72 final 的三种用法是什么?

本课时我们主要讲解 final 的三种用法。

final 的作用

final 是 Java 中的一个关键字，简而言之，final 的作用意味着“**这是无法改变的**”。不过由于 final 关键字一共有三种用法，它可以用来修饰**变量**、**方法**或者**类**，而且在修饰不同的地方时，效果、含义和侧重点也会有所不同，所以我们需要把这三种情况分开介绍。

我们先来看一下 final 修饰变量的情况。

final 修饰变量

作用

关键字 final 修饰变量的作用是很明确的，那就是意味着这个变量**一旦被赋值就不能被修改了**，也就是说只能被赋值一次，直到天涯海角也不会“变心”。如果我们尝试对一个已经赋值过 final 的变量再次赋值，就会报编译错误。

我们来看下面这段代码示例：

```
/**
 * 描述：      final变量一旦被赋值就不能被修改
 */
public class FinalVarCantChange {
    public final int finalVar = 0;
    public static void main(String[] args) {
        FinalVarCantChange finalVarCantChange = new FinalVarCantChange();
//        finalVarCantChange.finalVar=9;     //编译错误，不允许修改final的成员变量
```

```
    }
}
```

在这个例子中，我们有一个 final 修饰的 int，这个变量叫作 finalVar，然后在 main 函数中，新建了这个类的实例，并且尝试去修改它的值，此时会报编译错误，所以这体现了 final 修饰变量的一个最主要的作用：一旦被赋值就不能被修改了。

目的

看完了它的作用之后，我们就来看一下使用 final 的目的，也就是为什么要对某个变量去加 final 关键字呢？主要有以下两点目的。

第一个目的是出于**设计角度**去考虑的，比如我们希望创建一个一旦被赋值就不能改变的量，那么就可以使用 final 关键字。比如声明常量的时候，通常都是带 final 的：

```
public static final int YEAR = 2021;
```

这个时候其实 YEAR 是固定写死的，所以我们为了防止它被修改，就给它加上了 final 关键字，这样可以让这个常量更加清晰，也更不容易出错。

第二个目的是从**线程安全的角度**去考虑的。**不可变**的对象天生就是线程安全的，所以不需要我们额外进行同步等处理，这些开销是没有的。如果 final 修饰的是**基本数据类型**，那么它自然就具备了不可变这个性质，所以自动保证了线程安全，这样的话，我们未来去使用它也就非常放心了。

这就是我们使用 final 去修饰变量的两个目的。

赋值时机

下面我们就来看一下被 final 修饰的变量的赋值时机，变量可以分为以下三种：

- 成员变量，类中的非 static 修饰的属性；
- 静态变量，类中的被 static 修饰的属性；
- 局部变量，方法中的变量。

这三种不同情况的变量，被 final 修饰后，其赋值时机也各不相同，我们逐个来看一下。

（1）成员变量

成员变量指的是一个类中的非 static 属性，对于这种成员变量而言，被 final 修饰后，它有三种赋值时机（或者叫作赋值途径）。

- 第一种是在声明变量的等号右边直接赋值，例如：

```
public class FinalFieldAssignment1 {
    private final int finalVar = 0;
}
```

在这个类中有 “private final int finalVar = 0”，这就是在声明变量的时候就已经赋值了。

- 第二种是在构造函数中赋值，例如：

```
class FinalFieldAssignment2 {
    private final int finalVar;
    public FinalFieldAssignment2() {
        finalVar = 0;
    }
}
```

在这个例子中，我们首先声明了变量，即 private final int finalVar，且没有把它赋值，然后在这个类的构造函数中对它进行赋值，这也是可以的。

- 第三种就是在类的构造代码块中赋值（不常用），例如：

```
class FinalFieldAssignment3 {
    private final int finalVar;
    {
        finalVar = 0;
    }
}
```

我们同样也声明了一个变量 private final int finalVar，且没有把它赋值，然后在下面的一个由大括号括起来的类的构造代码块中，对变量进行了赋值，这也是合理的赋值时机。

需要注意的是，这里讲了三种赋值时机，我们**必须从中挑一种来完成对 final 变量的赋值**。如果不是 final 的普通变量，当然可以不用在这三种情况下赋值，完全可以在其他的时机赋值；或者如果你不准备使用这个变量，那么自始至终不赋值甚至也是可以的。但是对于 final 修饰的成员变量而言，必须在三种情况中任选一种来进行赋值，而不能一种都不挑、完全不赋值，那是不行的，这是 final 语法所规定的。

空白 final

下面讲解一种概念：“**空白 final**”。如果我们声明了 final 变量之后，并没有立刻在等号右侧对它赋值，这种情况就被称为“**空白 final**”。这样做的好处在于增加了 final 变量的灵活性，比如可以在构造函数中根据不同的情况，对 final 变量进行不同的赋值，这样的话，被 final 修饰的变量就不会变得死板，同时又能保证在赋值后保持不变。我们用下面这个代码来说明：

```
/**
 * 描述：        空白final提供了灵活性
 */
public class BlankFinal {
    //空白final
    private final int a;
    //不传参则把a赋值为默认值0
    public BlankFinal() {
        this.a = 0;
    }
    //传参则把a赋值为传入的参数
    public BlankFinal(int a) {
        this.a = a;
    }
}
```

在这个代码中，我们有一个 private final 的 int 变量叫作 a，该类有两个构造函数，第一个构造函数是把 a 赋值为 0，第二个构造函数是把 a 赋值为传进来的参数，所以你调用不同的构造函数，就会有不同的赋值情况。这样一来，利用这个规则，我们就可以根据业务去给

final 变量设计更灵活的赋值逻辑。所以利用空白 final 的一大好处，就是可以让这个 final 变量的值并不是说非常死板，不是绝对固定的，而是可以根据情况进行灵活的赋值，只不过一旦赋值后，就不能再更改了。

（2）静态变量

静态变量是类中的 static 属性，它被 final 修饰后，只有两种赋值时机。

第一种同样是在声明变量的等号右边直接赋值，例如：

```
/**
 * 描述：        演示final的static类变量的赋值时机
 */
public class StaticFieldAssignment1 {
    private static final int a = 0;
}
```

第二种赋值时机就是它可以在一个静态的 static 初始代码块中赋值，这种用法不是很多，例如：

```
class StaticFieldAssignment2 {
    private static final int a;
    static {
        a = 0;
    }
}
```

在这个类中有一个变量 private static final int a，然后有一个 static，接着是大括号，这是静态初始代码块的语法，在这里面我们对 a 进行了赋值，这种赋值时机也是允许的。以上就是静态 final 变量的两种赋值时机。

需要注意的是，我们不能用普通的非静态初始代码块来给静态的 final 变量赋值。同样有一点比较特殊的是，这个 **static 的 final 变量不能在构造函数中进行赋值。**

（3）局部变量

局部变量指的是方法中的变量，如果你把它修饰为了 final，它的含义依然是**一旦赋值就不能改变。**

但是它的赋值时机和前两种变量是不一样的，因为它是在方法中定义的，所以它没有构造函数，也同样不存在初始代码块，所以对应的这两种赋值时机就都不存在了。实际上，对于 final 的局部变量而言，它是不限定具体赋值时机的，只要求我们**在使用之前必须对它进行赋值**即可。

这个要求和方法中的非 final 变量的要求也是一样的，对于方法中的一个非 final 修饰的普通变量而言，它其实也是要求在使用这个变量之前对它赋值。我们来看下面这个代码的例子：

```
/**
 * 描述：        本地变量的赋值时机：使用前赋值即可
 */
public class LocalVarAssignment1 {
    public void foo() {
        final int a = 0;//等号右边直接赋值
    }
}
class LocalVarAssignment2 {
    public void foo() {
        final int a;//这是允许的，因为a没有被使用
    }
}
class LocalVarAssignment3 {
    public void foo() {
        final int a;
        a = 0;//使用前赋值
        System.out.println(a);
    }
}
```

首先我们来看下第一个类，即 LocalVarAssignment1，然后在 foo() 方法中有一个 final 修饰的 int a，最后这里直接在等号右边赋值。

下面看第二个类，由于我们后期没有使用到这个 final 修饰的局部变量 a，所以这里实际上自始至终都没有对 a 进行赋值，即便它是 final 的，也可以对它不赋值，这种行为是语法所允许的。

第三种情况就是先创造出一个 final int a，并且不在等号右边对它进行赋值，然后在使用之前对 a 进行赋值，最后再使用它，这也是允许的。

总结一下，对于这种局部变量的 final 变量而言，它的赋值时机就是**要求在使用之前进行赋值**，否则使用一个未赋值的变量，自然会报错。

特殊用法：final 修饰参数

关键字 final 还可以用于修饰方法中的参数。在方法的参数列表中是可以把参数声明为 final 的，这意味着**我们没有办法在方法内部对这个参数进行修改**。例如：

```
/**
 * 描述：      final参数
 */
public class FinalPara {
    public void withFinal(final int a) {
        System.out.println(a);//可以读取final参数的值
//        a = 9; //编译错误，不允许修改final参数的值
    }
}
```

在这个代码中有一个 withFinal 方法，而且这个方法的入参 a 是被 final 修饰的。接下来，我们首先把入参的 a 打印出来，这是允许的，意味着我们可以读取到它的值；但是接下来我们假设想在方法中对这个 a 进行修改，比如改成 a = 9，这就会报编译错误，因此不允许修改 final 参数的值。

以上我们就把 final 修饰变量的情况都讲完了，其核心可以用一句话总结：**一旦被赋值就不能被修改了**。

final 修饰方法

下面来看一看 final 修饰方法的情况。选择用 final 修饰方法的原因之一是为了**提高效率**，因为在早期的 Java 版本中，会把 final 方法转为内嵌调用，可以消除方法调用的开销，以提高程序的运行效率。不过在后期的 Java 版本中，JVM 会对此自动进行优化，所以不需要我们程序员去使用 final 修饰方法来进行这些优化了，即便使用也不会带来性能上的提升。

目前我们使用 final 去修饰方法的唯一原因，就是想把这个方法锁定，意味着任何继承类都不能修改这个方法的含义，也就是说，被 final 修饰的方法**不可以被重写**，不能被 override。我们来举一个代码的例子：

```
/**
 * 描述：      final的方法不允许被重写
 */
public class FinalMethod {
    public void drink() {
    }
    public final void eat() {
    }
}
class SubClass extends FinalMethod {
    @Override
    public void drink() {
        //非final方法允许被重写
    }
//    public void eat() {}//编译错误，不允许重写final方法
//    public final SubClass() {} //编译错误，构造方法不允许被final修饰
}
```

在这个代码中一共有两个类，第一个是 FinalMethod，它里面有一个 drink 方法和 eat 方法，其中 eat 方法是被 final 修饰的；第二个类 SubClass 继承了前面的 FinalMethod 类。

然后我们去尝试对 drink 方法进行 Override，这当然是可以的，因为它是非 final 方法；接

着尝试对 eat 方法进行 Override，你会发现，在下面的子类中去重写这个 eat 方法是不行的，会报编译错误，因为不允许重写 final 方法。

同时这里还有一个注意点，在下方我们又写了一个 public final SubClass () {}，这是一个构造函数，这里也是编译不通过的，因为**构造方法不允许被 final 修饰。**

特例：final 的 private方法

这里有一个特例，那就是用 final 去修饰 private 方法。我们先来看看下面这个看起来可能不太符合规律的代码例子：

```
/**
 * 描述：      private方法隐式指定为final
 */
public class PrivateFinalMethod {
    private final void privateEat() {
    }
}
class SubClass2 extends PrivateFinalMethod {
    private final void privateEat() {//编译通过，但这并不是真正的重写
    }
}
```

在这个代码例子中，首先有个 PrivateFinalMethod 类，它有个 final 修饰的方法，但是注意这个方法是 private 的，接下来，下面的 SubClass2 extends 第一个 PrivateFinalMethod 类，也就是说继承了第一个类；然后子类中又写了一个 private final void privateEat() 方法，而且这个时候编译是通过的，也就是说，子类有一个方法名字叫 privateEat，而且是 final 修饰的。同样的，这个方法一模一样的出现在了父类中，那是不是说这个子类 SubClass2 成功的重写了父类的 privateEat 方法呢？是不是意味着我们之前讲的“被 final 修饰的方法，不可被重写”，这个结论是有问题的呢？

其实我们之前讲的结论依然是对的，但是类中的所有 private 方法都是隐式的指定为自动被 final 修饰的，我们额外的给它加上 final 关键字并不能起到任何效果。由于我们这个方法是 private 类型的，所以对于子类而言，根本就获取不到父类的这个方法，就更别说重写了。在上面这个代码例子中，其实**子类并没有真正意义上的去重写父类的 privateEat 方法**，子类和父类的这两个 privateEat 方法彼此之间是独立的，只是方法名碰巧一样而已。

为了证明这一点，我们尝试在子类的 privateEat 方法上加个 Override 注解，这个时候就会提示“Method does not override method from its superclass”，意思是“该方法没有重写父类的方法”，就证明了**这不是一次真正的重写**。

以上就把 final 修饰方法的情况讲解完了。

final 修饰类

下面我们再来看下 final 修饰类的情况，final 修饰类的含义很明确，就是这个类“**不可被继承**”。我们举个代码例子：

```
/**
 * 描述:       测试final class的效果
 */
public final class FinalClassDemo {
    //code
}
//class A extends FinalClassDemo {}//编译错误，无法继承final的类
```

有一个 final 修饰的类叫作 FinalClassDemo，然后尝试写 class A extends FinalClassDemo，结果会报编译错误，因为语法规定无法继承 final 类，那么我们给类加上 final 的目的是什么呢？如果我们这样设计，就代表不但我们自己不会继承这个类，也不允许其他人来继承，它就不可能有子类的出现，这在一定程度上可以**保证线程安全**。

比如非常经典的 String 类就是被 final 修饰的，所以我们自始至终也没有看到过哪个类是继承自 String 类的，这对于保证 String 的**不可变性**是很重要的，这一点我们会在第 74 讲展开讲解。

但这里有个注意点，假设我们给某个类加上了 final 关键字，这并不代表里面的成员变量自动被加上 final。事实上，这两者之间不存在相互影响的关系，也就是说，类是 final 的，不代表里面的属性就会自动加上 final。

不过我们也记得，final 修饰方法的含义就是这个方法不允许被重写，而现在如果给这个类都加了 final，那这个类连子类都不会有，就更不可能发生重写方法的情况。所以，其实在 final 的类里面，所有的方法，不论是 public、private 还是其他权限修饰符修饰的，都会**自动的、隐式的被指定为是 final 修饰的**。

如果必须使用 final 方法或类，请说明原因

这里有一个注意点，那就是如果我们真的要使用 final 类或者方法的话，需要注明原因。为什么呢？因为未来代码的维护者，他可能不是很理解为什么我们在这里使用了 final，因为使用后，对他来说是有影响的，比如用 final 修饰方法，那他就不能去重写了，或者说我们用 final 修饰了类，那他就不能去继承了。

所以为了防止后续维护者有困惑，我们其实是**有必要或者说有义务说明原因**，这样也不至于发生后续维护上的一些问题。

在很多情况下，我们并需要不急着把这个类或者方法声明为 final，可以到开发的中后期再去决定这件事情，这样的话，我们就能更清楚的明白各个类之间的交互方式，或者是各个方法之间的关系。所以你可能会发现根本就不需要去使用 final 来修饰，或者不需要把范围扩得太大，我们可以重构代码，把 final 应用在更小范围的类或方法上，这样造成更小的影响。

总结

本课时我们主要讲解了 final 的作用，它用在变量、方法或者类上时，其含义是截然不同的，所以我们就逐个对这 3 种情况进行了讲解：修饰变量意味着一旦被赋值就不能被修改；修饰方法意味着不能被重写；修饰类意味着不能被继承。

在讲解 final 修饰变量的时候，我们也分别对成员变量、静态变量和局部变量这三种不同的情况进行了展开分析，可以看到，它们的赋值时机也是各有不同的；如果我们利用空白 final，可以让变量变的更加灵活。还有一种特例那就是 final 去修饰参数，代表着不允许去改变这个参数的内容。

最后介绍了如果我们对方法或者类去使用 final 的话，最好能注明原因，描述清楚我们的设计思想。

上一页 下一页