红岩Android第三次课 - java面向对象进阶

1. final和static关键字

1.1 final关键字

`final`是`最终的`意思，它作为java里的一个关键字，可以用来**修饰变量、方法和类**，分别表示**变量不可变(常量)，方法不可覆盖和类不可继承。**即被final修饰的内容一旦赋值之后，就不能再被改变。

接下来我们从这三个方面来介绍final的特性。

1.1.1 变量不可变

不可变的变量也叫做常量。java中的变量可以分为**成员变量**和**局部变量**，我们就分别以这两种情况来说明被final修饰的变量情况。

1.1.1.1 final修饰成员变量

通常每个类中的成员变量又分为两种：**类变量**和**实例变量。**

类变量即被static修饰的变量，**它属于类本身，而不属于任何对象。**这个我们放在后面介绍`static关键字`的时候再讲。

我们先来看看final修饰类中**实例变量**的情况。在开始之前，我们需要注意的有两点：

1. 被`final`修饰的变量**不会被系统进行隐式初始化。**

   > 什么是隐式初始化？一般情况下，对于类中未被初始化的非final成员变量，都会在调用前被赋予一个默认值，例如，int类型的变量会被初始化为0，float、double类型的变量会被初始化为0.0，boolean类型的变量会被初始化为false，String或者其他对象会被初始化为null。
   >
   > 与之相对应的显示初始化，即我们通过new关键字，调用一个类中的构造函数，通过构造函数来创建一个对象。

2. 被`final`修饰的变量**不能被再次改变其内容。**

既不能被自动赋值，也不能被二次修改，因此我们必须在程序运行之前带给它一个默认的**初始值**，这样才能让被final修饰的变量有意义。

对于实例变量的初始化，有三种不同的方法，这里我们以ClassA为例说明，首先我们在其中定义了一个整型final变量a：

```
public class ClassA {
    final int a; // 报错
}
```

此时你会发现编译器提示有错误代码，将鼠标移至爆红位置可以发现编译器告诉我们，变量a未被初始化(Variable 'a' might not have been initialized)。

由此引出第一种初始化实例变量的方法，**直接赋值**：

```
public class ClassA {
    final int a = 0;
}
```

第二种初始化实例变量的方法，是在类的**实例代码块**(也叫非静态代码块)中对变量进行赋值，如下代码所示，其中`{}`中的内容即为实例代码块，它在一个类对象刚加载时被执行，可以**确保变量在使用之前被初始化**：

```
public class ClassA {
    final int a; // 定义一个final变量a

    // 实例化代码块
    {
        a = 0;
    }
}
```

第三种初始化实例变量的方法，是在**类的构造函数**中对变量进行赋值，如下代码所示：

```
public class ClassA {
    final int a; // 定义一个final变量a

    public ClassA() {
        a = 0;
    }
}
```

> 类的初始化顺序：**静态代码、静态代码块 > 成员变量、实例代码块 > 构造函数。**如果有继承关系，则**先父后子。**

当我们初始化完变量的内容后，可以发现，当我们再次想修改该变量的内容时，会导致程序报错无法正常运行。

1.1.1.2 final修饰局部变量

局部变量即为在函数内部创建的变量，用final对其修饰后只能初始化赋值一次，无法修改其内容：

```
public class ClassA {
    void test(final int a) { // 这里的变量a大小不能被改变
        final int b = 1; // 这里变量b被初始化赋值为1，之后其值无法再被改变
    }
}
```

1.1.1.3 final修饰引用数据类型

上面两个我们讲的都是使用final修饰基本数据类型时的情况，被修饰的变量无法被二次更改其值，那么，对于被final修饰的引用数据类型的值可以被改变吗？

首先引出结论，**final对象本身不可变，但其内包含的字段属性值可以改变。**这里我们以`Main`、`ClassA`和`ClassB`三个类为示例进行说明。

首先分别定义一个类ClassA，其中包含一个整型变量num和一个构造器：

```java
// ClassA.java
public class ClassA {
    public int num;

    public ClassA(int num) {
        this.num = num;
    }
}
```

在`main`方法中，我们新建一个final的ClassA对象，并对其直接进行初始化赋值。此时我们是无法再次对变量classA重写赋值的，但是却可以直接改变ClassA中变量num的值：

```java
// Main.java
public class Main {
    public static void main(String[] args) {
        final ClassA classA = new ClassA(123);
//        classA = new ClassA(456); 这是不允许的
        classA.num = 456; // 这里可以对classA对象中的num变量进行重新赋值
    }
}
```

1.1.2 方法不可覆盖

在面向对象三大特性中，多态常常由**重写**和**重载**构成，这里我们分别以这两种情形介绍final关键字修饰方法时的情况。

1.1.2.1 final修饰的方法不可被重写

这里我们给ClassA新增一个final方法`test()`，并让ClassB继承自ClassA：

```java
// ClassA.java
public class ClassA {
    public ClassA() {
    }

    final void test() {
        System.out.println("ClassA Test");
    }
}

// ClassB.java
public class ClassB extends ClassA{
    public ClassB() {
    }

    void test() {

    }
}
```

可以看到，我们试图在ClassB中重写`test()`方法，但很遗憾的是编译器此时会将此处代码爆红，提醒你当前重写的方法被final修饰，无法重写：

> 'test()' cannot override 'test()' in 'ClassA'; overridden method is final

1.1.2.2 final修饰的方法可以被重载

我们直接对上面例子中的ClassA进行`test()`方法的重载，如下，可正常运行：

```
public class ClassA {
    public ClassA() {
    }

    final void test() {
        System.out.println("ClassA Test");
    }

    void test(int a) {
        System.out.println("重载方法");
    }
}
```

你可以看到，我重载的方法并未使用final修饰，那么这是ClassB是可以直接重写具有一个整型参数的test()方法的。

1.1.3 类不可继承

即使用final修饰的类无法被继承：

```
// ClassA.java
public final class ClassA {
}

// ClassB.java
public class ClassB extends ClassA{
}
```

此时这里的ClassB会报错，提示你无法继承自final类：Cannot inherit from final 'ClassA'。

1.1.4 final的优势

通过上面三种情况的介绍，现在的你对final的使用应该已经具备初步的认知，那么你知道使用final关键字有哪些优势吗？具体的优势有两个方面：

1. 提高了程序运行性能和方法调用速度。

   > 提高性能：JVM在常量池中会缓存final变量。
   >
   > 提高方法调用速度：final方法是静态编译的。

2. 在多线程中安全可共享，无需额外开销。

1.1.5 final进阶

关于final关键字在java多线程中的作用，需要深入到JVM底层知识去学习了解，这里我们就不具体讲了，感兴趣的同学可以去网上搜索相关文章学习，这里提供几篇我参考的final文章：

[你以为你真的了解final吗？]

[重新认识 java（七） - final 关键字]

[final关键字深入解析]

1.2 static关键字

`static`是`静态的`意思，作为java中的一个关键词，它可以用于修饰**变量、方法、代码块和内部类**，表示某个特定的成员，**只属于某个类本身**，而不属于该类的实例化对象。

1.2.1 静态字段

上面这段话是什么意思呢？我们举个简单的例子来说明一下。我们在ClassA中定义一个静态变量a，一个普通变量b以及一个无参构造器：

```
public class ClassA {
    static int a = 0;
    int b = 1;

    public ClassA() {
    }
}
```

首先我们来看看如何在外部类中获取到ClassA的这两个变量的值吧：

```
public class Main {
    public static void main(String[] args) {
        // 获取a的值
        int a = ClassA.a;
        // 获取b的值
        int b = new ClassA().b;
    }
}
```

可以看到，我们可以直接通过`类名`+`.`的方法获取静态变量的引用；而对于普通的变量b，我们需要先创建一个ClassA的对象才能获取到该成员变量的值。这也意味着我们改变ClassA中a、b变量的方法也不一样。

这里我们新建两个不同的ClassA对象，并改变其中普通变量b的值，最后打印输出：

```
public class Main {
    public static void main(String[] args) {
        ClassA classA1 = new ClassA();
        ClassA classA2 = new ClassA();
        classA1.b = 123;
        classA2.b = 456;
        System.out.println("classA1 b = " + classA1.b);
        System.out.println("classA2 b = " + classA2.b);
    }
}
```

最终输出结果如下：

```
classA1 b = 123
classA2 b = 456
```

因此得出结论：**ClassA中的普通变量b被其每一个实例化对象独有，不会在不同对象之间共享该变量**，当我们创建1000个ClassA对象时，就会有1000个变量b。那么与之相对的是静态变量a，它属于ClassA这个类本身，无论我们创建多少个ClassA对象，**静态变量a有且只有一个，且所有该类的实例对象都共享该静态变量a**，所以static也可以表示`全局的`意思。

> 关于字段和变量的区别：事实上这两者经常用来表达相同的意思，但字段可以存储对象的状态属性，但变量并不全都可以这样，例如局部变量和函数形参就不能被称作字段。因此可以得出，字段都是变量，但变量并不全是字段。**static无法作用于局部变量和函数形参**，因此这里的标题我们叫做静态字段，而非静态常量。
>
> class variables and instance variables are fields while local variables and parameter variables are not. All fields are variables.
>
> 参考文章Java 中字段和变量的的区别（Fields vs Variables in Java）
>
> Java 中field 和variable 区别及相关术语解释

1.2.2 静态常量

静态变量平常使用的比较少，但静态常量却经常使用。静态常量即同时使用`final`和`static`关键字来修饰一个变量。例如，我们在写程序要用到PI值时经常会使用`Math.PI`来调用，这里的PI其实就是`Math类`中的一个静态常量：

```
public class Math {
    public static final double PI = 3.14159265358979323846;
}
```

我们平常使用的输出语句`System.out.println();`中的`out`其实也是`System`类中的一个静态常量：

```
public class System {
    public static final PrintStream out = null;
}
```

1.2.3 静态方法

首先我们引入《Java编程思想》中对于静态方法的经典描述：

> **static 方法就是没有 this 的方法，在 static 内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用 static 方法，这实际上是 static 方法的主要用途。**

总结下来静态方法的使用有三个需要注意的地方：

1. 静态方法内不能使用非静态方法
2. 任何方法内部都可直接调用静态方法
3. 调用静态方法不需要创建对象，可直接通过`类名`+`.`访问，这点同静态字段一样

我们在ClassA中分别定义两个方法，一个静态方法fun1，一个非静态方法fun2，并在main函数中分别调用这两个方法：

```
// ClassA.java
public class ClassA {
    public ClassA() {
    }

    static void fun1() {
        System.out.println("这是一个静态方法");
```

```
    }

    void fun2() {
        System.out.println("这是一个非静态方法");
    }
}

//Main.java
public class Main {
    public static void main(String[] args) {
        // 调用静态方法fun1
        ClassA.fun1();
        // 调用非静态方法fun2
        new ClassA().fun2();
    }
}
```

相信细心的你此时已经发现我们**程序的入口`main`函数也是一个静态方法**，事实上，在程序刚刚启动时还不存在任何的对象，此时静态的main方法将执行并构造程序所需要的对象。

另外，**每个类中都可以含有一个main方法用作单元测试**，例如我们在刚刚的ClassA类中添加一个main方法，并点击其旁边的绿色运行按钮，会发现我们的程序只执行了ClassA类中的main方法，而没有执行Main类中的main方法：

```
public class ClassA {
    public ClassA() {
    }

    static void fun1() {
        System.out.println("这是一个静态方法");
    }

    void fun2() {
        System.out.println("这是一个非静态方法");
    }

    public static void main(String[] args) {
        System.out.println("ClassA单元测试");
    }
}
```

1.2.4 静态代码块

在上面*final*修饰成员变量中我们已经提到过，成员变量包含两种，一种是类变量，一种是实例变量。类变量就是这里我们讲的静态变量，对于被final修饰的类变量（即静态常量）有两种赋值方法：

1. 直接赋值，和1.2.2中举的例子相同
2. 在静态代码块中赋值

那么什么是静态代码块呢？其实就是在`{}`的前面使用`static`关键字进行修饰的代码部分，这里我们以ClassA为例来说明如何使用静态代码块为静态常量赋初始值。

```
public class ClassA {
    static final int a; // 静态常量a

    // 这是一个静态代码块
    static {
        a = 0;
    }
}
```

那么对于静态代码块，它的作用不仅仅是给静态常量赋初始值的，正如之前讲使用实例代码块给非静态常量赋初始值那样，实例代码块和静态代码块都在类的构造函数之前被执行。

不仅如此，相比于实例代码块，静态代码块执行的要跟早一些，也就是它在实例代码块之前执行（可以查看上面1.1.1.1中最后的类初始化顺序，另外感兴趣的也可去了解一下，子类静态代码块比父类构造函数还要先执行）；除此之外，一个类中可以包含多个静态代码块，在类被初次加载时，静态代码块会按照顺序依次执行，且从始至终只执行一次，而实例代码块则在每次新建对象时都会执行一次。因此我们常常将只需要执行一次的代码放到静态代码块中，以优化程序性能。

总结一下静态代码块的注意事项：

1. 可以在静态代码块中对静态常量进行初始化赋值。
2. 静态代码块优先于实例代码块和类构造函数被执行。
3. 静态代码块只被加载一次，提高程序性能。

1.2.5 static的优势

讲了这么多static的用法，那你知道使用static关键字的优势在哪里吗？static的优势可以体现在以下几个方面：

1. 不需要创建对象也可以使用相应的变量和方法。
2. 被static修饰的变量或方法只属于类本身，且被该类的所有实例对象所共享。
3. 被static修饰的部分只在类第一次加载时被初始化一次，提高程序性能。

1.2.6 static进阶

除了上述知识以外，还包括static静态导包，static修饰内部类以及static底层原理和单例模式等知识，感兴趣的同学可以自行前去学习理解，static修饰内部类我们会在后面将内部类时提到。

参考文章：

一个 static 还能难得住我?

static独特之处

面试官：兄弟，说说Java的static关键字吧

重新认识java（六） ---- java中的另类：static关键字

2. 接口 - Interface

2.1 接口的理解

在Java中有一个重要的概念，叫做`Interface`，翻译过来是`接口`的意思。

那么什么是接口？**对象通过它们公开的方法来定义与外界的交互行为，而这些方法就形成了与外界交互的接口。**

例如，每部手机都提供了一个USB充电口，用来匹配充电器的USB接头，以达到为手机充电的目的。那么这里的USB充电器就可以类比为一个对象，该对象实现了USB接口，可以用来给手机充电，这个充电的方法就是上述所说的公开的、用来与外界交互的方法。

同时我们的手机也不需要去关心插过来的充电器是什么品牌的，只需要这个充电器拥有USB插头即可，就好像市场上有很多种不同类型的充电器一样，它们都可以为手机提供充电功能。

总结来说，**接口规范了一个类应该做什么，而不指定它具体要如何做。**

2.2 接口的实现

现在让我们来按照上面所讲的例子来通过代码实现为手机充电的一个流程吧。

首先，我们需要定义一个USB接口，它是手机和充电器之间的**契约**，在**规范了一个充电器应该具备有充电功能**的同时，告诉手机的充电口当前插过来的东西**是否具备充电的能力。**

```
interface USB {
    void charge(); // 充电的能力
}
```

接下来，让我们根据USB定义好的规范来创建一个充电器的类Charger，这个类将实现USB接口，并重写其中的充电方法：

```
// 充电器
class Charger implements USB {
    @Override
    public void charge() {
        System.out.println("开始充电");
    }
}
```

其中`@Override`是对当前方法的一个注解，表示该方法是一个被重写的方法。

然后让我们定义一个手机的类，这个类对外提供一个充电的方法，即手机的充电口，并在方法内执行充电器的充电方法：

```
class Phone {
    // 该方法即为手机的充电口，可以接收具备USB接口的充电器来充电
    public void charge(USB charger) {
        charger.charge();
    }
}
```

最后，让我们在主函数中实现充电的流程吧。首先新建一个手机对象，再新建一个充电器的对象，最后再调用方法让充电器给手机完成充电即可。

```
public static void main(String[] args) {
    Phone phone = new Phone();
    USB charger = new Charger(); // 向上转型
    phone.charge(charger);
}
```

最终输出结果如下：

```
开始充电
```

> 向上转型与向下转型：
>
> 向上转型很好理解，即子类转型成自己的父类，相同方法名保留子类的内容，但子类特有的属性和方法不可再被使用。
>
> 向下转型的例子比较复杂，假如商店里提供了很多不同品牌的手机充电器，使用`ArrayList<USB> charges = new ArrayList<USB>()`来存储充电器的对象，那么当我们新加一个具体的实现对象时，会自动实现向上转型，从而丢失子类的一些方法（例如小米原装充电器在识别出是小米手机时，可以在手机显示快充的图标，但向上转型之后的小米充电器将不再具备识别小米手机的能力了），此时我们需要再向下转型一次才能恢复原来子类中具有的独特方法。
>
> 因此向下转型需要有严格的前提条件，即**只有引用子类对象的父类引用才能被向下转型为子类对象。**
>
> 参考文章：Java 向下转型的意义
>
> 聊聊java的向上转型与向下转型

2.3 接口与父类的不同

2.3.1 从设计层面来讲

看了上面接口使用的例子后，你可能会觉得接口的设计有些多余，我直接使用类的继承不就可以了吗？

但实际上，接口为我们的开发定义了一套规范，**让类能够按照接口制定好的规则去实现相应的方法**，也就是说**接口更加注重对行为的描述，而继承更多的是在强调一种从属关系。**

举个例子，鸟和飞机都具备飞行的能力，那么我们可以飞行的行为抽象成一个接口，让鸟和飞机的类都去实现该接口。但是很明显，鸟和飞机不属于一个物种，因此不能使用继承的关系。但是大雁和老鹰它们就都属于鸟类，此时可以就可以让大雁和老鹰对应的类都继承自Bird类。

同样的，上面的充电器能够为手机充电强调的是**充电器本身所具备的一种能力，即USB**，而手机刚好需要具备这种能力的对象来给自己充电。但具备USB接口的却又不只是充电器，还可以是其他对象，这些对象虽然不是同一类东西，却拥有相同的方法。

总结来说，接口就是把具有相同功能，但是本身却没有任何关系的类的功能抽象出来，然后让我们自定义具体的实现。

> 接口就是一份契约，由类实现契约。契约中我编写了某些大的方针与前提，而签了约的类可以具体问题具体分析来实现自己的功能

在实际的开发中，架构师通常会先使用接口来定好开发的规范，然后工程师们会根据规范在类中具体实现接口中的方法，以实现团队间高效的合作开发。

2.3.2 从语法角度来讲

接口与类主要有以下几点区别：

1. **接口不能被实例化**，一般我们都采用向上转型的方法来创建接口的实现类（当然也可以直接实例化接口的实现类）。
2. **接口中的所有方法必须被其实现类全部实现。**
3. **一个类可以实现多个接口，但只能继承自一个父类。**
4. **接口中所有方法默认是公开的、抽象的、不可被默认实现的**，即默认的前缀修饰符为`public abstract`。同时，其中的**所有成员变量都是公开的、静态的、不可变的**，即默认的前缀修饰符为`public static final`。

前两点在上面的例子中大家已经见到了。接下来我们讲一下剩下两点内容。

2.4 接口的注意事项

2.4.1 静态、私有、默认方法

- **接口中可以声明静态方法**，同时必须要在接口中实现静态方法，我们可以通过`接口名`+`.`+`静态方法名`的方式来调用接口中的静态方法，但实际上这样设计的意义不大，我们应该将静态方法放入相应的类中。
- **接口中也可以声明私有方法**，一样的私有方法必须要在接口中被实现，但是私有方法只能在接口内部使用，所以它通常是作为接口中其他方法的辅助方法，作用也不大。
- **接口中可以声明默认方法**，默认方法也必须要在接口中实现，默认方法有一个重要的用处就是"**接口演化**"，即当我们要为接口新加入一个默认方法时，不需要再为其每个实现类重写该方法。

  假如我们在USB接口中新加一个彩蛋的方法，当用户输入密码时会提示这是一个彩蛋：

  ```
  interface USB {
      void charge(); // 充电的能力

      default void surprise(int secret) {
          if (secret == 888)
              System.out.println("这是一个彩蛋");
      }
  }
  ```

  这样的话我们就不需要再挨个地去为每个USB实现类重写该方法了，可以直接调用相应的方法：

  ```
  charger.surprise(888);
  ```

2.4.2 实现多个接口

从设计层面来讲，**继承是一对多的关系**，即一个父类可以被多个子类继承，但是一个子类只能继承一个父类，也就是说一个子类只能有一种从属关系，就好比现实中一个父亲可以多个亲生孩子，而这些孩子却只能有这一个亲生父亲一样。但是接口就不一样了，**接口可以是多对多的关系**，即一个类可以实现多个接口，接口也可以被多个类实现，就好比上面说的充电器，它不仅拥有USB接口，还可以拥有自己的充电器品牌，例如是小米充电器，同样的，USB接口也不仅仅是充电器拥有的，小米生产的商品也不仅仅只有充电器一样。

2.4.2.1 多接口的简单使用

这里我们就以上面的充电器为例来说明多接口实现情况。这里我们再新定义一个接口叫做XiaoMi，当我们使用小米充电器给小米手机充电时会显示快充图标：

```
interface XiaoMi {
    void fastCharge(); // 快充
}
```

接下来，让我们制造一款小米充电器，即Charger类同时实现了USB和XiaoMi两个接口，并重写fastCharge方法：

```
// 充电器
class Charger implements USB, XiaoMi {
    @Override
    public void charge() {
        System.out.println("开始充电");
    }

    @Override
    public void fastCharge() {
        System.out.println("小米快充");
    }
}
```

最后，我们只需要在给手机充电的方法里判断一下当前传进来的充电器是否小米牌子的即可，这里我们使用`instanceof`关键字来判断一个类是否是另一个类的子类，如果是的话，我们将该类强转为XiaoMi类，并调用其中的快充方法：

```
class Phone {
    public void charge(USB charger) {
        if (charger instanceof XiaoMi) {
            ((XiaoMi) charger).fastCharge();
        }
    }
}
```

2.4.2.2 多接口实现带来的问题

从语法角度来讲，java中假如能够允许多继承，那么在子类中调用父类同名方法时就会产生冲突，而接口的出现就很好的解决了这个问题，当实现类继承的接口中出现了一模一样的方法时，编译器会强制要求在实现类重写覆盖该方法，并以实现类中重写的方法作为最终的方法调用。

假如我们在XiaoMi接口中也加一个方法charge()，那么最终还是以实现类Charger中的charge方法为标准调用：

```
interface XiaoMi {
    void charge();
    void fastCharge(); // 快充
}
```

这里我们可以借此来思考一下为什么java中不能有多继承？假如USB和XiaoMi分别是两个父类，并在各自的类中默认实现了charge()方法，那么在通过子类对象调用父类该方法时编译器将不知道最终应该调用哪个方法了：（注意，**下面这个示例代码不能被运行成功**）

```
class USB {
    void charge(){
        System.out.println("USB充电");
    }
}

class XiaoMi {
    void charge(){
        System.out,println("小米充电");
    }
    void fastCharge() { }
}
```

```
// 充电器
class Charger extends USB, XiaoMi {
}

public static void main(String[] args) {
    new Charger().charge(); // 此时编译器不知道该输出什么
}
```

那么爱思考的你此时可能已经发现出一个问题了，上面我们提到了接口中也可以默认实现方法，那么假如接口中有相同的默认方法时，编译器会强制要求实现类重写该方法，你可以在方法中通过`接口名`+`.`+`super`+`.`+`方法名`调用父类的方法：

```
interface USB {
    void charge(); // 充电的能力

    default void surprise(int secret) {
        if (secret == 888)
            System.out.println("这是一个彩蛋");
    }
}

interface XiaoMi {
    void charge();
    void fastCharge(); // 快充

    default void surprise(int secret) {
        System.out.println("XiaoMi彩蛋");
    }
}

// 充电器
class Charger implements USB, XiaoMi {
    @Override
    public void charge() {
        System.out.println("开始充电");
    }

    @Override
    public void surprise(int secret) {
        XiaoMi.super.surprise(secret);
    }

    @Override
    public void fastCharge() {
        System.out.println("小米快充");
    }
}
```

2.5 接口与回调

回调（callback）是一种常见的程序设计模式。我们可以通过回调来指定某个特定事件发生时需要采取的动作。

// 回调与普通参数不同的时，我们其实是将一段代码的实现传入函数中，并且函数不用去关心那段那个方法的具体实现，只负责调用即可。

举个通俗易懂的例子，我们现在希望给充电器加上一种功能，当它为手机充电点后，会通知手机停止接收任何电量。接下来我们**只需要三步**就可以实现接口回调的简单应用。

首先由于USB接口需要监听到何时手机电量能够充满，所以**USB的实现类需要持有Phone类的引用才行**，因此我们对USB接口作出下面的修改，对charge方法新加一个参数Phone：

```
interface USB {
    void charge(Phone phone); // 充电
}
```

其次，我们需要在Phone类中修改charge方法中对USB对象的调用，将自身作为参数传递进去；除此之外，我们还需要一个方法stop()，当USB接口检测到电量已满时，**会调用Phone的stop方法让手机停止继续接收电量**：

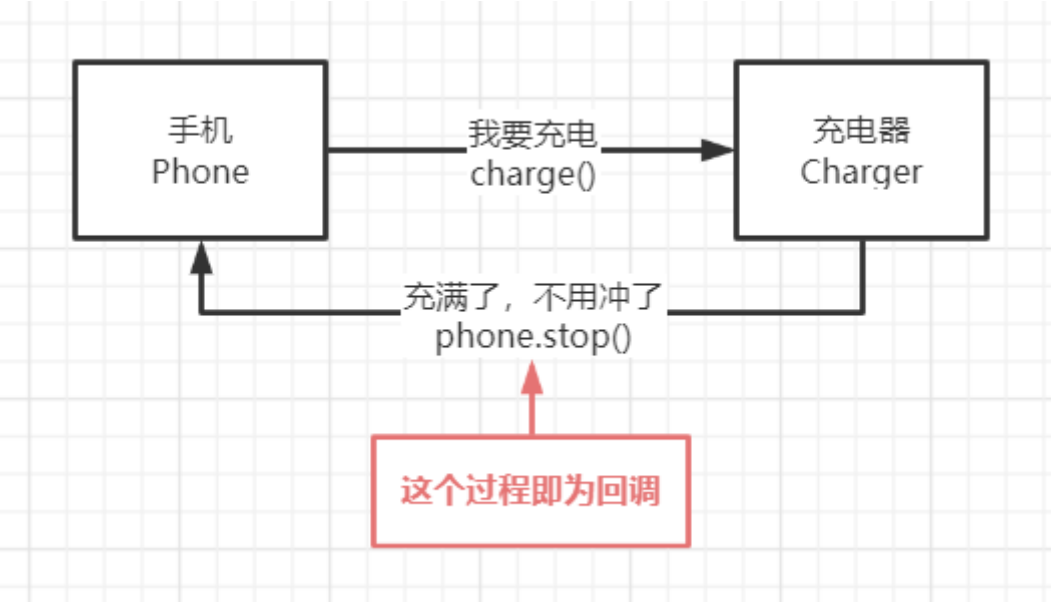
```
class Phone {
    public void charge(USB charger) {
        charger.charge(this); // 开始充电
    }

    public void stop() {
        System.out.println("停止接收电量");
    }
}
```

最后，我们重写Charger类中的charge方法，**当电量充满时告诉手机要停止接收电量了**：

```
// 充电器
class Charger implements USB {
    @Override
    public void charge(Phone phone) {
        System.out.println("开始充电");
        System.out.println("正在充电中");
        System.out.println("电量已经充满");
        phone.stop(); // 通知手机停止接收电量
    }
}
```

整个过程**通俗点来讲**就是：手机先使用得到的充电器进行充电，然后充电器在给手机充满电后通知手机不要再继续充了。

具体点来讲就是：Phone类先调用Charger类的charge()方法，然后在特定的时间（即充完电后）Charger再调用Phone类中的stop()方法，回调主要就指的是Charger往回调用的过程，被回调的函数stop()也被称为**回调函数**。



2.6 简洁的lambda表达式

2.6.1 为什么需要lambda表达式

在我们编写函数的时候应该都有过想将一段代码块传入一个函数中使用的想法，就比如上述的回调中，我们实际上是将`stop()`方法传入了充电器的`charge()`方法中，并在最后执行了`stop()`函数里的代码的，这种将一个函数作为参数传入方法的编程方式被称为**函数式编程**。

那么传统的函数式编程其实是都是通过对象来实现的，要想传入一段代码，我们首先要声明一个类，然后把想传入的代码块放入该类的某个方法中，最后再新建该类的一个对象作为参数传入方法中去。

这显然是一个非常繁琐的过程，那么为了解决这种麻烦，Java引进了lambda表达式来帮助我们简化函数式编程的步骤，只需一步即可实现相同的功能，大大减少了多余代码的编写。

2.6.2 lambda的使用

```
interface USB {
    void charge(Phone phone); // 充电
}
```

首先让我们可以看到USB接口中只有一个抽象方法charge()，charge方法需要传入一个参数Phone，在Charger类中我们对charge方法的实现是通知充电器停止充电的语句，分析完后我们就可以使用下面这行lambda表达式代替一个Charger类的定义了：

```
USB charger = (Phone phone) -> { phone.stop() };
```

其中小括号内代表的是该接口唯一抽象方法的所有参数，可以有多个，右边被大括号包裹起来的部分即该抽象方法的具体实现内容，参数和代码块通过`->`来连接，这就是一个简单的lambda表达式了。

如果只有一行代码，可以将右边的大括号省略：

```
USB charger = (Phone phone) -> phone.stop();
```

如果该抽象方法只有一个参数，且这个参数的类型可以被编译器自动推导出来的时候，可以省略小括号：

```
USB charger = phone -> phone.stop();
```

如果右边的代码块中只涉及参数的调用，则可以进一步简写为下面这样：

```
USB charger = Phone::stop;
```

这就是我们最终简化版的代码了，表示通过lambda表达式实现了USB接口，并在重写的charge()方法中调用参数Phone的stop()方法。

那么这里我们就可以新建一个Phone的对象，并传入该接口的实现即可：

```
Phone phone = new Phone();
phone.charge(Phone::stop);
```

这样的话，充电器Charger类这个中转站就被我们使用lambda彻底简化掉了。

关于lambda表达式的用法还有很多，这里就不详细介绍了，感兴趣的同学可以下去自行了解学习。

参考文章：

如何理解接口-Java系列

Java继承与实现的区别与联系

Java回调机制解读

「Java8系列」神秘的Lambda

Java Lambda表达式 实现原理分析

3. 抽象类

3.1 抽象类的理解

在了解抽象类之前，我们先来了解一下**抽象方法。**抽象方法就是使用`abstract`关键字修饰的方法，它是一种特殊的方法：**它只有声明，而没有具体的实现。**下面就是一个简单的抽象方法的定义：

```
abstract void function();
```

那么什么是抽象类呢？为什么要在介绍抽象类之前先介绍什么是抽象方法？

在《Java编程思想》中，作者将抽象类定义成了"包含抽象方法的类"，即**拥有抽象方法的类就叫做抽象类。**那为什么要这么说呢？原因有两点：

1. 只有抽象类中才能含有抽象方法，也就是说，当我们在一个普通的方法前面加上`abstract`关键字后，其对应的类也要变成抽象类，同样得加上`abstract`关键字。
2. 如果一个抽象类中不包含任何的抽象方法，即它里面实现的都是非抽象方法的话，那么这时将该类被设计为抽象类将会毫无意义。

除了上面的定义，我觉得抽象类就像是**类和接口的混合体**，为什么这么说呢？首先它和类一样表示的是从属关系，只能被单继承，并且在抽象类中可以直接实现普通方法的内容；同时它也和接口类似，抽象类中的所有抽象方法都必须被其子类实现。

3.2 抽象类的实现

这里我们以电池为例来说明抽象类，生活中的电池有很多种不同的型号，有1号电池（也叫做D型电池）被用于热水器、手电筒等，也有7号电池（也叫AAA型电池）被用于遥控器，鼠标等。

这里不管是1号电池还是七号电池，都属于电池这一类物品，那么我们就可以依据此来新建一个抽象类Battery，并实现其两个子类D_Battery和AAA_Battery分别表示1号电池和7号电池，并在其中提供一个抽象方法获取当前电池的名称：

```
abstract class Battery {
    abstract String getName();
}

class D_Battery extends Battery {
    @Override
    String getName() {
        return "1号电池";
    }
}

class AAA_Battery extends Battery {
    @Override
    String getName() {
        return "7号电池";
    }
}
```

然后在main方法中新建一个1号电池的对象，并打印其名字：

```
public static void main(String[] args) {
    Battery battery = new D_Battery();
    System.out.println(battery.getName());
}
```

3.3 抽象类的不同

3.3.1 抽象类与普通类的不同

1. 抽象方法必须声明为`public`或者`protected`，如果未加任何修饰符则默认为public类型。
2. 抽象类不能通过new关键字直接实例化为一个对象，因为其内的抽象方法未被具体实现。
3. 抽象类的普通子类必须实现其中的所有抽象方法，如果子类也为抽象类则可以选择不实现抽象方法。

那么假如当我们将两个联系密切的类提取出一个父类时，应该选择抽象类还是普通类呢？这个答案显然是不唯一的，**如果你不想在提取出的公共类中具体实现某个方法，而是想让其子类去具体实现的话**，那么抽象类毫无疑问是你的最佳选择。同时，抽象类的使用也提升了对子类代码的约束性，使开发更加规范。

3.3.2 抽象类与接口的不同

3.3.2.1 从设计层面来讲

抽象类是对一类事物的抽象，而接口是对共同行为的类似。我们依然拿电池和充电器为例来说明。1号电池和7号电池都属于电池，但它们的名字、电池容量等都不相同，需要在具体的子类中去实现，因此可以将它们共有的方法、属性（包括行为）等进行抽象，形成一个新的抽象类；但是电池和充电器却不是同一类物质，但它们俩却有着共同的行为关系，就是可以为其他设备供电，那么此时我们就可以将供电这一行为提取出来形成一个接口。

3.3.2.2 从语法角度来讲

1. 抽象类中可以为非抽象方法提供实现内容，而接口中的所有方法全都是`public abstract`的，不能在接口中被实现，只能在其实现类中具体实现。
2. 抽象类中的成员变量类型没有限制，而接口中的成员变量类型全都是`public static final`的。
3. 一个类只能继承一个抽象类，却能实现多个接口。

3.3.2.3 接口与抽象类的选择

- 抽象类适合被多个具有相同联系的类所继承，并且在抽象类中你可以实现非抽象方法以及非静态、非final变量，这些都是接口不能做到的。
- 接口适合被多个不相关的类所实现，并且如果你只想指定类的特定行为，但不关心具体如何被实现，或者想要利用一个类可以实现多个接口的优势的话，接口是较好的选择。

3.4 抽象类的注意事项

3.4.1 抽象类可以继承父类以及实现多个接口

这里我们仍然以上面的电池为例来说明，新建一个父类Goods表示商品，新加一个接口Charge，其中提供一个方法charge()，这里我们让抽象类Battery继承自Goods并实现Charge接口，表示电池属于一种商品，并且拥有供电能力：

```
class Goods {

}

interface Charge {
    void charge();
}

abstract class Battery extends Goods implements Charge{
    abstract String getName();
}
```

其中抽象类可以选择不实现接口中的方法，那么此时该抽象类的所有子类就必须去实现接口中的方法。如果抽象类选择实现了该方法，则其子类可以不用重写该方法了。

3.4.2 抽象类有构造方法

虽然抽象类不能被实例化，但作为一个可被继承的父类，子类在实例化时会优先调用父类的构造方法：

```
class Goods {
    public Goods() {
        System.out.println("Goods");
    }
}

abstract class Battery extends Goods{
    public Battery() {
        System.out.println("Battery");
    }
}

class D_Battery extends Battery {
    public D_Battery() {
        System.out.println("D_Battery");
```

```
    }
}
```

然后在main函数中创建一个1号电池对象，并允许观察结果：

```
public static void main(String[] args) {
    Goods battery = new D_Battery();
}

// 输出结果
Goods
Battery
D_Battery
```

可以从结果发现，当我们调用子类的构造方法时，会先依次调用其父类的构造方法，哪怕是抽象类也不例外。

3.4.3 在抽象类的构造方法中调用抽象方法

既然抽象类拥有构造方法，那么假如我们在抽象类的构造方法调用其抽象方法会输出什么结果呢？让我们来动手试验一下。在抽象类Battery的构造方法中我们将抽象方法getName的返回值打印出来，在其子类D_Battery中定义一个成员变量

```
abstract class Battery {
    abstract String getName();

    public Battery() {
        System.out.println(getName());
    }
}

class D_Battery extends Battery {
    String name = "1号电池";
    @Override
    String getName() {
        return name;
    }
}
```

最后我们在main函数中新建一个D_Battery对象，并观察控制台输出结果：

```
public static void main(String[] args) {
    Battery battery = new D_Battery();
}

// 输出结果
null
```

可以发现它并没有按照我们想的那样输出"1号电池"，而是输出了String变量的默认值null。让我们来分析一下原因，当我们在新建D_Battery对象时，会先访问其父类的构造方法`Battery()`，在父类的构造方法中调用了其抽象方法，但是此时该抽象方法在父类中并未实现，因此**程序会先找到子类中对该方法的实现**，在子类的getName方法中我们返回的是一个成员变量name的值，但此时程序只加载到了父类的构造方法，因此**子类中的该成员变量在此时仍未被初始化**，即为默认值null。

3.5 抽象类实践 - 减少if-else和switch-case

在实际开发过程中，可能会有小伙伴困扰于过多的if-else嵌套，也有人相使用switch-case来让多重判断看上去尽量简洁一些，那么今天我们就利用抽象类的知识来减少这些嵌套判断语句吧。

这里我们先假设一个应用场景，当我们想出门旅游时会有很多交通工具的选择，选择不同的交通工具最终到达目的地的时间也不相同，如果让你用程序将选择不同交通工具到达目的所需时间写出来，你会怎么写呢？这里我们先使用普通的类来实现。首先我们定义一个交通工具类，出于示例考虑，我们直接让它的两个属性对外开放：

```
// 交通工具
class Vehicle {
    public String name; // 工具名
    public int time; // 到达目的地所需的时间

    public Vehicle(String name, int time) {
        this.name = name;
        this.time = time;
    }
}
```

那么第一版，我们在主程序通过if-else语句来实现该功能：

```
public static void main(String[] args) {
        Vehicle bike = new Vehicle("自行车", 100);
        Vehicle car = new Vehicle("汽车", 50);
        Scanner scanner = new Scanner(System.in);
        System.out.println("输入你的交通工具：");
        String name = scanner.next();
        if (bike.name.equals(name)) {
            System.out.println(bike.time);
        } else if (car.name.endsWith(name)) {
            System.out.println(car.time);
        } else {
            System.out.println("无法找到该交通工具");
        }
    }
```

第二版，我们使用switch-else语句来完成该功能，注意case后必须为一个常量：

```
public static void main(String[] args) {
        Vehicle bike = new Vehicle("自行车", 100);
        Vehicle car = new Vehicle("汽车", 50);
        Scanner scanner = new Scanner(System.in);
        System.out.println("输入你的交通工具：");
        switch (scanner.next()) {
            case "自行车":
                System.out.println(bike.time);
                break;
            case "汽车":
                System.out.println(car.time);
                break;
            default:
                System.out.println("无法找到该交通工具");
        }
```

```
    }
```

第三版，我们使用面向对象的思维来解决问题，将Vehicle类变为抽象类，并定义两个抽象方法，然后分别定义交通工具的具体实现类：

```
// 交通工具
abstract class Vehicle {
    abstract String getName(); // 获取工具名
    abstract int getTime(); // 获取到达目的地的时间
}

class Bike extends Vehicle {
    @Override
    String getName() {
        return "自行车";
    }

    @Override
    int getTime() {
        return 100;
    }
}

class Car extends Vehicle{
    @Override
    String getName() {
        return "汽车";
    }

    @Override
    int getTime() {
        return 50;
    }
}
```

在主函数中，只需要根据工具名称获取相应的时间即可：

```
public static void main(String[] args) {
        Vehicle[] vehicles = { new Bike(), new Car() };
        Scanner scanner = new Scanner(System.in);
        System.out.println("输入你的交通工具：");
        String name = scanner.next();
        for (Vehicle vehicle: vehicles) {
            if (vehicle.getName().equals(name))
                System.out.println(vehicle.getTime());
        }
    }
```

此时假如你想添加一个交通工具时，只需要添加一个新的类，并将其实例对象添加至初始数组中即可。

参考文章：

深入理解Java的接口和抽象类

Abstract Methods and Classes

【Medium翻译】Java抽象类有什么用?

Java 抽象类你必须了解的一些知识

4. 内部类

内部类即**定义在一个类中的类**，与内部类相对的有一个概念叫外围类，即包裹着内部类的类。那我们为啥需要用到内部类呢？原因有两点：

1. 内部类可以对同一个包下的其他类隐藏，其他同一个包下的其他类无法访问到另一个类的内部类。
2. 内部类里的方法可以直接访问到外围类的私有成员变量。

> 在lambda表达式出现之前，内部类可以简洁地实现回调，事实上lambda表达式的底层原理就是利用了静态内部类来实现的，有兴趣的同学可以自行了解。

4.1 成员内部类

这里我们先简单地定义一个类ClassA，在其内包含着一个内部类ClassB：

```
class ClassA {
    private int num;

    class ClassB {
        void test() {
            System.out.println(num);
        }
    }
}
```

可以发现，在内部类ClassB中的方法是可以直接调用外部类的私有成员变量的。

除此之外，内部类还可访问外围类中的方法，并且内部类可以独立的实现父类和继承接口。

正常情况下，内部类访问外部类的方法应该是通过`ClassA.this.num`访问，但为什么这里的内部类只写一个num即可了呢？

```
class ClassA$ClassB  {
    ClassA$ClassB(ClassA this$0) {
        this.this$0 = this$0;
    }
    public void test() {
        System.out.println(this.this$0.num);
    }
}
```

通过上面代码我们可以发现，在内部类的构造方法中外部类的引用被传递进来，并在每一次使用外部类成员变量或方法时自动将该引用加上。

4.2 局部内部类

局部内部类是定义在外围类方法中的内部类，下面的ClassB就是一个局部内部类：

```
class ClassA {
    private int num;

    public void test() {
        class ClassB {

        }
    }
}
```

局部内部类有两个特点：

1. 它对外部世界完全隐藏，出来test()方法以外，ClassA类中的其他任何方法都无法访问到该类。
2. 局部类不仅可以访问到外部类的变量，可以访问局部变量，但这些局部变量一旦被访问后必须不可变。

> 这是因为局部类会在访问到局部变量时，在其内部为该变量建立相同的、由final修饰的变量。

4.3 静态内部类

在静态内部类里，不会生成外围类对象的引用(防止内存泄漏)，因此也无法访问外围内的非静态对象。

当一个内部类不需要访问到外围类的非静态对象时，应该使用静态内部类。

```
class ClassA {
    private static int num;

    static class ClassB {
        void test() {
            System.out.println(num);
        }
    }
}
```

4.4 匿名内部类

当我们使用局部内部类时还可以再进一步，如果只想创建这个类的一个对象的话，甚至可以不需要为该内部类指定特定的名字，这样的类就被叫做匿名内部类。

我们拿上面USB的接口为例来说明如何使用匿名内部类：

```
USB usb = new USB() {
    @Override
    public void charge() {
        // 在此处实现方法
    }
};
```

这段代码就相当于新建了一个类的对象，且这个类实现了USB接口和方法。

新建匿名内部类需要注意以下几点：

1. 匿名内部类中不能存在任何静态成员或方法
2. 匿名内部类是没有构造方法的，因为它没有类名

3. 与局部内部相同匿名内部类也可以引用局部变量。此变量也必须声明为 final（局部变量与匿名内部类的生命周期不同）

参考文章：

搞懂 JAVA 内部类

死磕Java内部类