If a task is in a call to one of the synchronized methods, all other tasks are blocked from entering any of the synchronized methods of that object until the first task returns from its call.【要控制对共享资源的访问，得先把它包装进一个对象。然后把所有要访问这个资源的方法标记为 synchronized。如果某个任务处于一个对标记为 synchronized 的方法的调用中，那么在这个线程从该方法返回之前，其他所有要调用类中任何标记为 synchronized 方法的线程都会被阻塞。 】
7. 所有的对象都自动含有单一的锁（也成为监视器）。**当在对象上调用其任意 synchronized 方法时，此对象都被加锁，这时该对象上的其他 synchronized 方法只有等到前一个方法调用完毕并释放了锁之后才能被调用。所以，对于某个特定对象来说，其所有 synchronized 方法共享同一个锁。**
8. 注意，在使用并发时，将域设置为 private 是非常重要的，否则，synchronized 关键字就不能防止其他任务直接访问域，这样就会产生冲突。
9. 一个任务可以多次获得对象的锁。如果一个方法在同一个对象上调用了第二个方法,后者又调用了同一个对象上的另一个方法,就会发生这种情况。JVM 负责跟踪对象被加锁的次数。如果一个对象被解锁（即锁被完全释放），其计数变为 0.在任务第一次给对象加锁时，计数变为 1.每当这个相同的任务在这个对象上获得锁时，计数都会递增。显然，只有首先获得了锁的任务才能允许继续获取多个锁。每当任务离开一个 synchronized 方法，计数递减，当计数为零时，锁被完全释放，此时别的任务就可以使用此资源。
10. 针对每个类，也有一个锁（作为类的 Class 对象的一部分），所以 synchronized static 方法可以在类范围内防止对 static 数据的并发访问。

11. 每个访问临界共享资源的方法都必须被同步，否则它们就不会正确地工作。
12. **原子操作**是不能被线程调度机制中断的操作。
13. 如果你将一个域声明为 volatile 的，那么只要对这个域产生了写操作，所有的读操作就都可以看到这个修改。即便使用了本地缓存，情况也确实如此，volatile 域会立即被写入到主存中，而读操作就发生在主存中。**当一个域的值依赖于它之前的值时（例如递增一个计数器），volatile 就无法工作了。如果某个域的值受到其他域的值限制，那么 volatile 也无法工作，例如 Range 类的 lower 和 upper 边界就必须遵循 lower<=upper 的限制。**（详见 Java 学习 - 并发杂记笔记 并发杂记）
14. 如果多个任务在同时访问某个域，那么这个域就应该是 volatile 的，否则，这个域就应该只能经由同步来访问。**同步也会导致向主存中刷新**，因此如果一个域完全由 synchronized 方法或语句块来防护，那就不必将其设置为是 volatile 的。
15. **你的第一选择应该是使用 synchronized 关键字，这是最安全的方式，而尝试其他任何方式都是有风险的。**
16. 线程状态：new、Runnable、Blocked、Dead
    1. 新建（new）：当线程被创建时，它只会短暂地处于这种状态。此时它已经分配了必需的系统资源，并执行了初始化。此刻线程已经有资格获得 CPU 时间了，之后调度器将把这个线程转变为可运行状态或阻塞状态。（如果此时调用 isAlive()方法，将返回 false）
    2. 就绪（Runnable）：在这种状态下，只要调度器把时间片分配给线程，线程就可运行。也就是说，在任意时刻，线程可以运行也可以不运行。只要调度器能分配时间片给线程，它就可以运行；这不同于死亡和阻塞状态。
    3. 阻塞（Blocked）：线程能够运行，但有某个条件阻止它运行。当线程处于阻塞状态时，调度器将忽略线程，不会分配给线程任何 CPU 时间。直到线程重新进入了就绪状态，它才有可能执行操作。
    4. 死亡（Dead）：处于死亡或终止状态的线程将不再是可调度的，并且再也不会得到 CPU 时间，它的任务已结束，或不再是可运行的。任务死亡的通常方式是从 run()方法返回，但是任务的线程还可以被中断。
17. jdk 中对于线程状态的描述：

    A thread can be in one of the following states:
    - NEW
      A thread that has not yet started is in this state.
    - RUNNABLE
      A thread executing in the Java virtual machine is in this state.
    - BLOCKED
      A thread that is blocked waiting for a monitor lock is in this state.

- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.

18. 一个任务进入阻塞状态，可能有如下原因：
    1. 通过调用 **sleep**(milliseconds)使任务**进入休眠状态**，在这种情况下，任务在指定的时间内不会运行；
    2. 通过调用 **wait()使线程挂起**。直到线程得到了 notify()或 notifyAll()消息（或者在 java.util.concurrent 类库中等价的 signal()或 signalAll()消息），线程才会进入就绪状态；
    3. 任务在**等待**某个**输入/输出**完成；
    4. 任务试图在某个对象上调用其同步控制方法，但是**对象锁不可用**，因为另一个任务已经获取了这个锁。
19. I/O 和在 synchronized 块上的等待是不可中断的；**你能够中断对 sleep 的调用（或者任何要求抛出 InterruptedException 的调用 ）。但是你不能中断正在试图获取 synchronized 锁或者试图执行 I/O 操作(PipedReader 是可中断的)的线程。**
20. 一旦底层资源被关闭，任务将解除阻塞。
21. 无论在任何时刻，只要任务以不可中断的方式被阻塞，那么都有潜在的会锁住程序的可能。Java SE5 并发类库中添加了一个特性，即在 ReentrantLock 上阻塞的任务具备可以被中断的能力（通过调用 lockInterruptibly()方法，如果有 interrupt()方法中断，则可以响应并抛出 InterruptedException 异常；但是如果只是单纯地调用 lock()方法，则永远不能被中断）。
22. 调用 sleep()的时候锁并没有被释放，yield()也是如此。但是当一个任务在方法里遇到了对 wait()的调用时，线程的执行被挂起，对象上的锁被释放。因为 wait()将释放锁，这就意味着另一个任务可以获得这个锁，因此在该对象（现在是未锁定的）中的其他 synchronized 方法可以在 wait()期间被调用。【书前面 sleep()和 yield()的例子通常用来配合同步的演示，到这里，讲 wait()时已经开始进入线程之间的协作了】
23. 只能在同步控制方法或同步控制块里调用 wait()、notify()、notifyAll()（因为不用操作锁，所以 sleep()可以在非同步控制方法里调用）。调用 wait()、notify()、notifyAll()的任务在调用这些方法前必须“拥有”（获取）对象的锁。

24. 当以下四个条件同时满足时，就会发生死锁：
    1. **互斥条件**。任务使用的资源中至少有一个是不能共享的；
    2. **持有与等待**。至少有一个任务它必须持有一个资源且正在等待获取一个当前被别的任务持有的资源；
    3. **不剥夺条件**。资源不能被任务抢占；
    4. 必须有**循环等待**。一个任务等待其他任务所持有的资源，后者又在等待另一个任务所持有的资源，这样一直下去，直到有一个任务在等待第一个任务所持有的资源，使得大家都被锁住。
25. 因为要发生死锁的话，所有这些条件必须全部满足；所以要防止死锁的话，只需破坏其中一个即可。其中最容易的是破坏第四个条件。
26. 新类库中的构件：
    1. CountDownLatch：它被用来同步一个或多个任务，强制它们等待由其他任务执行的一组操作完成。可以向 CountDownLatch 对象设置一个初始计数值，任何在这个对象上调用 await()的方法都将阻塞，直至这个计数值到达 0。其他任务在结束其工作时，可以在该对象上调用 countDown()来减小这个计数值。CountDownLatch 被设计为只触发一次，计数值不能被重置。如果你需要能够重置计数值的版本，则可以使用 CyclicBarrier。调用 countDown()的任务在产生这个调用时并没有被阻塞，只有对 await()的调用会被阻塞，直至计数值到达 0。**CountDownLatch 的典型用法是将一个程序分为 n 个互相独立的可解决任务，并创建值为 n 的 CountDownLatch。当每个任务完成时，都会在这个 latch（锁存器）上调用** countDown()。**等待问题被解决的任务在这个** latch **上调用** await()，**将它们自己拦住，直至** latch **计数结束。**
    2. CyclicBarrier：