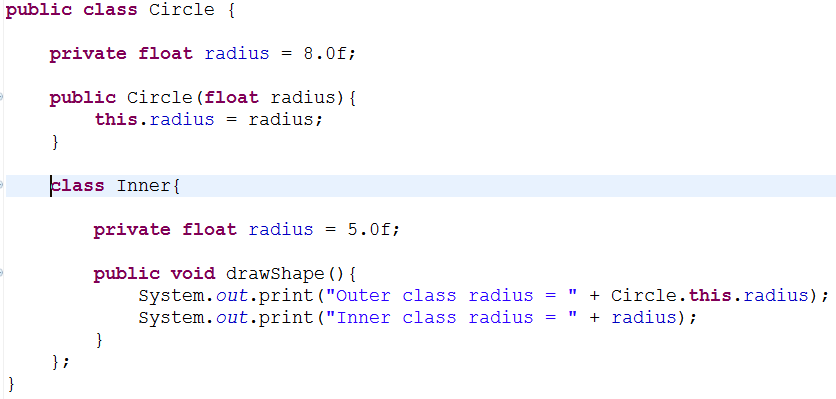
**内部类基础**



成员内部类是最普通的内部类，它的定义位于另一个类的内部。在上面的例子中，Circle称为外部类，成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员），不过需要注意的是，当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，及默认情况下访问的是成员内部类的成员，如果要访问外部类的同名成员，需要以下面的形式进行访问：

外部类.this.成员变量

外部类.this.成员方法

虽然成员内部类可以无条件地访问外部类的成员，而外部类想访问内部类就不是这么简单了，在外部类中如果要访问成员内部类的成员，必须先创建一个成员内部类的对象，在通过指向这个对象的引用来访问。

成员内部内是依附外部类而存在的，也就是说，如果要创建成员内部类的对象，前提是必须存在一个外部类的对象，创建成员内部类对象的一般方式如下：

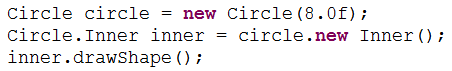
Outter outer = new Outter();

Outter.Inner inner = outer.new Inner(); // 必须通过Outter对象来创建

或者：

Outter.Inner innter1 = outer.getInnerInstance(); // 这种方式是在Outter的方法中new

在上面的例子中

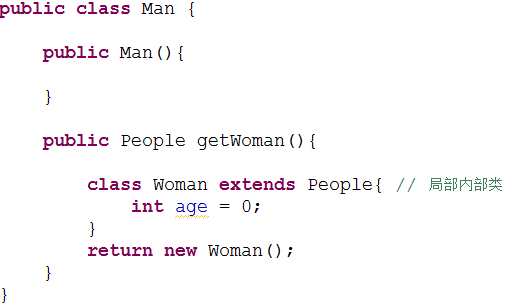




内部类可以拥有private、protected和public访问权限。比如上面的例子，如果inner用private修饰，则只能在外部类的内部访问，如果用public，则任何地方都能访问，如果用protected，则只能在同一个包下或者继承外部类的情况下访问；如果是默认访问权限，则只能在同一个包下访问。

**局部内部类**

局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问权限仅限于方法内或者该作用域内。



注意：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修饰的。（已验证）

**匿名内部类**

通常在各种设置监听器的时候会用到

同样的，匿名内部类也是不能有访问修饰符和static修饰符的

匿名内部类是位移一种没有构造器的类，正因为其没有构造器，所有匿名内部类的使用范围非常有限，大部分匿名内部类用于接口回调，匿名内部类在编译的时候由系统自动起名为Outer$1.class。一般来说，匿名内部类用于继承其他类或是实现几口，并不需要增加额外的方法。只是对继承方法的实现或重写。

**静态内部类**

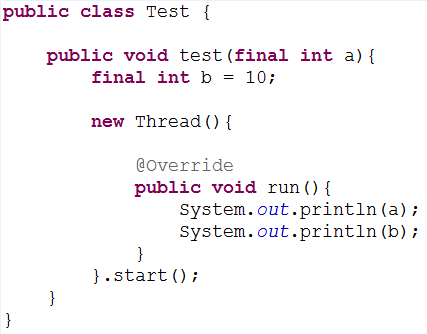
静态内部类也是定义在另一个类里面的类，只不过在类的前面多了一个关键字static，静态类是不需要依赖于外部类的，这点和类的静态成员属性有点类似，并且它不能使用外部类的非static成员变量或者方法，这点很好理解，因为在没有外部类对象的情况下，可以创建静态内部类的对象，如果允许访问外部类的非static成员就会产生矛盾，因此外部类的非static成员必须依附于具体的对象

注意：static关键字的作用是把类的成员变成类相关，而不是实例相关，即static修饰的成员属于整个类，而不属于单个对象。外部类的上一级程序单元是包，所以不可使用static修饰；而内部类的上一级程序单元是外部类，使用static修饰可以将内部类编程外部类相关，而不是外部类实例相关，因此，static关键字不可修饰外部类，但可以修饰内部类

**为什么成员内部类可以无条件访问外部类的成员？**

对于上面的例子，我们发现内部类生成的.class文件名字为，实际上编译器会默认为成员内部类添加一个指向外部类对象的引用，那么这个引用是如何赋初始值的呢？实际上，虽然我们定义的内部类构造器是无参构造器，编译器还是会默认添加一个参数，该参数的类型为指向外部类对象的一个引用，所以成员内部类中的指针便指向了外部类对象，因此可以在成员内部类随意访问外部类的成员。从这里也间接说明了成员内部类是依赖于外部类的，如果没有创建外部类的对象，则无法对内部内进行初始化赋值，也就无法创建成员内部类的对象了。

**为什么局部内部类和匿名内部类只能访问局部final变量？**





默认情况下，编译器会为匿名内部类和局部内部类起名为Outterx.class(x为正整数)

这段代码中，如果把变量a和b前面的任一个final去掉，这段代码都编译不过。（已验证）

当test方法执行完毕时，变量b的生命周期就结束了，而此时Thread对象的生命周期很可能还没有结束，那么在Thread的run方法中继续访问变量b就变成不可能了，但又要实现这样的效果，怎么办呢？java采用了复制的手段来解决这个问题。

通过反编译可以发现，有一条指定是将操作数10压栈，表示使用的是一个本地局部变量。这个过程在编译期间由编译器默认进行，如果这个变量的值在编译期间可以去顶，则编译器默认会在匿名内部类（局部内部类）的常量池中添加一个内容相等的字面量或直接将相应的字节码嵌入到执行字节码中，这样一来，匿名内部类使用的变量是另一个局部变量，只不过值和方法中局部变量的值相等，因此和方法中的局部变量完全独立开。

我们可以看到匿名内部类中的构造器有一个是指向外部类对象的引用，一个是int型变量。从上面的分析可以看出，在run方法中访问的变量b根本就不是test方法中的局部变量a，这样一来就解决了前面所说的生命周期不一致的问题，但是如果在run方法中改变了a的值，会出现什么情况呢？对，会造成数据不一致。为了解决这个问题，java编译器限定就限定必须将变量限制为final变量。