- 为什么使用内部类?
  - 使用内部类最吸引人的原因是：每个内部类都能独立地继承一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响
  - 使用内部类最大的优点就在于它能够非常好的解决多重继承的问题,使用内部类还能够为我们带来如下特性:
    - 内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独。
    - 在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。
    - 创建内部类对象的时刻并不依赖于外围类对象的创建。
    - 内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。
    - 内部类提供了更好的封装，除了该外围类，其他类都不能访问。
- 内部类分类:
  - 成员内部类:
  - ```
    public class Outer{
        private int age = 99;
        String name = "Coco";
        public class Inner{
            String name = "Jayden";
            public void show(){
                System.out.println(Outer.this.name);
                System.out.println(name);
                System.out.println(age);
            }
        }
        public Inner getInnerClass(){
            return new Inner();
        }
        public static void main(String[] args){
            Outer o = new Outer();
            Inner in = o.new Inner();
            in.show();
        }
    }

    /*
    1.Inner 类定义在 Outer 类的内部，相当于 Outer 类的一个成员变量的位置，Inner 类可以使用任意访问控制符，如 public 、 protected 、 private 等

    2.Inner 类中定义的 show() 方法可以直接访问 Outer 类中的数据，而不受访问控制符的影响，如直接访问 Outer 类中的私有属性age
    ```
    ```

```
3.定义了成员内部类后，必须使用外部类对象来创建内部类对象，而不能直接去 new 一个内部类对象，即：内部类 对象名 = 外部类对象.new 内部类( );

4.编译上面的程序后，会发现产生了两个 .class 文件：
Outer.class,Outer$Inner.class{}

5.成员内部类中不能存在任何 static 的变量和方法,可以定义常量：

(1).因为非静态内部类是要依赖于外部类的实例,而静态变量和方法是不依赖于对象的,仅与类相关,简而言之:在加载静态域时,根本没有外部类,所在在非静态内部类中不能定义静态域或方法,编译不通过;非静态内部类的作用域是实例级别
(2).常量是在编译器就确定的,放到所谓的常量池了


★★友情提示：
1.外部类是不能直接使用内部类的成员和方法的，可先创建内部类的对象，然后通过内部类的对象来访问其成员变量和方法；

2.如果外部类和内部类具有相同的成员变量或方法，内部类默认访问自己的成员变量或方法，如果要访问外部类的成员变量，可以使用 this 关键字,如:Outer.this.name

*/
```

- 静态内部类: 是 static 修饰的内部类,
  - 静态内部类不能直接访问外部类的非静态成员，但可以通过 new 外部类().成员 的方式访问
  - 如果外部类的静态成员与内部类的成员名称相同，可通过“类名.静态成员”访问外部类的静态成员；
  - 如果外部类的静态成员与内部类的成员名称不相同，则可通过“成员名”直接调用外部类的静态成员
  - 创建静态内部类的对象时，不需要外部类的对象，可以直接创建 内部类 对象名 = new 内部类();

```
public class Outer{
    private int age = 99;
    static String name = "Coco";
    public static class Inner{
        String name = "Jayden";
        public void show(){
            System.out.println(Outer.name);
            System.out.println(name);
        }
    }
    public static void main(String[] args){
        Inner i = new Inner();
        i.show();
    }
}
```

- 方法内部类:访问仅限于方法内或者该作用域内
  - 局部内部类就像是方法里面的一个局部变量一样，是不能有 public、protected、private 以及 static 修饰符的
  - 只能访问方法中定义的 final 类型的局部变量,因为:
    - 当方法被调用运行完毕之后，局部变量就已消亡了。但内部类对象可能还存在, 直到没有被引用时才会消亡。此时就会出现一种情况，就是内部类要访问一个不存在的局部变量;
    - 使用final修饰符不仅会保持对象的引用不会改变,而且编译器还会持续维护这个对象在回调方法中的生命周期.
    - 局部内部类并不是直接调用方法传进来的参数，而是内部类将传进来的参数通过自己的构造器备份到了自己的内部, 自己内部的方法调用的实际是自己的属性而不是外部类方法的参数;

```
/*
使用的形参为何要为 final???
在内部类中的属性和外部方法的参数两者从外表上看是同一个东西，但实际上却不是，所以他们两者是可以任意变化的，也就是说在内部类中我对属性的改变并不会影响到外部的形参，而然这从程序员的角度来看这是不可行的，毕竟站在程序的角度来看这两个根本就是同一个，如果内部类该变了，而外部方法的形参却没有改变这是难以理解和不可接受的，所以为了保持参数的一致性，就规定使用 final 来避免形参的不改变
*/
public class Outer{
    public void Show(){
        final int a = 25;
        int b = 13;
        class Inner{
            int c = 2;
            public void print(){
                System.out.println("访问外部类:" + a);
                System.out.println("访问内部类:" + c);
            }
        }
    Inner i = new Inner();
    i.print();
    }
    public static void main(String[] args){
        Outer o = new Outer();
        o.show();
    }
}
```

- 注意:在JDK8版本之中,方法内部类中调用方法中的局部变量,可以不需要修饰为 final,匿名内部类也是一样的，主要是JDK8之后增加了 Effectively final 功能. 反编译jdk8编译之后的class文件,发现内部类引用外部的局部变量都是 final 修饰的
- 匿名内部类:
  - 匿名内部类是直接使用 new 来生成一个对象的引用;
  - 对于匿名内部类的使用它是存在一个缺陷的，就是它仅能被使用一次，创建匿名内部类时它会立即创建一个该类的实例，该类的定义会立即消失，所以匿名内部类是不能够被重复使用;
  - 使用匿名内部类时，我们必须是继承一个类或者实现一个接口，但是两者不可兼得，同时也只能继承一个类或者实现一个接口;
  - 匿名内部类中是不能定义构造函数的,匿名内部类中不能存在任何的静态成员变量和静态方法;
  - 匿名内部类中不能存在任何的静态成员变量和静态方法,匿名内部类不能是抽象的,它必须要实现继承的类或者实现的接口的所有抽象方法

- 匿名内部类初始化:使用构造代码块！利用构造代码块能够达到为匿名内部类创建一个构造器的效果

- ```
  public class OuterClass {
      public InnerClass getInnerClass(final int   num,String str2){
          return new InnerClass(){
              int number = num + 3;
              public int getNumber(){
                  return number;
              }
          };         /* 注意：分号不能省 */
      }
      public static void main(String[] args) {
          OuterClass out = new OuterClass();
          InnerClass inner = out.getInnerClass(2, "chenssy");
          System.out.println(inner.getNumber());
      }
  }
  interface InnerClass {
      int getNumber();
  }
  ```
  ```