第八章

类

类声明定义了一个新类并描述了它是如何实现的 (§8.1)。

顶级类(§7.6)是直接在编译单元中声明的类。

嵌套类是其声明发生在另一个类或接口声明体中的任何类。嵌套类可以是成员类(§8.5、§9.5)、局部类(§14.3)或匿名类(§15.9.5)。

一些类型的嵌套类是内部类(§8.1.3)，它是一个可以引用包围类实例、局部变量和类型变量的类。

枚举类(§8.9)是一个用缩写语法声明的类，它定义了一组命名的类实例。

一个记录类(§8.10)是一个用缩写语法声明的类，它定义了一个简单的值聚合。

本章讨论所有类的通用语义。特定于特定类型类的细节将在专门讨论这些构造的小节中讨论。

一个类可以声明为公共的（§8.1.1），因此可以从其模块的任何包中的代码中引用，也可以从其他模块中的代码引用。

一个类可以被声明为抽象类（§8.1.1.l），如果它未完全实现，则必须被声明为抽象类；这样的类不能实例化，但可以通过子类进行扩展。可以明确控制一个类的扩展程度（§8.1.1.2）：可以声明为密封类以限制其子类，也可以声明为 final 类以确保没有子类。除 Object 外的每个类都是单个现有类（§8.1.4）的扩展（即，子类），可以实现接口（§8.1.5）。

一个类可以是泛型的（§8.1.2），也就是说，它的声明可以引入类型变量，这些变量的绑定在该类的不同实例之间有所不同。

类声明可以用注解来修饰（§9.7），就像任何其他类型的声明一样。

类的主体声明成员（字段、方法、类和接口）、实例和静态初始值以及构造函数（§8.1.7）。成员（§8.2）的作用域（§6.3）是该成员所属类别声明的整体。字段、方法、成员类、成员接口和构造函数声明可能包括访问修饰符 public、protected 或 private（§6.6）。类的成员包括声明成员和继承成员（§8.2）。新声明的字段可以隐藏在超类或超接口中声明的字段。新声明的成员类和成员接口可以隐藏在超类或超接口中声明的成员类和成员接口。新声明的方法可以隐藏、实现或重写在超类或超接口中声明的方法。

字段声明（§8.3）描述了类变量和实例变量，类变量只具体化一次，而实例变量则是针对类的每个实例进行具体化的。一个字段可以被声明为 final(§8.3.1.2)，在这种情况下，它只能被赋值一次。任何字段声明都可以包含初始化式。

成员类声明(§8.5)描述了作为周围类成员的嵌套类。成员类可以是静态的，在这种情况下，它们不能访问周围类的实例变量;或者它们可能是内部类。

成员接口声明(§8.5)描述了作为周围类成员的嵌套接口。

方法声明(§8.4)描述了可以被方法调用表达式(§15.12)调用的代码。类方法是相对于类调用的;实例方法是针对某个特定对象调用的，该对象是一个类的实例。如果一个方法的声明没有指明它是如何实现的，那么它必须被声明为 abstract。一个方法可以被声明为 final(§8.4.3.3)，在这种情况下，它不能被隐藏或重写。方法可以通过平台相关的本地代码实现(§8.4.3.4)。`synchronized` 方法(§8.4.3.6)在执行对象体之前自动锁定对象，返回时自动解锁对象，就像使用 `synchronized` 语句一样(§14.19)，从而允许它的活动与其他线程的活动同步(§17(线程和锁))。

方法名可以重载 (§8.4.9)。

实例初始化器(§8.6)是可执行代码块，可用于在创建实例时帮助初始化实例(§15.9)。

静态初始化器(§8.7)是可执行代码块，可用于帮助初始化类。

构造函数(§8.8)类似于方法，但不能通过方法调用直接调用；它们用于初始化新的类实例。与方法一样，它们也可以被重载（§8.8.8）。

8.1 类声明

类声明指定一个类。

类声明有三种：普通类声明、枚举类声明（§8.9）和记录类声明（§9.10）。

ClassDeclaration:
  NormalClassDeclaration
  EnumDeclaration
  RecordDeclaration

NormalClassDeclaration:
  {ClassModifier} `class` *TypeIdentifier [TypeParameters] [ClassExtends]*
    [ClassImplements] [ClassPermits] ClassBody

类也由类实例创建表达式（§15.9.5）和以类主体（§8.9.1）结尾的枚举常量隐式声明。

类声明中的 TypeIdentifier 指定类的名称。

如果一个类与它的任何封闭类或接口具有相同的简单名称，则这是一个编译时错误。

§6.3 和§6.4.1 规定了类声明的作用域和遮蔽。

8.1.1 类修饰符

类声明可以包括类修饰符。

ClassModifier:
(one of)
Annotation `public protected private`
`abstract static final sealed non-sealed strictfp`

关于类声明注解修饰符的规则在§9.7.4 和§9.7.5 中有规定。

访问修饰符 public（§6.6）仅适用于顶级类（§7.6）和成员类（§8.5，§9.5），不适用于局部类（§14.3）或匿名类（§15.9.5）。

protected 和 private 访问修饰符仅适用于成员类。

静态修饰符仅适用于成员类和局部类。

如果同一关键字作为类声明的修饰符出现多次，或者类声明具有多个访问修饰符 public、protected 和 private，则这是编译时错误。

如果类声明有多个修饰符 sealed, non-sealed, 和 final，则这是编译时错误。

> 如果一个类声明中出现两个或多个（不同的）类修饰符，则通常（尽管不是必需的），它们的出现顺序与上面在 ClassModifier 的产品中显示的顺序一致。

8.1.1.1 抽象类

抽象类是不完整或被认为不完整的类。

如果试图使用类实例创建表达式创建抽象类的实例，则为编译时错误（§15.9.1）。

抽象类的子类本身不是抽象的，可以实例化、执行抽象类的构造函数，进而执行该类实例变量的字段初始化器。

只有当一个普通类是抽象类时，它才可能有抽象方法，即已声明但尚未实现的方法（§8.4.3.1）。如果非抽象的普通类具有抽象方法，则这是编译时错误。

如果满足以下任一条件，则类 C 具有抽象方法：

- 任何 C 的成员方法 (§8.2)，声明的或继承的，是抽象的。
- C 的任何超类都有一个通过包访问声明的抽象方法，并且不存在重写 C 或 C 超类中抽象方法的方法。

声明抽象类类型是编译时错误，因此不可能创建实现其所有抽象方法的子类。如果类的成员中有两个抽象方法，它们具有相同的方法签名（§8.4.2），但没有任何类型可以同时匹配这两个方法的返回类型（§8.4.5），则会出现这种情况。

例子 8.1.1.1-1. 抽象类声明

```
abstract class Point {
   int x = 1, y = 1;
   void move(int dx, int dy) {
      x += dx;
      y += dy;
      alert();
   }
   abstract void alert();
}
abstract class ColoredPoint extends Point {
   int color;
}
class SimplePoint extends Point {
   void alert() { }
}
```

这里声明了一个类 Point，它必须声明为抽象的，因为它包含一个名为 alert 的抽象方法的声明。Point 的子类名为 ColoredPoint 继承了抽象方法 alert, 所以它也应该被声明为抽象的。另一方面，Point 的子类名为 SimplePoint 提供了 alert 的实现，所以它可以不被声明为抽象的。

语句:

```
Point p = new Point();
```

会产生编译错误; 类 Point 不能被初始化因为它是抽象的。但是，Point 变量可以通过对 Point 的任何子类的引用正确地初始化，而且 SimplePoint 类不是抽象的, 所以语句:

```
Point p = new SimplePoint();
```

是正确的。SimplePoint 的实例化会导致执行 Point 的 x 和 y 的默认构造函数和字段初始化器。

例子 8.1.1.1-2. 禁止子类的抽象类声明

```
interface Colorable {
   void setColor(int color);
}
abstract class Colored implements Colorable {
   public abstract int setColor(int color);
}
```

这些声明会导致编译时错误: Colored 类的任何子类都不可能提供名为 setColor 的方法的实现，该方法采用 int 类型的一个参数，可以满足两种抽象方法规范，因为 Colorable 接口中的一个方法要求相同的方法不返回值，而 Colored 类中的另一个方法要求同一方法返回 int 类型的值（§8.4）。

只有当意图是创建子类以完成实现时，才应将类类型声明为抽象类型。如果目的只是为了阻止类的实例化，那么正确的表达方式是声明一个没有参数的构造函数（§8.8.10），使其为私有的，从不调用它，并且不声明其他构造函数。这种形式的类通常包含类方法和变量。

类 Math 是无法实例化的类的示例；其声明如下:

```
public final class Math {
    private Math() { }       // never instantiate this class
    . . . declarations of class variables and methods . . .
```

```
}
```

8.1.1.2 sealed, non-sealed, 和 final 类

如果一个类在声明时，它的所有直接子类都是已知的，并且不需要其他直接子类，那么这个类就可以被声明为密封的。

> 当类层次结构用于对域中的各种值进行建模，而不是作为代码继承和重用的机制时，对类的直接子类的显式和详尽的控制非常有用。直接子类本身可以声明为密封的，以便进一步控制类层次结构。

如果一个类的定义是完整的，并且不需要子类，那么它可以被声明为 final。

如果类同时声明为 final 类和抽象类，则为编译时错误，因为此类的实现永远无法完成（§8.1.1.l）。

因为 final 类从来没有任何子类，所以 final 类的方法永远不会被重写（§8.4.8.1）。

如果一个类的直接超类没有被密封（§8.1.4），并且它的直接超接口没有被密封（§8.1.5），并且它本身既不是密封的，也不是 final 的，那么它是可自由扩展的。

具有密封直接超类或密封直接超接口的类可以自由扩展，当且仅当其声明为非密封时。

如果类具有密封的直接超类或密封的直接超级接口，并且未显式或隐式声明为 final、密封或非密封，则这是编译时错误。

> 因此，sealed 关键字的作用是强制所有直接子类显式声明无论它们是 final、sealed 还是 non-sealed。这避免了意外地将密封的类层次结构暴露给不需要的子类。
>
> 枚举类是隐式 final 或隐式密封的，因此它可以实现密封接口。类似地，记录类是隐式 final 的，因此它也可以实现密封接口。

如果类声明为非密封的，但既没有密封的直接超类，也没有密封的直接超接口，则这是编译时错误。

> 因此，非密封类的子类本身不能声明为非密封。

8.1.1.3 精确浮点类

类声明上的 strictfp 修饰符已过时，不应在新代码中使用。它的存在或不存在在编译时或运行时没有影响。

8.1.1.4 静态类

静态修饰符指定嵌套类不是内部类（§8.1.3）。正如类的静态方法在其主体中没有类的当前实例一样，静态嵌套类在其主体内也没有直接封闭的实例。

不允许从静态嵌套类引用词法封闭类、接口或方法声明的类型参数、实例变量、局部变量、形式参数、异常参数或实例方法（§6.5.5.1、§6.6.1 和§15.12.3）。

静态修饰符并不适用于所有嵌套类。它只适用于成员类，其声明可以使用静态修饰符，而不适用于局部类或匿名类，其声明不能使用静态修饰符（§14.3，§15.9.5）。然而，有些局部类是隐式静态的，即局部枚举类和局部记录类，因为所有嵌套枚举类和嵌套记录类都是隐式静态的（§8.9，§8.10）。

8.1.2 泛型类和类型参数

如果类声明声明了一个或多个类型变量，则类是泛型的 (§4.4)。

这些类型变量称为类的类型参数。类型参数部分跟在类名后面，用尖括号分隔。

TypeParameters:
  < *TypeParameterList* >

TypeParameterList:
  TypeParameter {，TypeParameter}

为了方便起见，这里列出了§4.4 的以下产品:

TypeParameter:
  {TypeParameterModifier} TypeIdentifier [TypeBound]

TypeParameterModifier:
  Annotation

TypeBound:
  `extends` *TypeVariable*
  `extends` *ClassOrInterfaceType {AdditionalBound}*

AdditionalBound:
  `&` *InterfaceType*

关于类型参数声明的注解修饰符的规则见§9.7.4 和§9.7.5。

在类的类型参数部分，如果 S 是 T 的边界，类型变量 T 直接依赖于类型变量 S，而如果 T 直接依赖于 S，或者 T 直接依赖于依赖于 S 的类型变量 U(递归使用此定义)，则 T 依赖于 S。

如果类的类型参数部分中的类型变量依赖于自身，则为编译时错误。

类的类型形参的作用域和遮蔽在§6.3 和§6.4.1 中有规定。

从静态上下文或嵌套类或接口引用类的类型形参是受限制的，如§6.5.5.1 中的规定。

泛型类声明定义了一组参数化类型（§4.5），每个可能的类型参数的参数化由类型参数决定。所有这些参数化类型在运行时共享同一个类。

例如，执行代码:

```
Vector<String> x = new Vector<String>();
Vector<Integer> y = new Vector<Integer>();
boolean b = x.getClass() == y.getClass();
```

将导致变量 b 的值为 true。

如果泛型类是 Throwable 的直接或间接子类（§11.1.1），则为编译时错误。

由于 Java 虚拟机的捕获机制仅适用于非泛型类，因此需要此限制。

例子 8.1.2-1. 互递归类型变量边界

```
interface ConvertibleTo<T> {
  T convert();
}
class ReprChange<T extends ConvertibleTo<S>,
                 S extends ConvertibleTo<T>> {
    T t;
    void set(S s) { t = s.convert(); }
    S get()       { return t.convert(); }
}
```

例子 8.1.2-2. 嵌套泛型类

```
class Seq<T> {
    T head;
    Seq<T> tail;

    Seq() { this(null, null); }
    Seq(T head, Seq<T> tail) {
      this.head = head;
      this.tail = tail;
    }
    boolean isEmpty() { return tail == null; }

    class Zipper<S> {
      Seq<Pair<T,S>> zip(Seq<S> that) {
        if (isEmpty() || that.isEmpty()) {
          return new Seq<Pair<T,S>>();
         } else {
               Seq<T>.Zipper<S> tailZipper =
                 tail.new Zipper<S>();
               return new Seq<Pair<T,S>>(
                 new Pair<T,S>(head, that.head),
                   tailZipper.zip(that.tail));
          }
       }
    }
}
class Pair<T, S> {
    T fst; S snd;
    Pair(T f, S s) { fst = f; snd = s; }
}
class Test {
    public static void main(String[] args) {
      Seq<String> strs =
           new Seq<String>(
             "a",
             new Seq<String>("b",
```

```
                                new Seq<String>()));
        Seq<Number> nums =
              new Seq<Number>(
                  new Integer(1),
                  new Seq<Number>(new Double(1.5),
                                  new Seq<Number>()));

        Seq<String>.Zipper<Number> zipper =
              strs.new Zipper<Number>();

        Seq<Pair<String,Number>> combined =
              zipper.zip(nums);

    }
}
```

8.1.3 内部类和封闭实例

内部类是不是显式或隐式静态的嵌套类。

内部类是以下之一:

- 不是显式或隐式静态的成员类 (§8.5)
- 不是隐式静态的局部类 (§14.3)
- 匿名类 (§15.9.5)

以下嵌套类是隐式静态的，所以不是内部类:

- 成员枚举类 (§8.9)
- 局部枚举类(§14.3)
- 成员记录类(§8.10)
- 局部记录类(§14.3)
- 接口的成员类(§9.5)

所有适用于嵌套类的规则也适用于内部类。特别是，内部类可以声明和继承静态成员(§8.2)，并声明静态初始化器(§8.7)，即使内部类本身不是静态的。

> 没有“内部接口”，因为每个嵌套接口都是隐式静态的(§9.1.1.3)。

例子 8.1.3-1. 内部类声明和静态成员

```
class HasStatic {
   static int j = 100;
}

class Outer {

  class Inner extends HasStatic{
     static {
        System.out.println("Hello from Outer.Inner");
     }
```

```
        static int x = 3;
        static final int y = 4;

        static void hello() {
            System.out.println("Hello from Outer.Inner.hello");
        }

        static class VeryNestedButNotInner
            extends NestedButNotInner {}
    }

    static class NestedButNotInner {
        int z = Inner.x;
    }

    interface NeverInner {}    // Implicitly static, so never inner
}
```

在 Java SE 16 之前，内部类不能声明静态初始化器，只能声明常量变量的静态成员(§4.12.4)。

一个构造(语句、局部变量声明语句、局部类声明、局部接口声明或表达式)发生在静态上下文中，如果最内层:

- 方法声明,
- 字段声明,
- 构造函数声明,
- 实例初始化器,
- 静态初始化器，或者
- 显式构造函数调用语句

包围该构造的是下列之一:

- 静态方法声明 (§8.4.3.2, §9.4)
- 静态字段声明 (§8.3.1.1, §9.3)
- 静态初始化器 (§8.7)
- 显式构造函数调用语句 (§8.8.7.1)

注意，出现在构造函数声明或实例初始器中的构造不会出现在静态上下文中。

静态上下文的目的是划分代码，这些代码不能显式或隐式引用类的当前实例，该类的声明在词法上包含静态上下文。因此，出现在静态上下文中的代码会受到以下方面的限制:

- this 表达式被禁止(包括限定的和非限定的) (§15.8.3, §15.8.4)。
- 字段访问、方法调用和方法引用不能由 super 限定(§15.11.2, §15.12.3, §15.13.1)。
- 不允许对任何词法封闭类或接口声明的实例变量进行非限定引用 (§6.5.6.1)。
- 不允许对任何词法封闭类或接口声明的实例方法进行非限定调用 (§15.12.3)。
- 不允许引用任何词法封闭类或接口声明的类型参数 (§6.5.5.1)。

- 不允许引用类型参数、局部变量、形式参数和异常参数，这些参数由位于直接封闭类或接口声明之外的任何词法封闭类或接口声明的方法或构造函数声明 (§6.5.5.1, §6.5.6.1)。
- 局部普通类的声明（与局部枚举类相反）和匿名类的声明都指定了内部类，但在实例化时没有直接封闭实例（§15.9.2）。
- 实例化内部成员类的类实例创建表达式必须是限定的 (§15.9)。

如果 O 是 C 的直接封闭类或接口声明，并且 C 的声明不发生在静态上下文中，则内部类 C 是类或接口 O 的直接内部类。

> 如果内部类是局部类或匿名类，则可以在静态上下文中声明，在这种情况下，它不被视为任何封闭类或接口的内部类。

如果类 C 是 O 的直接内部类或 O 的内部类的内部类，则它是类或接口 O 的内部类。

> 内部类的直接封闭类或接口声明是接口，这是不寻常的，但也是可能的。只有当类是在默认或静态方法体中声明的局部或匿名类时才会发生这种情况（§9.4）。

类或接口 O 是其自身的第 0 个词法封装类或接口声明。

如果类 O 是 C 的第 n-1 个词法封闭类声明的直接封闭类声明，则它是 C 的第 n 个词法封闭类声明。

类或接口 O 的直接内部类 C 的实例 i 与 O 的实例相关联，称为 i 的直接封闭实例。对象的直接封闭实例（如有）在对象创建时确定（§15.9.2）。

对象 o 是其自身的第零个词法封闭实例。

如果对象 o 是实例 i 的第 n-1 个词法封闭实例的直接封闭实例，则对象 o 是实例 i 的第 n 个词法封闭实例。

在静态上下文中声明的内部局部类或匿名类的实例没有直接封闭实例。此外，静态嵌套类的实例（§8.1.1.4）没有直接封闭实例。

对于 C 的每个超类 S，它本身是一个类或接口 SO 的直接内部类，存在一个与 i 关联的 SO 实例，称为 i 相对于 S 的直接封闭实例。当通过显式构造函数调用语句调用超类构造函数时，确定对象相对于其类的直接超类（如果有）的直接封闭实例 (§8.8.7.1)。

当一个内部类（其声明在静态上下文中不出现）引用一个实例变量，该实例变量是词法封闭类或接口声明的成员时，将使用相应词法封闭实例的变量。

按照§6.5.6.1 的规定，在内部类中使用但未声明的任何局部变量、形式参数或异常参数必须是 final 的或实际 final 的（§4.12.4）。

在内部类中使用但未声明的任何局部变量必须在内部类主体之前进行明确赋值（§16），否则会发生编译时错误。

> 关于变量使用的类似规则适用于 lambda 表达式的主体 (§15.27.2)。

词法封闭类或接口声明的空 final 字段（§4.12.4）可能无法在内部类中赋值，否则会出现编

译时错误。

例子 8.1.3-2. 内部类声明

```
class Outer {
    int i = 100;
    static void classMethod() {
        final int l = 200;
        class LocalInStaticContext {
            int k = i; // Compile-time error
            int m = l; // OK
        }

    void foo() {
        class Local {  // A local class
            int j = i;
        }
    }
}
```

由于在静态方法 classMethod 中，类 LocalInstasticContext 的声明发生在静态上下文中。类 Outer 的实例变量在静态方法体中不可用。特别是，Outer 的实例变量在 LocalInStaticContext 的主体内不可用。然而，来自周围方法的局部变量可以被引用而没有错误（前提是它们被声明为 final 或实际上是 final）。

声明不在静态上下文中出现的内部类可以自由引用其封闭类声明的实例变量。实例变量始终是相对于实例定义的。对于封闭类声明的实例变量，必须针对内部类的封闭实例定义实例变量。例如，上面的类 Local 有一个类 Outer 的封闭实例。作为另一示例:

```
class WithDeepNesting {
    boolean toBe;
    WithDeepNesting(boolean b) { toBe = b; }

    class Nested {
        boolean theQuestion;
        class DeeplyNested {
            DeeplyNested(){
               theQuestion = toBe || !toBe;
            }
          }
        }
     }
```

在这里，WithDeepNesting.Nested.DeeplyNested 的每个实例有一个类 WithDeepNesting.Nested 的封闭实例(它的立即封闭实例)以及类 WithDeepNesting 的封闭实例 (它的第二个词法封闭实例)。

8.1.4 超类和子类

普通类声明中的可选 extends 子句指定要声明的类的直接超类类型。

ClassExtends:
`extends` *ClassType*

extends 子句不能出现在 Object 类的定义中，否则会发生编译时错误，因为它是原始类，没有直接的超类类型。

ClassType 必须命名一个可访问的类（§6.6），否则会发生编译时错误。

如果 ClassType 命名了一个密封的类（§8.1.1.2），并且被声明的类不是命名类的允许直接子类（§9.1.6），则这是编译时错误。

如果 ClassType 将类命名为 final，则是编译时错误，因为 final 类不允许有子类（§8.1.1.2）。

如果 ClassType 将只能通过枚举类扩展的类命名为 Enum(§8.9)，或者将只能通过记录类扩展的类命名为 Record(§8.10)，则会出现编译时错误。

如果 ClassType 有类型参数，它必须表示一个格式良好的参数化类型(§4.5)，并且任何类型参数都不能是通配符类型参数，否则会发生编译时错误。

声明中缺少 extends 子句的类的直接超类类型如下:

- Object 类没有直接超类类型。
- 对于普通类声明的 Object 以外的类，直接超类类型是 Object。
- 对于枚举类 E, 直接超类类型是 Enum<E>。
- 对于匿名类, 直接超类类型定义在§15.9.5。

类的直接超类是由其直接超类类型命名的类。直接超类很重要，因为它的实现用于得到被声明的类的实现。

超类关系是直接超类关系的传递闭包。如果下列任意一个为真，则 A 类是 C 类的超类:

- A 是 C 的直接超类。
- 其中类 B 是 C 的直接超类，A 是 B 的超类，递归地应用这个定义。

一个类被称为它的直接超类的直接子类，以及它的每个超类的子类。

例子 8. 1. 4-1. 直接超类和子类

```
class Point { int x, y; }
final class ColoredPoint extends Point { int color; }
class Colored3DPoint extends ColoredPoint { int z; }  // error
```

在这里，关系如下:

- 类 Point 是 Object 的直接子类。
- 类 Object 是类 Point 的直接超类。
- 类 ColoredPoint 是类 Point 的直接子类。
- 类 Point 是类 ColoredPoint 的直接超类。

类 Colored3dPoint 的声明产生编译时错误，因为它尝试扩展 final 类 ColoredPoint。

例子 8.1.4-2. 超类和子类

```
class Point { int x, y; }
```

```
class ColoredPoint extends Point { int color; }
final class Colored3dPoint extends ColoredPoint { int z; }
```

在这里，关系如下:

- 类 Point 是类 ColoredPoint 的超类。
- 类 Point 是类 Colored3dPoint 的超类。
- 类 ColoredPoint 是类 Point 的子类。
- 类 ColoredPoint 是类 Colored3dPoint 的超类。
- 类 Colored3dPoint 是类 ColoredPoint 的子类。
- 类 Colored3dPoint 是类 Point 的子类。

如果在 C 的 extends 或 implements 子句中将 A 作为超类或超接口，或者作为超类或超接口名称的完全限定形式的限定符，则类 C 直接依赖于类或接口 A。

如果下列任一为真，则类 C 依赖于类或接口 A:

- *C* 直接依赖 *A*。
- C 直接依赖于依赖于 A(§9.1.3)的接口 I。
- C 直接依赖于依赖于 A 的类 B，递归地应用这个定义。

如果类依赖于自身，则为编译时错误。

如果循环声明的类在运行时被检测到，在类加载时，则会抛出 ClassCircularityError(§12.2.1)。

例子 8. 1. 4-3. 类依赖于它自己

```
class Point extends ColoredPoint { int x, y; }
class ColoredPoint extends Point { int color; }
```

这个程序会导致编译时错误，因为类 Point 依赖于它自己。

8.1.5 超接口

类声明中的可选 implements 子句指定要声明的类的直接超接口类型。

ClassImplements:
`implements` *InterfaceTypeList*

InterfaceTypeList:
InterfaceType {, InterfaceType}

每一个 InterfaceType 必须命名一个可访问的接口 (§6.6), 否则会发生编译时错误。

如果任何 InterfaceType 命名了一个被密封的接口(§9.1.1.4)，并且声明的类不是命名接口的被允许的直接子类(§9.1.4)，那么这是一个编译时错误。

如果 InterfaceType 有类型参数，它必须表示一个格式良好的参数化类型(§4.5)，并且任何类型参数都不能是通配符类型参数，否则会发生编译时错误。

如果同一个接口在一个 implements 子句中由直接超接口类型命名不止一次，则会出现编译时错误。即使接口以不同的方式命名也是如此。

例子 8. 1. 5-1. 非法超接口

```
class Redundant implements java.lang.Cloneable, Cloneable {
   int x;
}
```

这个程序会导致编译时错误，因为名称 java.lang.Cloneable 和 Cloneable 引用相同的接口。

一个没有 implements 子句的类没有直接的超接口类型，只有一个例外:匿名类可以有超接口类型(§15.9.5)。

如果接口由类的直接超接口类型之一命名，则接口是类的直接超接口。

如果下列任何一项为真，接口 I 就是类 C 的超接口:

- I 是 C 的直接超接口。
- C 有一些直接的超接口 J，其中 I 是一个超级接口，使用§9.1.3 中给出的“接口的超接口”的定义。
- I 是 C 的直接超类的超接口。

一个类可以以多种方式拥有超接口。

一个类被称为直接实现其直接超接口，并实现其所有超接口。

一个类被称为它的直接超接口的直接子类，以及它所有超接口的子类。

一个类不能声明一个直接超类类型和一个直接超接口类型，或两个直接超接口类型，它们是或具有超类型（§4.10.2），这些超类型是相同泛型接口（§9.1.2）的不同参数化，或泛型接口的参数化和命名相同泛型接口的原始类型。在这种冲突的情况下，会发生编译时错误。

引入此要求是为了支持按类型擦除的转换（§4.6）。

例子 8.1.5-2. 超接口

```
interface Colorable {
    void setColor(int color);
    int getColor();
}
enum Finish { MATTE, GLOSSY }
interface Paintable extends Colorable {
    void setFinish(Finish finish);
    Finish getFinish();
}
```

```
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {
     int color;
     public void setColor(int color) { this.color = color; }
     public int getColor() { return color; }
}
class PaintedPoint extends ColoredPoint implements Paintable {
     Finish finish;
     public void setFinish(Finish finish) {
        this.finish = finish;
     }
     public Finish getFinish() { return finish; }
}
```

在这里，关系如下:

- 接口 Paintable 是类 PaintedPoint 的超接口。
- 接口 Colorable 是类 ColoredPoint 和类 PaintedPoint 的超接口。
- 接口 Paintable 是接口 Colorable 的子接口, Colorable 接口是 Paintable 的超接口, 正如 §9.1.3 定义的。

PaintedPoint 类有 Colorable 作为超接口，这是因为它是 ColoredPoint 的超接口，也因为它是 Paintable 的超接口。

例子 8.1.5-3. 接口的非法多重继承

```
interface I<T> {}
class B implements I<Integer> {}
class C extends B implements I<String> {}
```

类 C 会导致编译时错误，因为它试图成为 I<Integer> 和 I<String>两者的子类型。

除非被声明的类是抽象的，否则每个直接超接口的所有抽象成员方法都必须通过该类的声明或从直接超类或直接超接口继承的现有方法声明来实现(§8.4.8.1)，因为非抽象的类不允许有抽象方法(§8.1.1.1)。

类的超接口的每个默认方法(§9.4.3)都可以被类中的方法重写;如果不是，则通常继承默认方法，其行为由默认主体指定。

允许一个类中的单个方法声明实现多个超接口的方法。

例子 8.1.5-4. 实现超接口的方法

```
interface Colorable {
    void setColor(int color);
    int getColor();
}
class Point { int x, y; };
class ColoredPoint extends Point implements Colorable {
    int color;
}
```

这个程序会导致编译时错误，因为 ColoredPoint 不是一个抽象类，但没有提供 Colorable 接口的 setColor 和 getColor 方法的实现。

在以下程序中:

```
interface Fish { int getNumberOfScales(); }
interface Piano { int getNumberOfScales(); }
class Tuna implements Fish, Piano {
    // You can tune a piano, but can you tuna fish?
    public int getNumberOfScales() { return 91; }
}
```

Tuna 类中的 getNumberOfScales 方法有一个名称、签名和返回类型，它与 Fish 接口中声明的方法相匹配，也与 Piano 接口中声明的方法相匹配;我们认为这两者都可以实现。

另一方面，在这样的情况下:

```
interface Fish       { int   getNumberOfScales(); }
interface StringBass { double getNumberOfScales(); }
class Bass implements Fish, StringBass {
    // This declaration cannot be correct,
    // no matter what type is used.
    public ?? getNumberOfScales() { return 91; }
}
```

不可能声明一个名为 getNumberOfScales 的方法，它的签名和返回类型与 Fish 接口和 StringBass 接口中声明的方法兼容，因为一个类不能有多个具有相同签名和不同原始返回类型的方法(§8.4)。因此，一个类不可能同时实现 Fish 和 StringBass 接口(§8.4.8)。

8.1.6 允许直接子类

在普通的类声明中，可选的 permits 子句指定了所有被声明类的直接子类(§8.1.1.2)。

ClassPermits:
`permits` *TypeName {, TypeName}*

如果类声明有一个 permits 子句但没有密封修饰符，则是编译时错误。

每个 TypeName 必须命名一个可访问类(§6.6)，否则会发生编译时错误。

如果在 permits 子句中不止一次地指定同一个类，则会出现编译时错误。即使类以不同的方式命名也是如此。

类的规范名称不需要在 permits 子句中使用，但是 permits 子句只能指定一个类一次。例如，以下程序编译失败:

```
package p;

sealed class A permits B, C, p.B {}      // error

non-sealed class B extends A {}
non-sealed class C extends A {}
```

如果一个密封类 C 与一个命名模块相关联（§7.3），则 C 声明的 permits 子句中指定的每个类都必须与 C 的同一个模块相关联，否则会发生编译时错误。

如果一个密封类 C 与一个未命名模块相关联（§7.7.5），则 C 声明的 permits 子句中指定的每个类必须与 C 属于同一个包，否则会发生编译时错误。

密封类及其直接子类需要以循环的方式相互引用，分别在 permits 和 extends 子句中。因此，在模块化代码库中，它们必须位于同一个模块中，因为不同模块中的类不能以循环的方式相互引用。在任何情况下，共同位置都是可取的，因为密封的类层次结构应该始终在单个维护域中声明，而同一开发人员或同一组开发人员负责维护这个层次结构。命名模块通常代表模块化代码库中的维护域。

如果密封类 C 的声明有一个 permits 子句，那么允许的 C 的直接子类是 permits 子句中指定的类。

permits 子句规定的每个被允许的直接子类必须是 C 的直接子类（§8.1.4），否则会发生编译时错误。

如果密封类 C 的声明缺少 permits 子句，则 C 的允许直接子类如下：

- 如果 C 不是枚举类，则其允许的直接子类是在与 C（§7.3）相同的编译单元中声明的类，这些类具有规范名称（§6.7），并且它们的直接超类是 C。

  也就是说，允许的直接子类被推断为指定 C 为其直接超类的同一编译单元中的类。规范名称的要求意味着不会考虑局部类或匿名类。

  如果密封的类 C 的声明缺少一个 permits 子句，并且 C 没有允许的直接子类，那么这是一个编译时错误。
- 如果 C 是一个枚举类，那么它允许的直接子类(如果有的话)在§8.9 中指定。

8.1.7 类主体和成员声明

类主体可以包含类成员的声明，即字段(§8.3)、方法(§8.4)、类和接口(§8.5)。

类主体也可以包含实例初始化器(§8.6)、静态初始化器(§8.7)和类的构造函数声明(§8.8)。

ClassBody:
{ *{ClassBodyDeclaration}* }

ClassBodyDeclaration:
ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration
ClassDeclaration
InterfaceDeclaration
;

在 C 类中声明或被 C 类继承的成员 m 的声明的作用域和遮蔽在§6.3 和§6.4.1 中规定。

如果 C 是嵌套类，则在封闭作用域中可能存在与 m 相同类型（变量、方法或类型）和名称的定义。(作用域可以是块、类或包。) 在所有这些情况下，在 C 中声明或由 C 继承的成员 m 遮蔽了相同类型和名称的其他定义。

8.2 类成员

一个类的成员包括以下所有成员：

- 成员继承自其直接超类类型（§8.1.4），但 Object 类中没有直接超类类型除外
- 成员继承自任何直接超接口类型 (§8.1.5)
- 在类主体中声明的成员 (§8.1.7)

声明为私有的类的成员不会被该类的子类继承。

只有声明为 protected 或 public 的类成员才能被声明在包中的子类继承，而不是在声明类的包中。

构造函数、静态初始化器和实例初始化器不是成员，因此不能继承。

我们用成员的类型这个短语来表示：

- 对于字段，它的类型。
- 对于方法，一个有序的四元组，由以下组成：
  - 类型参数: 方法成员的任何类型参数的声明。
  - 参数类型:方法成员的参数类型列表。
  - 返回类型: 方法成员的返回类型。
  - throws 子句: 方法成员的 throws 子句中声明的异常类型。

一个类的字段、方法、成员类和成员接口可能有相同的名称，因为它们用于不同的上下文中，并且通过不同的查找过程消除了歧义(§6.5)。然而，作为一种风格，这是不鼓励的。

例子 8.2-1. 使用类成员

```
class Point {
   int x, y;
   private Point() { reset(); }
   Point(int x, int y) { this.x = x; this.y = y; }
   private void reset() { this.x = 0; this.y = 0; }
}
class ColoredPoint extends Point {
   int color;
   void clear() { reset(); }    // error
}
class Test {
   public static void main(String[] args) {
      ColoredPoint c = new ColoredPoint(0, 0);    // error
```

```
        c.reset(); // error }
    }
```

这个程序会导致四个编译错误。

出现一个错误是因为 ColoredPoint 没有像 main 中使用的那样声明两个 int 形参的构造函数。这说明了 ColoredPoint 没有继承其超类 Point 的构造函数的事实。

另一个错误发生是因为 ColoredPoint 没有声明构造函数，因此它的默认构造函数是隐式声明的(§8.8.9)，这个默认构造函数等价于:

```
ColoredPoint() { super(); }
```

它调用了类 ColoredPoint 的直接超类的无参构造函数。错误在于没有参数的 Point 的构造函数是私有的，因此即使通过超类构造函数调用，也不能通过 Point 类之外访问(§8.8.7)。

由于类 Point 的方法 reset 是私有的，因此不会被类 ColoredPoint 继承，所以还会出现另外两个错误。因此，ColoredPoint 类的方法 clear 和 Test 类的方法 main 中的方法调用是不正确的。

例子 8.2-2. 使用包访问继承类成员

考虑这样一个例子:points 包声明了两个编译单元:

```
package points;
public class Point {
   int x, y;
   public void move(int dx, int dy) { x += dx; y += dy; }
}
```

以及:

```
package points;
public class Point3d extends Point {
   int z;
   public void move(int dx, int dy, int dz) {
      x += dx; y += dy; z += dz;
   }
}
```

还有第三个编译单元，在另一个包里，是:

```
import points.Point3d;
class Point4d extends Point3d {
    int w;
    public void move(int dx, int dy, int dz, int dw) {
        x += dx; y += dy; z += dz; w += dw; // compile-time errors
    }
}
```

在这里，points 包中的两个类都可以编译。类 Point3d 继承了类 Point 的字段 x 和 y, 因为它和 Point 在同一个包中。类 Point4d 是一个不同的包，不继承类 Point 的字段 x 和 y 或者类 Point3d 的字段 z，因此编译失败。

编写第三个编译单元的更好方法是:

```
import points.Point3d;
class Point4d extends Point3d {
   int w;
   public void move(int dx, int dy, int dz, int dw) {
      super.move(dx, dy, dz); w += dw;
```

```
    }
}
```

使用超类 Point3d 的 move 方法来处理 dx、dy 和 dz。如果 Point4d 是以这种方式编写的，那么它将不会编译错误。

例子 8.2-3. 公有和受保护类成员的继承

给定类 Point:

```
package points;
public class Point {
    public int x, y;
    protected int useCount = 0;
    static protected int totalUseCount = 0;
    public void move(int dx, int dy) {
        x += dx; y += dy; useCount++; totalUseCount++;
    }
}
```

public 和 protected 的字段 x, y, useCount 和 totalUseCount 在 Point 的所有子类中继承。

因此，该测试程序在另一个包中可以成功编译:

```
class Test extends points.Point {
  public void moveBack(int dx, int dy) {
        x -= dx; y -= dy; useCount++;totalUseCount++; }
}
```


例子 8.2-4. 继承私有类成员

```
class Point {
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy; totalMoves++;
    }
    private static int totalMoves;
    void printMoves() { System.out.println(totalMoves); }
}
class Point3d extends Point {
    int z;
    void move(int dx, int dy, int dz) {
    super.move(dx, dy); z += dz; totalMoves++; // error
    }
}
```

类变量 totalMoves 只能在类 Point 里使用；它不被子类 Point3d 继承。因为类 Point3d 的方法 move 试图增加 totalMoves，所以会发生编译时错误。

例子 8.2-5. 访问不可访问类的成员

即使一个类可能不会被声明为公共的，但在运行时，该类的实例也可以在包外部进行编码，在包中通过公共超类或超接口进行声明。类的实例可以赋值给这种公共类型的变量。如果调用此类变量引用的对象的公共方法实现或重写了公共超类或超接口的方法，则可以调用该类的方法。(在这种情况下，方法必须

声明为公共的，即使它是在非公共的类中声明的。)

考虑编译单元:

```
package points;
public class Point {
    public int x, y;
    public void move(int dx, int dy) {
        x += dx; y += dy;
    }
}
```

以及另一个包的另一个编译单元:

```
package morePoints;
class Point3d extends points.Point {
    public int z;
    public void move(int dx, int dy, int dz) {
       super.move(dx, dy); z += dz;
    }
    public void move(int dx, int dy) {
       move(dx, dy, 0);
    }
}
public class OnePoint {
    public static points.Point getOne() {
        return new Point3d();
    }
}
```


在第三个包里调用 morePoints.OnePoint.getOne()会返回一个 Point3d，它可以用作 Point，即使类型 Point3d 在包 morePoints 外不可用。然后可以为该对象调用方法 move 的两个参数版本，这是允许的，因为 Point3d 的方法 move 是公共的(因为它必须是，对于任何重写公共方法的方法本身必须是公共的，这样类似的情况才能正确解决)。该对象的字段 x 和 y 也可以从这样的第三个包访问。

虽然类 Point3d 的字段 z 是公共的，但不可能从包 morePoints 之外的代码访问这个字段，只通过一个 Point 类型的变量 p 中的类 Point3d 实例的引用。这是因为表达式 p.z 是不正确的，因为 p 的类型为 Point，类 Point 没有名为 z 的字段；并且，表达式((Point3d)p).z 也不正确，因为类类型 Point3d 不能在包 morePoints 之外引用。

但是，将字段 z 声明为 public 并不是无用的。如果在包 morePoints 中有一个 Point3d 的公共子类 Point4d:

```
package morePoints;
public class Point4d extends Point3d {
    public int w;
    public void move(int dx, int dy, int dz, int dw) {
       super.move(dx, dy, dz); w += dw;
    }
}
```

那么类 Point4d 将继承字段 z，它是公共的，然后可以通过 Point4d 公共类型的变量和表达式，由包中的代码访问而不是 morePoints。

8.3 字段声明

类的变量通过字段声明引入。

FieldDeclaration:
  {FieldModifier} UnannType VariableDeclaratorList ;

VariableDeclaratorList:
  VariableDeclarator {, VariableDeclarator}

VariableDeclarator:
  VariableDeclaratorId [= VariableInitializer]

VariableDeclaratorId:
  Identifier [Dims]

VariableInitializer:
  Expression
  ArrayInitializer

UnannType:
  UnannPrimitiveType
  UnannReferenceType

UnannPrimitiveType:
  NumericType
  `boolean`

UnannReferenceType:
  UnannClassOrInterfaceType
  UnannTypeVariable
  UnannArrayType

UnannClassOrInterfaceType:
  UnannClassType
  UnannInterfaceType

UnannClassType:
  TypeIdentifier [TypeArguments]
  PackageName . {Annotation} TypeIdentifier [TypeArguments]
  UnannClassOrInterfaceType . {Annotation} TypeIdentifier
    [TypeArguments]

UnannInterfaceType:
  UnannClassType
UnannTypeVariable:

TypeIdentifier

UnannArrayType:
UnannPrimitiveType Dims
UnannClassOrInterfaceType Dims
UnannTypeVariable Dims

为了方便起见，以下是§4.3 的产品:

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

FieldDeclaration 中的每个声明符声明一个字段。声明符中的 Identifier 可以用于名称中引用字段。

通过使用多个声明符，可以在一个 FieldDeclaration 中声明多个字段; FieldModifiers 和 UnannType 应用于声明中的所有声明符。

FieldModifier 子句的描述见§8.3.1。

如果 UnannType 和 VariableDeclaratorId 中没有出现方括号对，则字段声明的类型由 UnannType 表示，否则由§10.2 指定。

字段声明的作用域和遮蔽在§6.3 和§6.4.1 中有规定。

类声明的主体声明两个具有相同名称的字段是编译时错误。

如果一个类声明了一个具有特定名称的字段，那么该字段的声明被认为隐藏了超类和类的超接口中任何具有相同名称的字段的可访问声明。

> 在这方面，字段的隐藏不同于方法的隐藏(§8.4.8.3)，因为在字段隐藏中不区分静态和非静态字段，而在方法隐藏中区分静态和非静态方法。

如果隐藏字段是静态的，可以通过使用限定名(§6.5.6.2)来访问，或者通过使用包含关键字 super(§15.11.2)的字段访问表达式或强制转换为超类类型来访问。

> 在这方面，隐藏字段类似于隐藏方法。

如果一个字段声明隐藏了另一个字段的声明，这两个字段不必具有相同的类型。

一个类从它的直接超类和直接超接口继承了超类和超接口的所有非私有字段，这些字段在类中可以访问(§6.6)，并且不会被类中的声明所隐藏。

超类的私有字段可以被子类访问-例如，如果两个类都是同一个类的成员。然而，私有字段永远不会被子类继承。

一个类可以从它的超类和超接口继承多个具有相同名称的字段，或者只从它的超接口继承。这种情况本身不会导致编译时错误。但是，在类体中试图通过简单的名称引用任何这样的字段都会导致编译时错误，因为引用是不明确的。

从接口继承相同字段声明可能有多个路径。在这种情况下，字段被认为只继承一次，并且可以通过它的简单名称引用它，而不会产生歧义。

例子 8.3-1. 多继承字段

一个类可以继承两个或多个具有相同名称的字段，可以来自它的超类和一个超接口，也可以来自两个超接口。如果试图通过简单名称引用任何不明确继承的字段，就会发生编译时错误。可以使用一个限定名或包含关键字 super(§15.11.2)的字段访问表达式来明确地访问这些字段。在程序中:

```
interface Frob { float v = 2.0f; }
class SuperTest { int v = 3; }
class Test extends SuperTest implements Frob {
  public static void main(String[] args) {
    new Test().printV();
  }
  void printV() { System.out.println(v); }
}
```

类 Test 继承了两个名为 v 的字段，一个来自于它的超类 SuperTest，另一个来自于它的超接口 Frob。这本身是允许的，但是由于在 printV 方法中使用了简单的名称 v，所以会出现编译时错误:无法确定要使用哪个 v。

下面的变体使用字段访问表达式 super.v 引用 SuperTest 类中声明的字段 v，并使用限定名 Frob.v 引用 Frob 接口中声明的字段 v:

```
interface Frob { float v = 2.0f; }
class SuperTest { int v = 3; }
class Test extends SuperTest implements Frob {
  public static void main(String[] args) {
    new Test().printV();
}
  void printV() {
    System.out.println((super.v + Frob.v)/2);
  }
}
```


程序编译并打印:

```
2.5
```

即使两个不同的继承字段具有相同的类型、相同的值，并且都是 final 字段，通过简单名称对任何一个字段的引用都会被认为是不明确的，并导致编译时错误。在程序中:

```
interface Color         { int RED=0, GREEN=1, BLUE=2;  }
interface TrafficLight { int RED=0, YELLOW=1, GREEN=2; }
class Test implements Color, TrafficLight {
    public static void main(String[] args) {
        System.out.println(GREEN);  // compile-time error
        System.out.println(RED);    // compile-time error
    }
}
```

对 GREEN 的引用被认为是不明确的并不令人惊讶，因为类 Test 用不同的值继承了 GREEN 的两个不同声明。这个示例的要点是，对 RED 的引用也被认为是不明确的，因为继承了两个不同的声明。两个名为 RED 的字段碰巧具有相同的类型和相同的不变值，这一事实并不影响这个判断。

例子 8.3-2. 字段的重新继承

如果同一个字段声明通过多个路径从接口继承，则认为该字段只继承了一次。例如，在代码中:

```
interface Colorable {
    int RED = 0xff0000, GREEN = 0x00ff00, BLUE = 0x0000ff;
}
interface Paintable extends Colorable {
    int MATTE = 0, GLOSSY = 1;
}
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {}
class PaintedPoint extends ColoredPoint implements Paintable {
    int p = RED;
}
```

字段 RED、GREEN 和 BLUE 被 PaintedPoint 类通过其直接超类 ColoredPoint 和直接超接口 Paintable 继承。尽管如此，在类 PaintedPoint 中可以毫无歧义地使用简单的名称 RED、GREEN 和 BLUE 来引用接口 Colorable 中声明的字段。

8.3.1 字段修饰符

FieldModifier:
(one of)
Annotation `public protected private static final transient volatile`

关于字段声明的注解修饰符的规则见§9.7.4 和§9.7.5。

如果同一个关键字多次作为字段声明的修饰符出现，或者字段声明有多个访问修饰符 public、protected 和 private(§6.6)，这将是编译时错误。

如果在字段声明中出现两个或多个(不同的)字段修饰符，习惯上(虽然不是必需的)，它们的出现顺序应该与上面 FieldModifier 产品中显示的顺序一致。

8.3.1.1 静态字段

如果一个字段被声明为静态的，那么无论最终创建了多少个类实例(可能为零)，该字段都只存在一个化身。静态字段，有时也称为类变量，在类初始化时被具体化(§12.4)。

未声明为静态的字段称为实例变量，有时也称为非静态字段。无论何时创建一个类的新实例（§12.5），都会为该类或其任何超类中声明的每个实例变量创建一个与该实例关联的新变量。

类变量的声明引入了静态上下文（§8.1.3），这限制了引用当前对象的构造的使用。值得注意的是，在静态上下文中禁止使用 this 和 super 关键字（§15.8.3，§15.11.2），对实例变量、实例方法和词法封装声明的类型参数的非限定引用也是如此（§6.5.5.1，§6.6.1，§15.12.3）。

根据§6.5.6.1 的规定，从静态上下文或嵌套类或接口引用实例变量是受限的。

例子 8.3.1. 1-1. 静态字段

```
class Point {
  int x, y, useCount;
  Point(int x, int y) { this.x = x; this.y = y; }
  static final Point origin = new Point(0, 0);
}
class Test {
    public static void main(String[] args) {
        Point p = new Point(1,1);
        Point q = new Point(2,2);
        p.x = 3;
        p.y = 3;
        p.useCount++;
        p.origin.useCount++;
        System.out.println("(" + q.x + "," + q.y + ")");
        System.out.println(q.useCount);
        System.out.println(q.origin == Point.origin);
        System.out.println(q.origin.useCount);
    }
}
```

程序打印:

```
(2,2)
0
true
1
```

表明改变 p 的字段 x、y 和 useCount 不会影响 q 的字段，因为这些字段是不同对象中的实例变量。在本例中, 类 Point 的类变量在 Point.origin 中使用类名作为修饰符来引用，也在字段访问表达式中使用类类型的变量，如 p.origin 和 q.origin 来引用。这两种访问 origin 类变量的方法访问的是同一个对象，引用相等表达式的值(§15.21.3)就是证明:

```
q.origin==Point.origin
```

表达式的值为 true。进一步的证据是:

```
p.origin.useCount++;
```

导致 q.origin.useCount 的值为 1；这是因为 p.origin 和 q.origin 指向同样的变量。

例子 8.3.1.1-2. 隐藏类变量

```
class Point {
    static int x = 2;
}
class Test extends Point {
    static double x = 4.7;
    public static void main(String[] args) {
        new Test().printX();
    }
    void printX() {
        System.out.println(x + " " + super.x);
    }
```

```
}
```

程序产生输出：

```
4.7 2
```

因为类 Test 的声明 x 隐藏了类 Point 里 x 的定义，因此类 Test 并不从它的超类 Point 继承字段 x。在类 Test 的声明中，简单名称 x 引用类 Test 中声明的字段。类 Test 中的代码可以将类 Point 的字段 x 引用为 super.x(或者因为 x 是静态的，也可以引用为 Point.x)。如果删除了声明 Test.x:

```
class Point {
    static int x = 2;
}
class Test extends Point {
    public static void main(String[] args) {
       new Test().printX();
    }
    void printX() {
       System.out.println(x + " " + super.x);
    }
}
```

那么类 Point 的字段 x 就不再隐藏在类 Test 中；相反，简单的名称 x 现在指的是字段 Point.x。类 Test 中的代码仍然可以引用与 super.x 相同的字段。因此，该变体程序的输出为：

```
2 2
```

例子 8.3.1.1-3. 隐藏实例变量

```
class Point {
   int x = 2;
}
class Test extends Point {
   double x = 4.7;
   void printBoth() {
      System.out.println(x + " " + super.x);
   }
   public static void main(String[] args) {
      Test sample = new Test();
      sample.printBoth();
      System.out.println(sample.x + " " + ((Point)sample).x);
   }
}
```

该程序产生输出：

```
4.7 2
4.7 2
```

因为类 Test 中 x 的声明隐藏了类 Point 中 x 的定义，所以类 Test 不会从其超类 Point 继承字段 x。然而，必须注意的是，虽然类 Point 的字段 x 不是由类 Test 继承的，但它是由类 Test 的实例实现的。换句话说，类 Test 的每个实例都包含两个字段，一个是 int 类型，另一个是 double 类型。这两个字段都有名称 x，但在类 Test 的声明中，简单名称 x 总是引用类 Test 中声明的字段。类 Test 的实例方法中的代码可以将类 Point 的实例变量 x 称为 super.x。

使用字段访问表达式访问字段 x 的代码将访问由引用表达式类型指示的类中名为 x 的字段。因此，表达式 sample.x 访问类 Test 声明的 double 类型的实例变量的值， 因为变量 sample 的类型是 Test， 但是由于强制转换为类型 Point，表达式((Point)sample).x 访问 int 类型的值, 它是类 Point 中声明的实例变量。

如果从类 Test 中删除了 x 的声明，如在程序中:

```
class Point {
    static int x = 2;
}
class Test extends Point {
    void printBoth() {
       System.out.println(x + " " + super.x);
    }
    public static void main(String[] args) {
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " " + ((Point)sample).x);
    }
}
```

则类 Point 的字段 x 不再隐藏在类 Test 中。在类 Test 声明中的实例方法中，简单名称 x 现在指的是类 Point 中声明的字段。类 Test 中的代码仍然可以引用与 super.x 相同的字段。表达式 sample.x 仍然引用类型 Test 中的字段 x，但该字段现在是继承字段，因此引用在类 Point 中声明的字段 x。这个变体程序的输出是:

```
2 2
2 2
```


8.3.1.2 final 字段

字段可以被声明为 final(§4.12.4)。类和实例变量(静态和非静态字段)都可以声明为 final。

一个空的 final 类变量必须由声明它的类的静态初始化器明确赋值，否则将发生编译时错误(§8.7，§16.8)。

一个空的 final 实例变量必须在声明它的类的每个构造函数的末尾明确赋值，而且不能明确取消赋值，否则会发生编译时错误（§8.8，§16.9）。

8.3.1.3 transient 字段

变量可以标记为 transient，以指示它们不是对象持久状态的一部分。

例子 8.3.1.3-1. transient 字段的持久性

如果类 Point 的一个实例:

```
class Point {
   int x, y;
   transient float rho, theta;
}
```

被系统服务保存到持久化存储中，那么只有字段 x 和 y 会被保存。本规范没有指定此类服务的细节;请参阅 java.io.Serializable 规范，以获得此类服务的示例。

8.3.1.4 volatile 字段

Java 编程语言允许线程访问共享变量(§17.1)。作为一条规则，为了确保共享变量得到一致和可靠的更新，线程应该通过获得锁来确保独占使用这些变量，按照惯例，锁会强制对这些共享变量进行互斥。

Java 编程语言提供了第二种机制，即 volatile 字段，它在某些用途上比锁更方便。

一个字段可以被声明为 volatile，在这种情况下，Java 内存模型确保所有线程都能看到这个变量的一致值(§17.4)。

如果 final 变量也被声明为 volatile，则是编译时错误。

例子 8.3.1.4-1. volatile 字段

如果，在下面的例子中，一个线程重复调用方法 one(但不超过 Integer.MAX_VALUE 次)，另一个线程重复调用方法 two：

```
class Test {
    static int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i=" + i + " j=" + j);

}
```

然后方法 two 可以偶尔打印一个大于 i 的 j 值，因为这个例子没有包含同步，而且根据§17.4 解释的规则，i 和 j 的共享值可能会被打乱顺序更新。

防止这种无序行为的一种方法是将方法 one 和方法 two 声明为 synchronized (§8.4.3.6):

```
class Test {
    static int i = 0, j = 0;
    static synchronized void one() { i++; j++; }
    static synchronized void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

这可以防止方法 one 和方法 two 并发执行，并进一步保证 i 和 j 的共享值在方法 one 返回之前都被更新。因此，方法 two 不会发现 j 的值大于 i 的值;事实上，i 和 j 的值总是相同的。

另一种方法是声明 i 和 j 为 volatile:

```
class Test {
    static volatile int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

这允许方法 one 和方法 two 并发执行，但保证对 i 和 j 共享值的访问发生的次数和顺序完全相同，就像它们在每个线程执行程序文本时发生的顺序一样。因此，j 的共享值永远不会大于 i 的共享值，因为在更新 j 之前，对 i 的每次更新都必须反映在 i 的共享值中。这是可能的，方法 two 的任何给定调用都可能观察到 j 的值远远大于 i 的值，因为在方法 two 获取 i 的值和方法 two 获取 j 的值之间，方法 one 可能会被执行

多次。

更多的讨论和例子见§17.4。

8.3.2 字段初始化

如果字段声明中的声明符有变量初始化器，那么声明符就具有赋值(§15.26)的语义。

如果声明符用于类变量(也就是静态字段)(§8.3.1.1)，那么下面的规则适用于它的初始化器:

- 初始化器不能像§15.8.3 和§15.11.2 那样使用关键字 this 或 super 来引用当前对象，也不能像§6.5.6.1 和§15.12.3 那样使用简单名称来引用任何实例变量或实例方法。
- 在运行时，初始化器被求值，赋值只执行一次，也就是类初始化的时候(§12.4.2)。

  请注意，作为常量变量的静态字段(§4.12.4)会在其他静态字段(§12.4.2，步骤 6)之前初始化。 这也适用于接口(§9.3.1)。当这样的字段被简单的名称引用时，它们将永远不会被观察到有它们的默认初始值(§4.12.5)。

如果声明符用于实例变量(即非静态字段)，那么以下规则适用于它的初始化器:

- 初始化器可以使用关键字 this 或关键字 super 来引用当前对象，也可以使用简单名称来指向在类中声明或由类继承的任何类变量，甚至是声明在初始化器右侧的变量(§3.5)。
- 在运行时，初始化器被求值，并在每次创建类实例时执行赋值操作(§12.5)。

如§8.3.3 和§16(明确赋值)所规定的，变量初始化器对尚未初始化的字段的引用是受限制的。

§11.2.3 中规定了在字段声明中对变量初始化器进行异常检查。

变量初始化器也用于局部变量声明语句中(§14.4)，每次执行局部变量声明语句时，都要计算初始化式，并执行赋值操作。

例子 8.3.2-1. 字段初始化

```
class Point {
   int x = 1, y = 5;
}
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + ", " + p.y);
    }
}
```

这个程序产生输出:

```
1, 5
```

因为每当创建一个新的 Point 时，都会对 x 和 y 进行赋值。

例子 8.3.2-2. 类变量的前向引用

```
class Test {
   float f = j;
```

```
    static int j = 1;
}
```

这个程序编译时不会出错; 当类 Test 初始化时，它将 j 初始化为 1，每次创建类 Test 的实例时，它将 f 初始化为 j 的当前值。

8.3.3 初始化器中字段引用的限制

对字段的引用有时会受到限制，即使该字段在作用域中。下面的规则约束对字段的前向引用(使用在字段声明之前)和自引用(字段在自己的初始化器中使用)。

对于通过简单名称引用在类或接口 C 中声明的类变量 f，会产生编译错误如果:

- 引用要么出现在 C 的类变量初始化器中，要么出现在 C 的静态初始化器中(§8.7);而且
- 该引用要么出现在 f 自己声明符的初始化器中，要么出现在 f 声明符左侧的某个点；而且
- 引用并不在赋值表达式的左边(§15.26)；而且
- 包含引用的最内部的类或接口是 C。

通过简单名称引用类 C 中声明的实例变量 f，会产生编译错误如果:

- 这个引用要么出现在 C 的实例变量初始化器中，要么出现在 C 的实例初始化器中(§8.6)；而且
- 该引用出现在 f 自己声明符的初始化器中，或者出现在 f 声明符左侧的一点;而且
- 该引用不在赋值表达式的左边(§15.26); 而且
- 包含引用的最里面的类是 C。

例子 8. 3. 3-1. 字段引用的限制

此程序发生编译时错误:

```
class Test1 {
   int i = j; // compile-time error:
              // incorrect forward reference
   int j = 1;
}
```

而下面的程序编译没有错误:

```
class Test2 {
    Test2() { k = 2; }
    int j = 1;
    int i = j;
    int k;
}
```

尽管 Test2(§8.8)的构造函数引用了三行之后声明的字段 k。

上面的限制旨在在编译时捕获循环或其他格式错误的初始化。因此，以下两个程序:

```
class Z {
   static int i = j + 2;
   static int j = 4;
}
```

和:

```
class Z {
    static { i = j + 2; }
    static int i, j;
    static { j = 4; }
}
```

都产生编译错误。方法的访问不会以这种方式进行检查，因此:

```
class Z {
   static int peek() { return j; }
   static int i = peek();
   static int j = 1;
}
class Test {
    public static void main(String[] args) {
       System.out.println(Z.i);
    }
}
```

产生输出:

```
0
```

因为 i 的变量初始化器使用类方法 peek 来访问变量 j 的值，在 j 被其变量初始化器初始化之前，此时仍有其默认值（§4.12.5）。

一个更详细的例子是:

```
class UseBeforeDeclaration {
    static {
        x = 100;
          // ok - assignment
        int y = x + 1;
          // error - read before declaration
        int v = x = 3;
          // ok - x at left hand side of assignment
        int z = UseBeforeDeclaration.x * 2;
          // ok - not accessed via simple name

        Object o = new Object() {
            void foo() { x++; }
              // ok - occurs in a different class
            { x++; }
              // ok - occurs in a different class
        };
    }
```

```
    {
        j = 200;
          // ok - assignment
        j = j + 1;
          // error - right hand side reads before declaration
        int k = j = j + 1;
          // error - illegal forward reference to j
        int n = j = 300;
          // ok - j at left hand side of assignment
        int h = j++;
          // error - read before declaration
        int l = this.j * 3;
          // ok - not accessed via simple name

        Object o = new Object() {
            void foo(){ j++; }
              // ok - occurs in a different class
            { j = j + 1; }
              // ok - occurs in a different class
        };
    }

    int w = x = 3;
       // ok - x at left hand side of assignment
    int p = x;
      // ok - instance initializers may access static fields

    static int u =
        (new Object() { int bar() { return x; } }).bar();
     // ok - occurs in a different class

    static int x;

    int m = j = 4;
      // ok - j at left hand side of assignment
    int o =
        (new Object() { int bar() { return j; } }).bar();
        // ok - occurs in a different class
    int j;
}
```


8.4 方法声明

方法声明可调用的可执行代码，将固定数量的值作为参数传递。

MethodDeclaration:
{MethodModifier} MethodHeader MethodBody

MethodHeader:
Result MethodDeclarator [Throws]
TypeParameters {Annotation} Result MethodDeclarator [Throws]

MethodDeclarator:
Identifier （ *[ReceiverParameter ,] [FormalParameterList]* ） *[Dims]*

ReceiverParameter:
{Annotation} UnannType [Identifier .] `this`

The following production from §4.3 is shown here for convenience:

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

§8.4.1 中描述了 FormalParameterList 子句，§8.4-3 中描述了 MethodModifier 子句，§4.4 中描述了 TypeParameters 子句，§5.4.5 中描述了 Result 子句，§6.4.6 中描述了 Throws 子句，§7.4.7 中描述了方法体。

MethodDeclarator 中的标识符可以用在引用方法的名称中 (§6.5.7.1, §15.12)。

方法声明的作用域和遮蔽在§6.3 和§6.4.1 中规定。

receiver 参数是实例方法或内部类构造函数的可选语法设备。对于实例方法，receiver 参数表示调用该方法的对象。对于内部类的构造函数，receiver 参数表示新构造对象的直接封闭实例。在这两种情况下，receiver 参数的存在只是为了允许在源代码中表示所表示对象的类型，以便可以对该类型进行注解（§9.7.4）。receiver 参数不是形式参数；更准确地说，它不是任何类型变量的声明（§4.12.3），它从不绑定到方法调用表达式或类实例创建表达式中作为参数传递的任何值，并且在运行时没有任何影响。

receiver 参数可以出现在实例方法的 MethodDeclarator 中，也可以出现在内部类的构造函数的 ConstructorDeclarator 中，内部类不是在静态上下文中声明的(§8.1.3)。如果 receiver 参数出现在任何其他类型的方法或构造函数中，则会发生编译时错误。

receiver 参数的类型和名称约束如下：

• 在实例方法中，receiver 参数的类型必须是声明方法的类或接口，receiver 参数的名称必须是 this；否则，将发生编译时错误。

• 在内部类的构造函数中，receiver 参数的类型必须是内部类的直接封闭类型声明的类或接口，receiver 参数的名称必须是 Identifier.this，Identifier 是类或接口的简单名称，它是内部类的直接封闭类型声明；否则，将发生编译时错误。

如果类声明体将两个具有重写等价签名的方法声明为成员(§8.4.2)，将会导致编译时错误。

返回数组的方法的声明允许将表示数组类型的部分或全部方括号对放在形式形参列表之后。支持这种语法是为了与早期版本的 Java 编程语言兼容。强烈建议在新代码中不要使用此语法。

8.4.1 形式参数

方法或构造函数的形参(如果有的话)由一组逗号分隔的形参说明符指定。每个参数说明符由一个类型(可选地前面有 final 修饰符和/或一个或多个注解)和一个标识符(可选地后面有括号)组成，标识符指定参数的名称。

如果方法或构造函数没有形式参数，也没有 receiver 参数，那么方法或构造函数的声明中将出现一对空圆括号。

FormalParameterList:
FormalParameter {, FormalParameter}

FormalParameter:
{VariableModifier} UnannType VariableDeclaratorId
VariableArityParameter

VariableArityParameter:
{VariableModifier} UnannType {Annotation} ... Identifier

VariableModifier:
Annotation `final`

为方便起见，此处显示了§8.3 和§4.3 中的以下产品：

VariableDeclaratorId:
Identifier [Dims]

Dims:
{Annotation} `[ ]` *{{Annotation}* `[ ]`*}*

方法或构造函数的形式参数可以是可变的参数数量，由类型后面的省略号表示。一个方法或构造函数最多允许一个可变参数。如果可变参数出现在参数说明符列表中除最后一个位置之外的任何位置，则为编译时错误。

在 VariablerityParameter 语法中，请注意省略号（…）是自身的标记（§3.11）。可以在它和类型之间添加空格，但出于风格的考虑，不鼓励这样做。

如果方法的最后一个形式参数是可变参数，则该方法是可变参数方法。否则，它是一个固定参数方法。

关于形式参数声明和 receiver 参数的注解修饰符的规则在§9.7.4 和§9.7.5 中规定。

如果 final 作为形式参数声明的修饰符出现多次，则这是编译时错误。

形式参数的作用域和遮蔽在§6.3 和§6.4 中有规定。

根据§6.5.6.1 的规定，从嵌套类或接口或 lambda 表达式中引用形式参数受到限制。

方法或构造函数声明两个同名的形式参数是编译时错误。(也就是说，它们的声明提到相同的标识符。)

如果在方法或构造函数体中对声明为 final 的形参赋值，则为编译时错误。

形式参数声明的类型取决于它是否为可变参数:

- 如果形式参数不是可变参数，那么如果 UnannType 和 VariableDeclaratorId 中没有出现

括号对，则声明的类型用 UnannType 表示，否则由§10.2 指定。

- 如果形式参数是可变参数，则声明的类型是§10.2 指定的数组类型。

如果一个可变参数声明的类型是一个不可具体化的元素类型(§4.7)，那么对该可变参数方法的声明将会出现一个编译时未检查的警告，除非该方法被@SafeVarargs(§9.6.4.7)注解或者该警告被@SuppressWarnings(§9.6.4.5)抑制。

调用方法或构造函数时（§15.12），每个已声明类型，在执行方法或构造函数体之前，实际参数表达式的值初始化新创建的参数变量。出现在 FormalParameter 中的标识符可以用作方法或构造函数主体中的简单名称，以引用形式参数。

可变参数方法的调用可能包含比形式参数更多的实际参数表达式。所有与可变参数前面的形式参数不对应的实际参数表达式都将被计算，结果存储到数组中，并传递给方法调用（§15.12.4.2）。

以下是实例方法和内部类构造函数中 receiver 参数的一些示例:

```
class Test {
    Test(/* ?? ?? */) {}
      // No receiver parameter is permitted in the constructor of
      // a top level class, as there is no conceivable type or name.

    void m(Test this) {}
      // OK: receiver parameter in an instance method

    static void n(Test this) {}
      // Illegal: receiver parameter in a static method

    class A {
        A(Test Test.this) {}
          // OK: the receiver parameter represents the instance
          // of Test which immediately encloses the instance
          // of A being constructed.

        void m(A this) {}
          // OK: the receiver parameter represents the instance
          // of A for which A.m() is invoked.

        class B {
            B(Test.A A.this) {}
              // OK: the receiver parameter represents the instance
              // of A which immediately encloses the instance of B
              // being constructed.

            void m(Test.A.B this) {}
              // OK: the receiver parameter represents the instance
              // of B for which B.m() is invoked.
        }
    }
}
```

B 的构造函数和实例方法表明，receiver 参数的类型可以像任何其他类型一样用限定的 TypeName 表示；但内部类的构造函数中 receiver 参数的名称必须使用封闭类的简单名称。

8.4.2 方法签名

如果两个方法或构造函数 M 和 N 具有相同的名称、相同的类型参数（如有）（§8.4.4），则 M 和 N 具有相同的签名，并且在将 N 的形式参数类型调整为 M 的类型参数后，具有相同的形式参数。

方法 m_1 的签名是方法 m_2 签名的子签名，如果：

- m_2 具有与 m_1 相同的签名
- m_1 的签名与 m_2 签名的擦除（§4.6）相同。

当 m_1 是 m_2 的子签名或 m_2 是 m_1 的子签名时，两个方法签名 m_1 和 m_2 是重写等价的。

在类中声明两个具有重写等价签名的方法是编译时错误。

例子 8.4.2-1. 重写等价签名

```
class Point {
   int x, y;
   abstract void move(int dx, int dy);
   void move(int dx, int dy) { x += dx; y += dy; }
}
```

此程序会导致编译时错误，因为它声明了两个具有相同（因此重写等价）签名的 move 方法。这是一个错误，即使其中一个声明是抽象的。

子签名的概念旨在表达两个方法之间的关系，这两个方法的签名不完全相同，但其中一个可以重写另一个。具体来说，它允许签名不使用泛型类型的方法重写该方法的任何泛型版本。这一点很重要，这样库设计者就可以独立于定义库的子类或子接口的客户端自由地泛化方法。

考虑例子：

```
class CollectionConverter {
    List toList(Collection c) {...}
}
class Overrider extends CollectionConverter {
    List toList(Collection c) {...}
}
```

现在，假设这段代码是在引入泛型之前编写的，现在 class CollectionConverter 的作者决定将代码泛化，因此：

```
class CollectionConverter {
    <T> List<T> toList(Collection<T> c) {...}
}
```

如果没有特别的特许，Overrider.toList 将不再重写 CollectionConverter.toList. 相反，该代码将是非法的。这将极大地抑制泛型的使用，因为库编写者在迁移现有代码时会犹豫不决。

8.4.3 方法修饰符

MethodModifier:
(one of)
Annotation `public protected private`
`abstract static final synchronized native strictfp`

有关方法声明的注解修饰符的规则在§9.7.4 和§9.7.5 中有规定。

如果同一关键字多次作为方法声明的修饰符出现，或者方法声明具有多个访问修饰符 public、protected 和 private（§6.6），则为编译时错误。

如果包含关键字 abstract 的方法声明也包含任何关键字 private, static, final, native, strictfp 或 synchronized，则为编译时错误。

如果包含关键字 native 的方法声明也包含 strictfp，则会出现编译时错误。

> 如果两个或多个(不同的)方法修饰符出现在一个方法声明中，习惯上(虽然不是必需的)，它们出现的顺序与上面 MethodModifier 的产品中显示的顺序一致。

8.4.3.1 abstract 方法

抽象方法声明将方法作为成员引入，提供其签名(§8.4.2)、结果(§8.4.5)和 throws 子句(§8.4.6)，但没有提供实现(§8.4.7)。不是抽象的方法可以被称为具体方法。

抽象方法 m 的声明必须直接出现在抽象类中(称为 A)，除非它出现在 enum 声明中(§8.9);否则，将发生编译时错误。

A 的每个子类如果不是抽象的(§8.1.1.1)，就必须为 m 提供一个实现，否则就会发生编译时错误。

抽象类可以通过提供另一个抽象方法声明来重写一个抽象方法。

> 这可以提供一个放置文档注释的地方，可以细化返回类型，或者声明当该方法由其子类实现时，该方法可以抛出的一组检查异常是更有限的。

非抽象的实例方法可以被抽象方法重写。

例子 8.4.3.1-1. 抽象/抽象方法重写

```
class BufferEmpty extends Exception {
    BufferEmpty() { super(); }
    BufferEmpty(String s) { super(s); }
}
class BufferError extends Exception {
    BufferError() { super(); }
    BufferError(String s) { super(s); }
}
interface Buffer {
    char get() throws BufferEmpty, BufferError;
}
abstract class InfiniteBuffer implements Buffer {
    public abstract char get() throws BufferError;
}
```

类 Infinitbuffer 中 get 方法的重写声明指出，在 Infinitbuffer 的任何子类中 get 方法永远不会抛出 BufferEmpty 异常，这可能是因为它在缓冲区中生成数据，因此永远不会耗尽数据。

例子 8.4.3.1-2. 抽象/非抽象的重写

我们可以声明一个抽象类 Point，要求它的子类实现 toString 方法，如果它们是完整的、可实例化的类：

```
abstract class Point {
   int x, y;
   public abstract String toString();
}
```

toString 的 abstract 声明重写类 Object 的非 abstract 的 toString 方法 (Object 是类 Point 的隐式直接超类。) 添加如下代码:

```
class ColoredPoint extends Point {
    int color;
    public String toString() {
        return super.toString() + ": color " + color; // error
    }
}
```

产生编译错误，因为调用 super.toString()引用了类 Point 的方法 toString, 该方法是 abstract 方法，不能被调用。只有当类 Point 通过其他方法显式地使其可用时，类 Object 的 toString 方法才能对类 ColoredPoint 可用，例如：

```
abstract class Point {
    int x, y;
    public abstract String toString();
    protected String objString() { return super.toString(); }
}
class ColoredPoint extends Point {
    int color;
    public String toString() {
            return objString() + ": color " + color; // correct
    }
}
```


8.4.3.2 static 方法

声明为静态的方法称为类方法。

类方法总是在不引用特定对象的情况下调用。类方法的声明引入了静态上下文（§8.1.3），这限制了引用当前对象的构造的使用。值得注意的是，在静态上下文中禁止使用 this 和 super 关键字（§15.8.3，§15.11.2），因为它们是对实例变量、实例方法和词法封装声明的类型参数的非限定引用（§6.5.5.1，§6.6.1，§15.12.3）。

未声明为静态的方法称为实例方法，有时称为非静态方法。

实例方法总是针对对象调用，该对象在方法体执行期间成为关键字 this 和 super 引用的当前对象。

根据§15.12.3 的规定，从静态上下文或嵌套类或接口引用实例方法受到限制。

8.4.3.3 final 方法

方法可以声明为 final，以防止子类重写或隐藏它。

试图重写或隐藏 final 方法是编译时错误。

私有方法和直接在 final 类中声明的所有方法（§8.1.1.2）的行为就像它们是 final 的一样，因为不可能重写它们。

在运行时，机器码生成器或优化器可以“内联”final 方法的主体，将方法调用替换为其主体中的代码。内联过程必须保留方法调用的语义。 特别是，如果实例方法调用的目标为 null，则即使该方法是内联的，也必须抛出 NullPointerException。Java 编译器必须确保将在正确的点引发异常，以便在方法调用之前，可以看到方法的实际参数已按正确的顺序进行了计算。

举个例子：

```
final class Point {
   int x, y;
   void move(int dx, int dy) { x += dx; y += dy; }
}
class Test {
    public static void main(String[] args) {
        Point[] p = new Point[100];
        for (int i = 0; i < p.length; i++) {
           p[i] = new Point();
           p[i].move(i, p.length-1-i);
        }
    }
}
```

在方法 main 中内联 Point 类的方法 move 会将 for 循环转换为以下形式：

```
for (int i = 0; i < p.length; i++) {
     p[i] = new Point();
     Point pi = p[i];
     int j = p.length-1-i;
     pi.x += i;
     pi.y += j;
}
```

然后，可能会对循环进行进一步优化。

这种内联不能在编译时完成，除非可以保证 Test 和 Point 总是一起重新编译，这样每当 Point -特别是它的 move 方法-发生变化时，Test.main 的代码也将更新。

8.4.3.4 native 方法

native 方法是在依赖于平台的代码中实现的，通常用另一种编程语言（如 C）编写。native 方法的主体仅以分号表示，而不是块，表示省略了实现，(§8.4.7）。

例如，包 java.io 的类 RandomAccessFile 声明以下的 native 方法：

```
package java.io;
public class RandomAccessFile
    implements DataOutput, DataInput {
    ...
    public native void open(String name, boolean writeable)
       throws IOException;
    public native int readBytes(byte[] b, int off, int len)
       throws IOException;
    public native void writeBytes(byte[] b, int off, int len)
       throws IOException;
```

```
    public native long getFilePointer() throws IOException;
    public native void seek(long pos) throws IOException;
    public native long length() throws IOException;
    public native void close() throws IOException;
}
```

8.4.3.5 strictfp 方法

方法声明上的 strictfp 修饰符已过时，不应在新代码中使用。它的存在或不存在在运行时没有影响。

8.4.3.6 synchronized 方法

同步方法在执行之前获取监视器（§17.1）。

对于类(静态)方法，将使用与方法的类的 Class 对象关联的监视器。

对于实例方法，将使用与 this 关联的监视器(调用该方法的对象)。

例子 8.4.3.6-1. synchronized 监视器

这些监视器也可以被同步语句使用(§14.19)。

因此，代码:

```
class Test {
   int count;
   synchronized void bump() {
      count++;
   }
   static int classCount;
   static synchronized void classBump() {
      classCount++;
   }
}
```

和以下代码效果相同:

```
class BumpTest {
   int count;
   void bump() {
      synchronized (this) { count++; }
   }
   static int classCount;
   static void classBump() {
      try {
         synchronized (Class.forName("BumpTest")) {
         classCount++;
         }
      } catch (ClassNotFoundException e) {}
   }
}
```

例子 8.4.3.6-2. synchronized 方法

```
public class Box {
    private Object boxContents;
    public synchronized Object get() {
         Object contents = boxContents;
```

```
        boxContents = null;
        return contents;
    }
    public synchronized boolean put(Object contents){
        if (boxContents != null) return false;
        boxContents = contents;
        return true;
    }
}
```

这个程序定义了一个用于并发使用的类。类 Box 的每个实例都有一个实例变量 boxContents，该变量可以保存对任何对象的引用。您可以通过调用 put 将对象放入 Box 中，如果 Box 已满，则返回 false。你可以通过调用 get 从 Box 中获取某些内容，如果 Box 为空，则 get 返回一个空引用。

如果 put 和 get 不是同步的，并且两个线程同时执行 Box 的同一个实例的方法，那么代码可能会出错。例如，它可能会因为要放置的两个调用同时发生而失去对对象的跟踪。

8.4.4 泛型方法

如果一个方法声明了一个或多个类型变量，它就是泛型的(§4.4)。

这些类型变量称为方法的类型参数。泛型方法的类型参数部分的形式与泛型类的类型参数部分相同(§8.1.2)。

泛型方法声明定义了一组方法，每个方法对应一个类型参数部分的可能调用。当调用泛型方法时，类型参数可能不需要显式提供，因为它们通常可以被推断(§18(类型推断))。

方法的类型参数的作用域和遮蔽在§6.3 和§6.4.1 中有规定。

从嵌套类或接口引用方法的类型参数是受限制的，如§6.5.5.1 所述。

如果以下两个都为真，两个方法或构造函数 M 和 N 具有相同的类型参数:

- M 和 N 具有相同数量的类型参数(可能为零)。
- 其中 $A_1, ..., A_n$ 是 M 的类型参数，$B_1, ..., B_n$ 是 N 的类型参数，设 $\theta = [B_1:=A_1,\cdots,B_n:=A_n]$。那么，对于所有 i ($1 \leqslant i \leqslant n$)，$A_i$ 的边界与应用于 B_i 边界的 θ 的类型相同。

如果两种方法或构造函数 M 和 N 具有相同的类型参数，则 N 中提到的类型可以通过对类型应用 θ（如上所述）来适应 M 的类型参数。

8.4.5 方法的结果

方法声明的结果要么声明该方法返回的值的类型(返回类型)，要么使用关键字 void 来表明该方法不返回值。

Result:
  UnannType
  `void`

如果结果不是 void，那么如果形式形参列表后没有出现括号对，方法的返回类型就用

UnannType 表示，否则由§10.2 指定。

如果返回类型是引用类型，则返回类型可能因重写彼此的方法而异。返回类型可替换性的概念支持协变返回，即返回类型对子类型的特殊化。

返回类型为 R_1 的方法声明 d_1 可以返回类型替换另一个返回类型为 R_2 的方法 d_2，当且仅当以下任一条件成立:

- 如果 R_1 是 void 那么 R_2 是 void。
- 如果 R_1 是原生类型，那么 R_2 与 R_1 相同。
- 如果 R_1 是引用类型，那么下面一个是真:
  - R_1 适应于 d_2(§8.4.4)的类型参数，是 R_2 的一个子类型。
  - R_1 可以通过未检查的转换转换为 R_2 的子类型(§5.1.9)。
  - d_1 和 d_2 的签名不同(§8.4.2), 并且 $R_1 = |R_2|$。

定义中允许进行未检查的转换(尽管不合理)，作为一种特殊允许，允许从非泛型代码平滑迁移到泛型代码。如果使用未检查的转换来确定 R1 对 R2 是返回类型可替换的，那么 R1 必然不是 R2 的子类型，并且重写规则(§8.4.8.3，§9.4.1)将需要一个编译时未检查的警告。

8.4.6 方法抛出异常

throws 子句用来表示任何被检查的异常类(§11.1.1)，方法或构造函数体中的语句可以抛出这些异常(§11.2.2)。

Throws:
`throws` *ExceptionTypeList*

ExceptionTypeList:
ExceptionType {，ExceptionType}

ExceptionType:
ClassType
TypeVariable

如果在 throws 子句中提到的 ExceptionType 不是 Throwable 的子类型(§4.10)，则会出现编译时错误。

类型变量在 throws 子句中是允许的，尽管在 catch 子句中是不允许的(§14.20)。

允许但不要求在 throws 子句中提及未检查的异常类(§11.1.1)。

throws 子句和方法或构造函数体的异常检查之间的关系见§11.2.3。

从本质上讲，对于每个检查的异常(可能是由于执行方法或构造函数体导致的)，除非在方法或构造函数声明的 throws 子句中提到异常类型或异常类型的超类型，否则将发生编译时错误。

声明检查异常的要求允许 Java 编译器确保包含了处理此类错误条件的代码。如果方法或构造函数的

throws 子句中缺少适当的异常类型，那么无法处理作为检查异常抛出的异常条件的方法或构造函数通常会导致编译时错误。因此，Java 编程语言鼓励采用一种编程风格，以这种方式记录罕见的或真正异常的情况。

§8.4.8.3 规定了方法的 throws 子句和被重写方法或隐藏方法的 throws 子句之间的关系。

例子 8.4.6-1. 类型变量作为抛出的异常类型

```
import java.io.FileNotFoundException;

interface PrivilegedExceptionAction<E extends Exception> {
   void run() throws E;
}
class AccessController {
    public static <E extends Exception>
    Object doPrivileged(PrivilegedExceptionAction<E> action) throws E{
       action.run();
       return "success";
    }
}
class Test {
    public static void main(String[] args) {
      try {
          AccessController.doPrivileged(
             new PrivilegedExceptionAction<FileNotFoundException>() {
                public void run() throws FileNotFoundException {
                          // ... delete a file ...
                }
             });
         } catch (FileNotFoundException f) { /* Do something */ }
    }
  }
```

8.4.7 方法体

方法主体要么是实现该方法的代码块，要么只是表示缺乏实现的分号。

MethodBody:
  Block

如果方法是 abstract 或 native，方法体必须是分号(§8.4.3.1，§8.4.3.4)。更准确地说应该是：

- 如果方法声明是 abstract 或 native，并且有一个块作为其主体，则为编译时错误。
- 如果方法声明既不是 abstract，也不是 native，并且其主体有分号，则为编译时错误。

  如果要为声明为 void 的方法提供实现，但该实现不需要可执行代码，那么方法体应该写为不包含语句的块:"{}"。

return 语句在方法体中的规则见§14.17。

如果一个方法被声明为有返回类型(§8.4.5)，那么如果该方法体可以正常完成(§14.1)，则会

发生编译时错误。

换句话说，具有返回类型的方法只能通过使用提供返回值的 return 语句返回;该方法不允许“脱离其主体的末端”。关于方法体中返回语句的精确规则见§14.17。

方法有返回类型，但不包含返回语句是可能的。举个例子:

```
class DizzyDean {
    int pitch() { throw new RuntimeException("90 mph?!"); } }
```

8.4.7 继承、重写和隐藏

类 C 从它的直接超类类型 D 继承所有具体方法 m(包括静态方法和实例方法)，以下所有都为真:

- m 是 D 的成员。
- m 是公共的，受保护的，或声明包访问与 C 在同一个包中。
- C 中声明的任何方法的签名都不是 m 作为 D 成员的签名的子签名（§8.4.2）。

类 C 从其直接超类类型和直接超接口类型继承所有抽象和默认（§9.4）方法 m，对于这些方法，以下所有条件均为真:

- m 是 C 的直接超类类型或直接超接口类型的成员，在任何情况下都称为 D。
- m 是公共的、受保护的，或者在与 C 相同的包中通过包访问来声明。
- C 中声明的任何方法的签名都不是 m 作为 D 的成员的签名的子签名（§8.4.2）。
- C 从其直接超类类型继承的任何具体方法都没有签名是作为 D 成员的 m 签名的子签名。
- 不存在属于直接超类类型或直接超接口类型 C，D'（m 不同于 m'，D 不同于 D'）的成员的方法 m'，因此 m'重写了 D'的类或接口对方法 m 的声明（§8.4.8.1，§9.4.1.1）。

类不会从其超接口类型继承私有或静态方法。

请注意，在逐个签名的基础上重写或隐藏方法。例如，如果一个类声明了两个同名的公共方法（§8.4.9），并且一个子类重写了其中一个，则该子类仍然继承另一个方法。

例子 8.4.8-1. 继承

```
interface I1 {
   int foo();
}

interface I2 {
   int foo();
}

abstract class Test implements I1, I2 {}
```

在这里，抽象类 Test 继承了接口 I1 的抽象方法 foo 和接口 I2 的抽象方法 foo。确定从 I1 继承 foo 的关键问题是：I2 中的方法 foo 是否重写了“从 I2”（§9.4.1.1）I1 中的方法 foo？不，因为 I1 和 I2 不是彼此的子接口。因此，从类 Test 的角度来看，从 I1 继承 foo 是不受限制的；类似地，从 I2 继承 foo 也是一样。根据 §8.4.8.4，类 Test 可以继承两种 foo 方法；显然，它必须被声明为 abstract，或者用具体的方法重写两个 abstract foo 方法。

请注意，继承的具体方法可以防止继承抽象或默认方法。(根据§8.4.8.1 和§9.4.1.1，具体方法将重写“来自 C”的抽象或默认方法。）此外，如果一个超类型方法“已经”重写了另一个超类型方法，则该超类型方法可能会阻止继承另一个超级类型方法-这与接口规则（§9.4.1）相同，并防止多个默认方法被继承而一个实现显然要取代另一个实现的冲突。

8.4.8.1 重写（通过实例方法）

在类 C 中声明或由类 C 继承的实例方法 m_C 重写了在类 A 中声明的另一个方法 m_A，当且仅当以下条件都为真：

- C 是 A 的子类。
- C 不继承 m_A。
- m_C 的签名是 m_A 的签名的子签名（§8.4.2），m_A 是命名为 A 的 C 超类型的成员。
- 以下之一为真：
  - m_A 是公共的。
  - m_A 是受保护的。
  - m_A 在与 C 相同的包中使用包访问声明，C 声明的 m_C 或 m_A 是 C 的直接超类类型的成员。
  - m_A 通过包访问来声明，m_C 从 C 的某个超类重写 m_A。
  - m_A 是通过包访问来声明的，m_C 重写了 C 中的方法 m'（m'不同于 m_C 和 m_A），因此 m'重写了 C 的某个超类中的 m_A。

如果 m_C 是非抽象的，并且从 C 重写了一个抽象方法 m_A，那么 m_C 就称作从 C 实现 m_A。

如果重写的方法 m_A 是静态方法，则这是编译时错误。

在这方面，重写方法不同于隐藏字段（§8.3），因为允许实例变量隐藏静态变量。

在类 C 中声明或由类 C 继承的实例方法 m_C 重写了在接口 I 中声明的另一个方法 m_I, 当且仅当以下为真：

- I 是 C 的超接口。
- m_I 不是静态方法。
- C 不继承 m_I。
- m_C 的签名是 m_I 的签名的子签名（§8.4.2），m_I 是命名为 I 的 C 的超类型的成员。

- m_I是公共的。

如果其中一个方法的形式参数具有原始类型，而另一个方法中的相应参数具有参数化类型，则重写方法的签名可能不同于被重写方法。这允许迁移预先存在的代码以利用泛型。

重写的概念包括从其声明类的某个子类重写另一个的方法。这可以通过两种方式实现：

- 在某些参数化下，泛型超类中的具体方法可以与该类中的抽象方法具有相同的签名。在这种情况下，具体方法是继承的，而抽象方法不是（如上所述）。然后应该考虑将继承的方法重写其来自 C 的抽象对等体。(这个场景由于包访问而变得复杂:如果 C 在不同的包中，那么无论如何 m_A 都不会被继承，并且不应该被认为是被重写的。)
- 从类继承的方法可以重写超接口方法。(幸运的是，包访问在这里并不重要。)

可以通过使用包含关键字 super 的方法调用表达式(§15.12)来访问被重写的方法。在试图访问被重写的方法时，限定名或强制转换为超类类型是无效的。

在这方面，重写方法不同于隐藏字段。

strictfp 修饰符的存在或不存在绝对不会影响重写方法和实现抽象方法的规则。例如，允许不是 strictfp 的方法重写 strictfp 方法，也允许 strictfp 方法重写不是 strictfp 的方法。

例子 8.4.8.1-1. 重写

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class SlowPoint extends Point {
    int xLimit, yLimit;
    void move(int dx, int dy) {
        super.move(limit(dx, xLimit), limit(dy, yLimit));
    }
    static int limit(int d, int limit) {
            return d > limit ? limit : d < -limit ? -limit : d;
    }
}
```

在这里，类 SlowPoint 用它自己的 move 方法重写了类 Point 的方法 move 的声明，这限制了每次调用方法时点可以移动的距离。当为类 SlowPoint 的实例调用 move 方法时，将始终调用类 SlowPoint 中的重写定义，即使对 SlowPoint 对象的引用取自类型为 Point 的变量。

例子 8.4.8.1-2. 重写

重写使子类很容易扩展现有类的行为，如下所示:

```
import java.io.IOException;
import java.io.OutputStream;

class BufferOutput {
    private OutputStream o;
    BufferOutput(OutputStream o) { this.o = o; }
    protected byte[] buf = new byte[512];
    protected int pos = 0;
    public void putchar(char c) throws IOException {
       if (pos == buf.length) flush();
```

```
        buf[pos++] = (byte)c;
    }
    public void putstr(String s) throws IOException {
        for (int i = 0; i < s.length(); i++)
            putchar(s.charAt(i));
    }
    public void flush() throws IOException {
        o.write(buf, 0, pos);
        pos = 0;
    }
}
class LineBufferOutput extends BufferOutput{
  LineBufferOutput(OutputStream o) { super(o); } public
  void putchar(char c) throws IOException{
     super.putchar(c);
     if (c == '\n') flush(); }
  }
}
class Test {
    public static void main(String[] args) throws IOException{
        LineBufferOutput lbo = new LineBufferOutput(System.out);
        lbo.putstr("lbo\nlbo");
        System.out.print("print\n");
        lbo.putstr("\n");
    }
}
```

程序产生输出:

```
lbo
print lbo
```

BufferOutput 类实现了一个非常简单的 OutputStream 的缓冲版本，当缓冲区满或调用 flush 时刷新输出。子类 LineBufferOutput 只声明了一个构造函数和一个方法 putchar，该方法重写了 BufferOutput 的 putchar 方法。它从类 BufferOutput 继承了方法 putstr 和 flush。

在 LineBufferOutput 对象的 putchar 方法里，如果字符参数是换行符，则它调用 flush 方法。在这个例子中，关于重写的关键点在于，在 BufferOutput 类中声明的方法 putstr 调用当前对象 this 定义的 putchar 方法，而这个 putchar 方法不一定是在 BufferOutput 类中声明的 putchar 方法。

因此，当使用 LineBufferOutput 对象 lbo 在 main 中调用 putstr 时，putstr 方法体中对 putchar 的调用是对对象 lbo 的 putchar 的调用，putchar 的重写声明检查换行符。这允许 BufferOutput 的子类更改 putstr 方法的行为，而无需重新定义它。

像 BufferOutput 这样的类的文档(它被设计为扩展的)应该清楚地表明类和它的子类之间的契约是什么，并且应该清楚地表明子类可以以这种方式重写 putchar 方法。因此，BufferOutput 类的实现者不会希望在 BufferOutput 的未来实现中更改 putstr 的实现，而不使用 putchar 方法，因为这将打破与子类之间已有的约定。参见§13(二进制兼容性)中对二进制兼容性的讨论，特别是§13.2。

8.4.8.2 隐藏 (通过类方法)

如果一个类 C 声明或继承了一个静态方法 m，那么 m 被认为隐藏了在类或接口 A 中声明

的任何方法 m'，这些方法的以下所有条件都为真：

- A 是 C 的超类或超接口。
- 如果 A 是一个接口，m' 是一个实例方法。
- m'对 C 可达(§6.6)。
- m 的签名是 m'签名的子签名(§8.4.2)，是命名为 A 的 C 的超类型的成员。

如果静态方法隐藏实例方法，则为编译时错误。

在这方面，隐藏方法不同于隐藏字段(§8.3)，因为静态变量可以隐藏实例变量。隐藏也不同于遮蔽(§6.4.1)和遮掩(§6.4.2)。

隐藏方法可以通过使用限定名或使用包含关键字 super 或强制转换为超类类型的方法调用表达式(§15.12)来访问。

在这方面，隐藏方法类似于隐藏字段。

例子 8.4.8.2-1. 隐藏类方法的调用

隐藏的类(静态)方法可以通过使用其类型为实际包含方法声明的类的类型的引用来调用。在这方面，隐藏静态方法不同于重写实例方法。例如：

```
class Super {
    static String greeting() { return "Goodnight"; } String
    name() { return "Richard"; }
}
class Sub extends Super {
    static String greeting() { return "Hello"; }
    String name() { return "Dick"; }
}
class Test {
    public static void main(String[] args) {
        Super s = new Sub();
        System.out.println(s.greeting() + ", " + s.name());
    }
}
```

产生输出：

```
Goodnight, Dick
```

因为 greeting 的调用使用 s 的类型，即 Super，在编译时找出调用哪个类方法，而 name 的调用使用 s 的类，即 Sub，在运行时找出调用哪个实例方法。

8.4.8.3 重写和隐藏的需求

如果返回类型为 R_1 的方法声明 d_1 重写或隐藏另一个返回类型为 R_2 的方法 d_2 的声明，则 d_1 必须是 d_2 的可替换返回类型（§8.4.5），否则会发生编译时错误。

此规则允许协变返回类型-在重写方法时优化方法的返回类型。

如果 R_1 不是 R_2 的子类型，则会出现编译时未检查警告，除非@SuppressWarnings（§9.6.4.5）抑制了该警告。

重写或隐藏另一个方法的方法（包括实现接口中定义的抽象方法的方法）不能声明为引发比重写或隐藏方法更多的检查异常。

在这方面，重写方法不同于隐藏字段（§8.3），因为允许字段隐藏另一种类型的字段。

更准确地说，假设 B 是类或接口，A 是 B 的超类或超接口，B 中的方法声明 m_2 重写或隐藏 A 中的方法声明 m_1。那么：

- 如果 m_2 有一个提到任何已检查异常类型的 throws 子句，那么 m_1 必须有一个 throws 子句，否则会发生编译时错误。
- 对于 m_2 的 throws 子句中列出的每个已检查异常类型，同一异常类型或其超类型之一必须发生在 m_1 的 throws 子句的擦除（§4.6）中；否则，将发生编译时错误。
- 如果 m_1 的未擦除 throws 子句不包含在 m_2 的 throws 子句中的每个异常类型的超类型（如有必要，根据 m_1 的类型参数进行调整），则会出现编译时未经检查的警告，除非被@SuppressWarnings（§9.6.4.5）抑制。

如果一个类或接口 C 有一个成员方法 m_1，并且存在一个在 C 中声明的方法 m_2 或为 C，A 的超类或超接口，则这是一个编译时错误，因此以下所有条件均为真：

- m_1 和 m_2 有相同的名字。
- m_2 对 C 可达(§6.6) 。
- m_1 的签名不是 m_2 签名的子签名（§8.4.2），m_2 是命名为 A 的 C 超类型的成员。
- m_1 或某些方法 m_1 重写的声明签名（直接或间接）与 m_2 或某些方法 m_2 重写（直接或直接间接）的声明签名具有相同的擦除。

这些限制是必要的，因为泛型是通过擦除实现的。上面的规则意味着在同一类中声明的具有相同名称的方法必须具有不同的擦除。它还意味着类或接口不能实现或扩展同一泛型接口的两个不同参数化。

重写或隐藏方法的访问修饰符必须至少提供与重写或隐藏的方法相同的访问，如下所示：

- 如果被重写或隐藏的方法是公共的，则重写或隐藏方法必须是公共的；否则，将发生编译时错误。
- 如果被重写或隐藏的方法受保护，则重写或隐藏方法必须受保护或公开；否则，将发生编译时错误。
- 如果被重写或隐藏的方法具有包访问权限，则重写或隐藏方法不能是私有的；否则，将发生编译时错误。

请注意，在这些术语的技术意义上，私有方法不能被重写或隐藏。这意味着子类可以在其一个超类中声明具有与私有方法相同签名的方法，并且不要求此类方法的返回类型或 throws 子句与超类中的私有方法

具有任何关系。

例子 8.4.8.3-1. 协变返回类型

以下声明在 Java SE 5.0 以后的 Java 编程语言中是合法的:

```
class C implements Cloneable {
    C copy() throws CloneNotSupportedException{
       return (C)clone();
    }
}
class D extends C implements Cloneable {
    D copy() throws CloneNotSupportedException{
       return (D)clone();
    }
}
```

重写的宽松规则还允许放宽实现接口的抽象类的条件。

例子 8.4.8.3-2. 返回类型的未检查警告

考虑:

```
class StringSorter {
    // turns a collection of strings into a sorted list
    List toList(Collection c) {...}
}
```

并假设某人继承了 StringSorter:

```
class Overrider extends StringSorter {
    List toList(Collection c) {...}
}
```

现在，StringSorter 的作者在某个时候决定泛化代码:

```
class StringSorter {
    // turns a collection of strings into a sorted list List<String>
    toList(Collection<String> c) {...}
}
```

当根据 StringSorter 的新定义编译 Overrider 时，会给出未检查的警告，因为 Overrider.toList 的返回类型是 List，它不是被重写的方法 List<String>的返回类型的子类型。

例子 8.4.8.3-3. 由于 throws 不正确的重写

该程序在 BadPointException 类的声明中使用了通常和常规的形式来声明新的异常类型:

```
class BadPointException extends Exception {
  BadPointException() { super(); }
  BadPointException(String s) { super(s); }
}
class Point {
  int x, y;
```

```
    void move(int dx, int dy) { x += dx; y += dy; }
}
class CheckedPoint extends Point {
    void move(int dx, int dy) throws BadPointException {
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

程序会导致编译时错误，因为类 CheckedPoint 中方法 move 的重写声明它将引发一个检查异常因为类 Point 里的 move 没有声明。如果这不被视为错误，则在引用类型 Point 上调用方法 move 将会发现如果抛出此异常，它和 Point 之间的契约已被破坏。

删除 throws 子句没有帮助:

```
class CheckedPoint extends Point {
    void move(int dx, int dy) {
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

现在发生了另一个编译时错误，因为方法 move 的主体无法抛出检查异常，即 BadPointException，该异常没有出现在 move 的 throws 子句中。

例子 8.4.8.3-4. 擦除影响重写

一个类不能有两个具有相同名称和类型擦除的成员方法:

```
class C<T> {
    T id (T x) {...}
}
class D extends C<String> {
    Object id(Object x) {...}
}
```

这是非法的因为 D.id(Object)是 D 的成员, C<String>.id(String)声明为 D 的超类型, 并且:

- 两个方法有同样的名字 id
- C<String>.id(String)对 D 可达
- D.id(Object)的签名不是 C<String>.id(String)的子签名
- 两个方法擦除相同

一个类的两个不同方法不能使用相同的擦除重写方法:

```
class C<T> {
    T id(T x) {...}
}
interface I<T> {
    T id(T x);
}
class D extends C<String> implements I<Integer> {
    public String id(String x) {...}
```

```
    public Integer id(Integer x) {...}
}
```

这也是非法的，因为 D.id(String)是 D 的成员, D.id(Integer)声明在类 D 中, 并且:

- 两个方法有同样的名字 id
- D.id(Integer) 对 D 可达
- 两个方法有同样的签名 (两者都不是互为子签名)
- D.id(String)重写 C<String>.id(String)并且 D.id(Integer)重写 I.id(Integer)然而两个被重写的方法有同样的擦除

8.4.8.4 继承具有重写等价签名的方法

一个类可以继承具有重写等价签名的多个方法（§8.4.2）。

如果类 C 继承了一个具体方法，该方法的签名与 C 继承的另一个方法重写等价，则这是一个编译时错误。

如果类 C 继承了一个默认方法，其签名与 C 继承的另一个方法重写等价，则这是编译时错误，除非存在一个在 C 的超类中声明并由 C 继承的抽象方法，该抽象方法与这两个方法重写等价。

当在超类中声明抽象方法时，就会出现严格的 default-abstract 和 default-default 冲突规则的这种例外:来自超类层次结构的抽象性断言本质上胜过默认方法，使得默认方法的行为就像它是抽象的一样。但是，来自类的抽象方法不重写默认方法，因为接口仍然允许提炼来自类层次结构的抽象方法的签名。

注意，如果 C 继承的所有重写等价抽象方法都在接口中声明，则不适用此异常。

否则，重写等价方法的集合至少包含一个抽象方法和零个或多个默认方法;那么这个类就必须是一个抽象类，并且被认为继承了所有的方法。

继承的方法中必须有一个对其他继承的方法具有返回类型可替换性;否则，将发生编译时错误。(在这种情况下，throws 子句不会导致错误。)

从接口继承相同方法声明可能有多个路径。这一事实不会造成任何困难，而且本身也不会导致编译时错误。

例子 8.4.8.4-1. 重写等价方法的继承

上面的第一个编译时错误是关于继承了具体方法的类 C，如果 C 的超类是泛型的，并且超类有两个方法，它们在泛型声明中是不同的，但在 C 使用的参数化(§4.5)中有相同的签名，就会发生。例如:

```
class A<T> {
    void m(String s) {} // 1
    void m(T t) {} // 2
}
class C extends A<String> {}
```

C 从它的直接超类类型 A<String>继承了两个方法: 方法 m(String)标记为 1, 并且（由于 C 对 A 的参数化）方法 m(String)标记为 2。这些方法有相同的签名，因此相互之间是重写等效的。

8.4.9 重载

如果一个类的两个方法(无论是在同一个类中声明，还是都被一个类继承，或者一个声明一个继承)具有相同的名称，但签名不是重写等价，那么该方法名被称为重载。

这个事实不会造成任何困难，本身也不会导致编译时错误。返回类型之间或具有相同名称的两个方法的 throws 子句之间没有必要的关系，除非它们的签名是重写等价的。

当一个方法被调用时(§15.12)，实际参数的数量(以及任何显式的类型参数)和参数的编译时类型在编译时被使用，以确定将要被调用的方法的签名(§15.12.2)。如果要调用的方法是实例方法，那么实际要调用的方法将在运行时使用动态方法查找来确定(§15.12.4)。

例子 8.4.9-1. 重载

```
class Point {
   float x, y;
   void move(int dx, int dy) { x += dx; y += dy; }
   void move(float dx, float dy) { x += dx; y += dy; }
   public String toString() { return "("+x+","+y+")"; }
}
```

在这里，类 Point 有两个成员，它们是同名的方法 move。为任何特定方法调用而选择的类 Point 的重载 move 方法在编译时由§15.12 中给出的重载解析过程确定。

总的来说，类 Point 的成员是在 Point 中声明的浮点实例变量 x 和 y，声明的两个 move 方法，声明的 toString 方法，以及 Point 从其隐式直接超类 Object 继承的成员(§4.3.2)，例如 hashCode 方法。注意，Point 没有继承类 Object 的 toString 方法，因为该方法被类 Point 中的 toString 方法声明重写了。

例子 8.4.9-2. 重载，重写和隐藏

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
    int color;
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
}
```

在这里，类 RealPoint 用它自己的浮点实例变量 x 和 y 隐藏了类 Point 的 int 实例变量 x 和 y 的声明，并用它自己的 move 方法重写了类 Point 的 move 方法。它还使用另一个具有不同签名的方法重载 move 方法(§8.4.2)。

在这个例子中，类 RealPoint 的成员包括继承自类 Point 的实例变量 color，在 RealPoint 中声明的浮点实例变量 x 和 y，以及在 RealPoint 中声明的两个 move 方法。

对于任何特定的方法调用，在编译时由§15.12 中描述的重载解析过程来决定将选择哪些类 RealPoint 的重载 move 方法。

下面这个程序是前一个程序的扩展变体:

```
class Point {
    int x = 0, y = 0, color;
```

```
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; } int getY() { return y; }
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    float getX() { return x; }
    float getY() { return y; }
}
```

在这里，类 Point 提供了 getX 和 getY 方法，它们返回字段 x 和 y 的值;类 RealPoint 然后通过声明具有相同签名的方法重写这些方法。结果是在编译时产生两个错误，每个方法一个错误，因为返回类型不匹配;Point 类中的方法返回 int 类型的值，但 RealPoint 类中想要重写的方法返回 float 类型的值。

此程序纠正前一程序的错误:

```
class Point {
   int x = 0, y = 0;
   void move(int dx,int dy) {x += dx;y += dy; }
   int getX() {return x;}
   int getY() {return y;}
   int color;
}
class RealPoint extends Point {
   float x = 0.0f, y = 0.0f;
   void move(int dx,int dy) {move((float)dx, (float)dy);}
   void move(float dx, float dy) {x += dx; y += dy;}
   int getX() {return (int)Math.floor(x); }
   int getY() {return (int)Math.floor(y); }
}
```


在这里，类 RealPoint 中的重写方法 getX 和 getY 与它们重写的类 Point 的方法具有相同的返回类型，因此可以成功编译此代码。

那么，考虑一下这个测试程序:

```
class Test {
    public static void main(String[] args) {
       RealPoint rp = new RealPoint();
       Point p = rp;
       rp.move(1.71828f, 4.14159f);
       p.move(1, -1);
       show(p.x, p.y);
       show(rp.x, rp.y);
       show(p.getX(), p.getY());
       show(rp.getX(), rp.getY());
    }
    static void show(int x, int y) {
       System.out.println("(" + x + ", " + y + ")");
    }
    static void show(float x, float y) {
       System.out.println("(" + x + ", " + y + ")");
    }
}
```

该程序的输出为:

```
(0, 0)
(2.7182798, 3.14159)
(2, 3)
(2, 3)
```

第一行输出说明了 RealPoint 的一个实例实际上包含在 Point 类中声明的两个整数字段；只是在 RealPoint 类（以及它可能拥有的任何子类）声明中出现的代码中隐藏了它们的名称。当使用 Point 类型变量中对 RealPoint 类实例的引用访问字段 x 时，将访问在 Point 类中声明的整数字段 x。其值为零的事实表明方法调用 p.move（1，-1）没有调用 Point 类的 move 方法；相反，它调用了 RealPoint 类的重写 move 方法。

第二行输出显示字段访问 rp.x 引用 RealPoint 类中声明的字段 x。此字段为浮点类型，第二行输出相应地显示浮点值。顺便说一句，这也说明了方法名 show 被重载的事实；方法调用中的参数类型决定将调用这两个定义中的哪一个。

最后两行输出显示，方法调用 p.getX（）和 rp.getX（），每个调用在类 RealPoint 中声明的 getX 方法。事实上，对于类 RealPoint 的实例，无论我们可以使用什么类型的变量来保存对对象的引用，都无法从 RealPoint 的主体外部调用类 Point 的 getX 方法。因此，我们看到字段和方法的行为不同：隐藏与重写不同。

8.5 成员类和接口声明

成员类是其声明直接包含在另一个类或接口声明体中的类（§8.1.7、§9.1.5）。

成员接口是其声明直接包含在另一个类或接口声明体中的接口。

成员类可以是普通类（§8.1）、枚举类（§7.9）或记录类（§9.10）。

成员接口可以是普通接口（§9.1）或注解接口（§8.6）。

类声明主体中成员类或接口声明的可访问性由其访问修饰符指定，如果缺少访问修饰符，则由§6.6 指定。

§8.1.1 中规定了类声明体中成员类声明修饰符的规则。

§9.1.1 规定了类声明体中成员接口声明修饰符的规则。

§6.3 和§6.4.1 规定了成员类或接口的作用域和遮蔽。

如果一个类声明了一个具有特定名称的成员类或接口，那么该成员类或接口的声明被认为隐藏了该类的超类和超接口中具有相同名称的任何和所有可访问的成员类和接口声明。

在这方面，成员类和接口的隐藏类似于字段的隐藏（§8.3）。

类从其直接超类和直接超接口继承超类和超接口的所有非私有成员类和接口，这些类和接口都可由类中的代码访问，并且不被类中的声明隐藏。

一个类可以从其超类和超接口或仅从其超接口继承多个同名成员类或接口。这种情况本身不会导致编译时错误。然而，在类的主体中，任何试图通过其简单名称引用任何此类成员类或接口的尝试都将导致编译时错误，因为引用是不明确的。

从接口继承同一成员类或接口声明可能有多条路径。在这种情况下，成员类或接口被视为

仅继承一次，并且可以通过其简单名称引用，而不会产生歧义。

8.6 实例初始化器

在类中声明的实例初始化器在创建该类的实例时执行（§12.5、§15.9、§8.8.7.1）。

InstanceInitializer:
Block

如果实例初始化器无法正常完成，则为编译时错误（§14.22）。

如果返回语句（§14.17）出现在实例初始化器中的任何位置，则为编译时错误。

允许实例初始化器使用关键字 this（§15.8.3）或关键字 super（§15.11.2，§15.12）引用当前对象，并使用作用域中的任何类型变量。

§8.3.3 规定了实例初始化器如何引用实例变量的限制，即使实例变量在作用域内。

§11.2.3 中规定了实例初始化器的异常检查。

8.7 静态初始化器

初始化类时，执行类中声明的静态初始化器（§12.4.2）。与类变量的任何字段初始化器（§8.3.2）一起，静态初始化器可用于初始化类的类变量。

StaticInitializer: `static` *Block*

如果静态初始化器无法正常完成，则为编译时错误（§14.22）。

如果返回语句（§14.17）出现在静态初始化器中的任何位置，则为编译时错误。

静态初始化器引入了静态上下文（§8.1.3，这限制了引用当前对象的构造的使用。值得注意的是，在静态上下文中禁止使用 this 和 super 关键字（§15.8.3，§15.11.2），对实例变量、实例方法和词法封装声明的类型参数的非限定引用也是如此（§6.5.5.1，§6.6.1，§15.12.3）。

§8.3.3 规定了静态初始化器如何引用类变量的限制，即使类变量在作用域内。

§11.2.3 规定了静态初始化器的异常检查。

8.8 构造函数声明

构造函数用于创建作为类实例的对象(§12.5, §15.9)。

ConstructorDeclaration:
{ConstructorModifier} ConstructorDeclarator [Throws] ConstructorBody

ConstructorDeclarator:

[TypeParameters] SimpleTypeName
（ *[ReceiverParameter ,] [FormalParameterList]* ）

SimpleTypeName:
TypeIdentifier

本节中的规则适用于所有类声明中的构造函数，包括枚举类声明和记录类声明。 但是，特殊规则适用于有关构造函数修饰符、构造函数体和默认构造函数的枚举声明；这些规则见§8.9.2。如§8.10.4 所述，特殊规则也适用于有关构造函数的记录声明。

ConstructorDeclarator 中的 SimpleTypeName 必须是包含构造函数声明的类的简单名称，否则会发生编译时错误。

在所有其他方面，构造函数声明看起来就像没有结果的方法声明（§8.4.5）。

构造函数声明不是成员。它们永远不会被继承，因此不会被隐藏或重写。

构造函数由类实例创建表达式（§15.9）、字符串连接运算符+（§15.18.1）引起的转换和连接以及其他构造函数的显式构造函数调用（§8.8.7）调用。对构造函数的访问由访问修饰符（§6.6）控制，因此可以通过声明不可访问的构造函数来防止类实例化（§8.8.10）。

构造函数永远不会被方法调用表达式调用(§15.12)。

例子 8.8-1. 构造函数声明

```
class Point {
   int x, y;
   Point(int x, int y) { this.x = x; this.y = y; }
}
```


8.8.1 形式参数

构造函数的形式参数在语法和语义上与方法的形式参数完全相同(§8.4.1)。

如果构造函数的最后一个形式参数是一个可变参数，那么构造函数就是一个可变参数构造函数。否则，它就是固定参数构造函数。

非私有内部成员类的构造函数隐式声明了一个变量，作为第一个形式形参，该变量表示类的第一个封闭实例(§15.9.2，§15.9.3)。

为什么只有这种类具有隐式声明的构造函数参数，这是很微妙的。下面的解释可能会有帮助:

1. 在非私有内部成员类的类实例创建表达式中，§15.9.2 指定了成员类的直接封闭实例。成员类可能由不同于类实例创建表达式的编译器的编译器发出。因此，创建表达式的编译器必须有一种标准方法将引用（表示直接封闭实例）传递给成员类的构造函数。因此，Java 编程语言在本节中认为，非私有内部成员类的构造函数隐式声明了直接封闭实例的初始参数。§15.9.3 规定将实例传递给构造函数。
2. 在内部局部类或匿名类的类实例创建表达式中（不在静态上下文中），§15.9.2 指定了局部/匿名类的直接封闭实例。局部/匿名类必须由与类实例创建表达式相同的编译器发出。该编译器可以根据自己的意愿表示直接封闭实例。Java 编程语言不需要在局部/匿名类的构造函数中隐式声明参数。
3. 在匿名类的类实例创建表达式中，如果匿名类的超类是内部类（不是静态上下文），则§15.9.2 指定

了匿名类相对于超类的直接封闭实例。这个实例必须从匿名类传输到它的超类，在超类中它将作为直接封闭实例。由于超类可能是由与类实例创建表达式的编译器不同的编译器发出的，因此有必要以标准的方式传递实例，将它作为第一个参数传递给超类的构造函数。请注意，匿名类本身一定是由与类实例创建表达式相同的编译器发出的，因此，在匿名类将实例传递给超类的构造函数之前，编译器可以按照自己的意愿将与超类相关的直接封闭实例传递给匿名类。然而，为了一致性，Java 编程语言在§15.9.5.1 中认为，在某些情况下，匿名类的构造函数隐式地声明了与超类相关的直接封闭实例的初始参数。

非私有内部成员类可能被不同于编译它的编译器访问，而内部局部类或匿名类总是被编译它的同一个编译器访问，这一事实解释了为什么非私有内部成员类的二进制名被定义为可预测的，而内部局部类或匿名类的二进制名却不可预测(§13.1)。

8.8.2 构造函数签名

在一个类中声明两个具有重写等价签名(§8.4.2)的构造函数是编译时错误。

在一个类中声明两个构造函数的签名具有相同的擦除(§4.6)是编译时错误。

8.8.3 构造函数修饰符

ConstructorModifier:
(one of)
Annotation `public protected private`

关于构造函数声明的注解修饰符的规则见§9.7.4 和§9.7.5。

如果同一个关键字不止一次作为修饰符出现在构造函数声明中，或者构造函数声明中有多个访问修饰符 public、protected 和 private(§6.6)，那将是编译时错误。

在普通的类声明中，没有访问修饰符的构造函数声明具有包访问权限。

如果两个或多个(不同的)方法修饰符出现在一个方法声明中，习惯上(虽然不是必需的)，它们出现的顺序与上面 MethodModifier 的产品中显示的顺序一致。

与方法不同，构造函数不能是 abstract、static、final、native、strictfp 或 synchronized:

- 构造函数不是继承的，因此不需要将其声明为 final。
- 抽象构造函数永远无法实现。
- 构造函数总是针对对象调用的，因此构造函数是静态的没有意义。
- 实际上并不需要对构造函数进行同步，因为它会锁住正在构造的对象，在对象的所有构造函数完成它们的工作之前，其他线程通常不会获得锁。
- 缺少 native 构造函数是一种任意的语言设计选择，这使得 Java Virtual Machine 的实现很容易验证在对象创建期间超类构造函数总是被正确调用。
- 不能将构造函数声明为 strictfp(与方法(§8.4.3)相反)是一种有意的语言设计选择，源于(现在已经过时)将类声明为 strictfp 的能力。

8.8.4 泛型构造函数

如果构造函数声明了一个或多个类型变量，那么它就是泛型的(§4.4)构造函数。

这些类型变量称为构造函数的类型参数。泛型构造函数的类型参数部分的形式与泛型类的类型参数部分相同(§8.1.2)。

构造函数可能是泛型的，与声明构造函数的类本身是否是泛型无关。

泛型构造函数声明定义了一组构造函数，每个构造函数针对类型参数对类型形参部分的可能调用。泛型构造函数声明定义了一组构造函数，每个构造函数用于按类型参数调用类型参数部分。调用泛型构造函数时，可能不需要显式提供类型参数，因为它们通常可以通过推断来提供（§18（类型推断））。

§6.3 和§6.4.1 规定了构造函数类型参数的作用域和遮蔽。

根据§6.5.5.1 的规定，从显式构造函数调用语句或嵌套类或接口中引用构造函数的类型参数受到限制。

8.8.5 构造函数抛出

构造函数的 throws 子句在结构和行为上与方法的 throws 子句相同（§8.4.6）。

8.8.6 构造函数类型

构造函数的类型由其签名和由其 throws 子句给出的异常类型组成。

8.8.7 构造函数体

构造函数主体的第一条语句可以是对同一类或直接超类的另一个构造函数的显式调用。

ConstructorBody:
{ *[ExplicitConstructorInvocation] [BlockStatements]* }

构造器通过一系列一个或多个显式构造器调用直接或间接调用自身是编译时错误。

如果构造函数体不是以显式构造函数调用开始，并且所声明的构造函数不是原始 Object 类的一部分，则构造函数体隐式以超类构造函数调用“super();”开始，对其直接超类的构造函数的调用不带参数。

除了显式调用构造函数的可能性，以及禁止显式返回值（§14.17），构造函数的主体与方法的主体类似（§8.4.7）。

如果返回语句不包含表达式，则可以在构造函数体中使用返回语句（§14.17）。

例子 8.8.7-1. 构造函数体

```
class Point {
   int x, y;
   Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
   static final int WHITE = 0, BLACK = 1;
   int color;
   ColoredPoint(int x, int y) {
      this(x, y, WHITE);
```

```
    }
    ColoredPoint(int x, int y, int color) {
      super(x, y);
      this.color = color;
    }
}
```

这里，ColoredPoint 的第一个构造函数调用第二个构造函数，提供一个附加参数；ColoredPoint 的第二个构造函数调用其超类 Point 的构造函数，传递坐标。

8.8.7.1 显式构造函数调用

ExplicitConstructorInvocation:
[TypeArguments] this ([ArgumentList]);
[TypeArguments] super ([ArgumentList]) ;
ExpressionName . [TypeArguments] super ([ArgumentList]) ;
Primary . [TypeArguments] super ([ArgumentList]) ;

为方便起见，此处显示了§4.5.1 和§15.12 中的以下产品：

TypeArguments:
< TypeArgumentList >

ArgumentList:
Expression {, Expression}

显式构造函数调用语句分为两类：

- 替代构造函数调用以关键字 this 开始（可能以显式类型参数开头）。它们用于调用同一个类的备用构造函数。
- 超类构造函数调用以关键字 super(可能以显式类型参数开头)或 Primary 表达式或 ExpressionName 开始。它们用于调用直接超类的构造函数。他们进一步分为：
  - 非限定超类构造函数调用以关键字 super 开始（可能以显式类型参数开头）。
  - 限定的超类构造函数调用以 Primary 表达式或 ExpressionName 开始。它们允许子类构造函数显式指定新创建的对象相对于直接超类的直接封闭实例（§8.1.3）。当超类是内部类时，这可能是必要的。

显式构造函数调用语句引入了静态上下文（§8.1.3），这限制了引用当前对象的构造的使用。值得注意的是，在静态上下文中禁止使用 this 和 super 关键字（§15.8.3，§15.11.2），对实例变量、实例方法和词法封装声明的类型参数的非限定引用也是如此（§6.5.5.1，§6.6.1，§15.12.3）。

如果 this 或 super 的左侧存在 TypeArguments，则如果任何类型参数是通配符，则为编译时错误（§4.5.1）。

设 C 是被实例化的类，设 S 是 C 的直接超类。

如果超类构造函数调用语句不合格，则：

- 如果 S 是内部成员类，但 S 不是包含 C 的类的成员，则会发生编译时错误。

否则，设 O 是 C 的最里面的封闭类，S 是 C 的成员。C 必须是 O 的内部类(§8.1.3)，否则会发生编译时错误。

- 如果 S 是一个内部局部类，并且 S 不出现在静态上下文中，则设 O 为 S 的直接封闭类或接口声明。C 必须是 O 的内部类，否则会发生编译时错误。

如果超类构造函数调用语句合格，则:

- 如果 S 不是内部类，或者 S 的声明发生在静态上下文中，则会发生编译时错误。
- 否则，设 p 为 Primary 表达式或紧跟在“.super”前面的 ExpressionName，设 O 为 S 的直接封闭类。如果 p 的类型不是 O 或 O 的子类，或者 p 的类型不可访问，则为编译时错误（§6.6）。

§11.2.2 规定了显式构造函数调用语句可以引发的异常类型。

对备用构造函数调用语句的求值过程是，首先从左到右求值构造函数的参数，就像在普通方法调用中一样；然后调用构造函数。

对超类构造函数调用语句的计算如下:

1. 假设 i 是正在创建的实例。必须确定 i 相对于 S(如果有的话)的直接封闭实例:
   - 如果 S 不是一个内部类，或者如果 S 的声明发生在静态上下文中，那么就不存在与 S 相关的 i 的直接封闭实例。
   - 否则，如果超类构造函数调用是不合格的，那么 S 必然是内部局部类或内部成员类。

     如果 S 是一个内部局部类，设 O 是 S 的直接封闭类或接口声明。

     如果 S 是内部成员类，设 O 是 C 中最内部的封闭类，S 是该类的成员。

     设 n 为整数（n≥ 1），使得 O 是 C 的第 n 个词法封闭类或接口声明。

     关于 S 的 i 的直接封闭实例是 this 的第 n 个词法封闭实例。

     > 虽然由于继承，S 可能是 C 的一个成员，但 this 的第零个词法封闭实例（即，它本身）从未用作 i 相对于 S 的直接封闭实例。

   - 否则，如果超类构造函数调用是限定的，则 Primary 表达式或紧接在".super"之前的 ExpressionName，p，被求值。

     如果 p 的计算结果为 null，则会引发 NullPointerException，并且超类构造函数调用会突然结束。

     否则，该求值的结果是 i 相对于 S 的直接封闭实例。
2. 在确定 i 相对于 S（如果有的话）的直接封闭实例之后，超类构造函数调用语句的计算通过从左到右计算构造函数的参数进行，就像在普通方法调用中一样；然后调用构造函数。

3. 最后，如果超类构造函数调用语句正常完成，则执行 C 的所有实例变量初始化器和 C 的所有实例初始化器。如果一个实例初始化器或实例变量初始化器 I 在文本上先于另一个实例初始器或实例变量初始化器 J，则 I 在 J 之前执行。

执行实例变量初始化器和实例初始化器，无论超类构造函数调用实际上是作为显式构造函数调用语句出现还是隐式提供。(备用构造函数调用不执行这个额外的隐式执行。)

例子 8.8.7.1-1. 显式构造函数调用语句的限制

如果§8.8.7 中示例中 ColoredPoint 的第一个构造函数更改如下:

```
class Point {
   int x, y;
   Point(int x, int y) { this.x = x; this.y = y; }
}
class ColoredPoint extends Point {
   static final int WHITE = 0, BLACK = 1;
   int color;
   ColoredPoint(int x, int y) {
      this(x, y, color); // Changed to color from WHITE
   }
   ColoredPoint(int x, int y, int color) {
      super(x, y);
      this.color = color;
   }
}
```

然后会发生编译时错误，因为显式构造函数调用语句不能使用实例变量 color。

例子 8.8.7.1-2. 限定超类构造函数调用

在下面的代码中，ChildOfInner 没有词法上的封闭类或接口声明，因此 ChildOfInner 的实例没有封闭实例。但是，ChildOfInner (Inner)的超类有一个词法上封闭的类声明(Outer)， Inner 的实例必须有一个 Outer 的封闭实例。当创建 Inner 的实例时，将设置 Outer 的封闭实例。因此，当创建 ChildOfInner 的实例(它隐式地是 Inner 的实例)时，必须在 ChildOfInner 的构造函数中通过限定的超类调用语句提供 Outer 的封闭实例。Outer 的实例被称为相对于 Inner 的 ChildOfInner 的直接封闭实例。

```
class Outer {
    class Inner {}
}
class ChildOfInner extends Outer.Inner {
    ChildOfInner() { (new Outer()).super(); }
}
```

也许令人惊讶的是，对于多个 ChildOfInner 实例，同一个 Outer 实例可以作为相对于 Inner 的 ChildOfInner 的直接封闭实例。这些 ChildOfInner 的实例隐式链接到同一个 Outer 实例。下面的程序通过将 Outer 的一个实例传递给 ChildOfInner 的构造函数来实现这一点，ChildOfInner 在一个限定的超类构造函数调用语句中使用该实例。显式构造函数调用语句的规则并不禁止使用包含该语句的构造函数的形式参数。

```
class Outer {
    int secret = 5;
```

```
    class Inner {
        int getSecret() { return secret; }
        void setSecret(int s) { secret = s; }
    }
}
class ChildOfInner extends Outer.Inner {
    ChildOfInner(Outer x) { x.super(); }
}

public class Test {
    public static void main(String[] args) {
        Outer x = new Outer();
        ChildOfInner a = new ChildOfInner(x);
        ChildOfInner b = new ChildOfInner(x);
        System.out.println(b.getSecret());
        a.setSecret(6);
        System.out.println(b.getSecret());
    }
}
```

程序产生输出:

```
5
6
```

其结果是，通过引用不同的 ChildOfInner 实例，可以看到 Outer 公共实例中对实例变量的操作，尽管这些引用不是传统意义上的别名。

8.8.8 构造函数重载

构造函数的重载在行为上与方法重载是相同的(§8.4.9)。重载在编译时由每个类实例创建表达式解析(§15.9)。

8.8.9 默认构造函数

如果类不包含构造函数声明，则隐式声明默认构造函数。顶级类、成员类或局部类的默认构造函数形式如下:

- 默认构造函数具有与类相同的访问修饰符，除非类缺少访问修饰符，在这种情况下，默认构造函数具有包访问权限 (§6.6)。
- 默认构造函数没有形式参数，除了在非私有内部成员类中，默认构造函数隐式声明了一个表示类的直接封闭实例的形式参数。
- 默认构造函数没有 throws 子句。
- 如果声明的类是原始类 Object，则默认构造函数体为空。否则，默认构造函数只调用超类构造函数，不带参数。

匿名类的默认构造函数的形式见§15.9.5.1。

如果隐式声明了默认构造函数，但超类没有不接受参数的可访问的构造函数，也没有 throws 子句，那么这是编译时错误。

例子 8.8.9-1. 默认构造函数

声明:

```
public class Point {
   int x, y;
}
```

和以下声明是等价的:

```
public class Point {
   int x, y;
   public Point() { super();
}
```

其中默认构造函数是 public，因为类 Point 是 public。

例子 8.8.9-2. 构造函数和类的可访问性

类的默认构造函数具有与类本身相同的可访问性的规则是简单而直观的。但是请注意，这并不意味着只要类是可访问的，构造函数就是可访问的。考虑:

```
package p1;
public class Outer {
    protected class Inner {}
}
package p2;
class SonOfOuter extends p1.Outer {
   void foo() {
      new Inner(); // compile-time access error
   }
}
```

Inner 的默认构造函数是受保护的。但是，构造函数是相对于 Inner 受保护的，而 Inner 是相对于 Outer 受保护的。所以，Inner 在 SonOfOuter 中是可访问的，因为它是 Outer 的一个子类。Inner 的构造函数在 SonOfOuter 中是不可访问的，因为 SonOfOuter 类不是 Inner 的子类! 因此，尽管 Inner 是可访问的，但它的默认构造函数是不可访问的。

8.8.10 防止类的实例化

通过声明至少一个构造函数、防止创建默认构造函数以及将所有构造函数声明为 private，可以将类设计为防止类声明之外的代码创建类的实例(§6.6.1)。

公共类同样可以通过声明至少一个构造函数来防止在其包外创建实例，通过声明没有公共或受保护的构造函数来防止创建具有公共访问权限的默认构造函数 (§6.6.2)。

例子 8.8.10-1. 通过构造函数可访问性防止实例化

```
class ClassOnly {
   private ClassOnly() { }
   static String just = "only the lonely";
```

```
}
```

在这里，类 ClassOnly 不能被实例化，而在以下代码中:

```
package just;
public class PackageOnly {
    PackageOnly() { }
    String[] justDesserts = { "cheesecake", "ice cream" };
}
```

公共类 PackageOnly 只能在声明它的包 just 中实例化。如果 PackageOnly 的构造函数受保护，则此限制也适用，尽管在这种情况下，其他包中的代码可能实例化 PackageOnly 的子类。

8.9 枚举类

枚举声明指定了一个新的枚举类，这是一种受限制的类，它定义了一组命名类实例。

EnumDeclaration:
{ClassModifier} `enum` *TypeIdentifier [ClassImplements] EnumBody*

枚举声明可以指定顶级枚举类(§7.6)、成员枚举类(§8.5、§9.5)或局部枚举类(§14.3)。

枚举声明中的 TypeIdentifier 指定了枚举类的名称。

如果枚举声明具有修饰符 abstract、final、sealed 或 non-sealed，则为编译时错误。

枚举类要么是隐式 final，要么是隐式密封的，如下所示:

- 如果枚举类的声明不包含具有类体的枚举常量，则枚举类为隐式 final 的 (§8.9.1)。
- 如果枚举类 E 的声明至少包含一个具有类体的枚举常量，则该枚举类是隐式密封的。E 允许的直接子类(§8.1.6)是由具有类体的枚举常量隐式声明的匿名类。

嵌套枚举类是隐式静态的。也就是说，每个成员枚举类和局部枚举类都是静态的。允许在声明成员枚举类时冗余指定静态修饰符，但不允许声明局部枚举类(§14.3)。

如果同一个关键字作为枚举声明的修饰符出现不止一次，或者一个枚举声明有多个访问修饰符 public、protected 和 private(§6.6)，那将是编译时错误。

枚举类 E 的直接超类类型是 Enum<E> (§8.1.4)。

> 枚举声明没有 extends 子句，因此不可能显式声明直接超类类型，即使 enum<E>也是如此。

枚举类除了由其枚举常量定义的实例外没有其他实例。试图显式实例化枚举类是编译时错误(§15.9.1)。

> 除了编译时错误之外，还有三种机制确保枚举类的实例不存在枚举常量定义以外的实例:
>
> - 枚举中的 final clone 方法确保永远无法克隆枚举常量。
> - 禁止枚举类的反射实例化。
> - 序列化机制的特殊处理可确保不会因反序列化而创建重复实例。

8.9.1 枚举常量

枚举声明的主体可以包含枚举常量。枚举常量定义了枚举类的一个实例。

EnumBody:
{ *[EnumConstantList] [,] [EnumBodyDeclarations]* }

EnumConstantList:
EnumConstant {, EnumConstant}

EnumConstant:
{EnumConstantModifier} Identifier [([ArgumentList])] [ClassBody]

EnumConstantModifier:
Annotation

为了方便起见，以下是§15.12 的产品:

ArgumentList:
Expression {, Expression}

关于枚举常量声明的注解修饰符的规则见§9.7.4 和§9.7.5。

EnumConstant 中的 Identifier 可以在名称中使用，以引用枚举常量。

枚举常量的作用域和遮蔽在§6.3 和§6.4.1 中有规定。

枚举常量可以后跟实参，当在类初始化期间创建该常量时，实参将传递给枚举的构造函数，本节稍后将对此进行描述。要调用的构造函数是使用重载解析的常规规则来选择的(§15.12.2)。如果省略参数，则假定参数列表为空。

枚举常量的可选类体隐式声明了一个匿名类(§15.9.5)，其(i)是直接封闭枚举类(§8.1.4)的直接子类(§8.1.1.2)，(ii)是 final 的(§8.1.1.2)。类体由匿名类的常规规则管理;特别是它不能包含任何构造函数。只有在这些类主体中声明的实例方法重写了封闭枚举类中的可访问方法时，才可以在封闭枚举类外部调用这些实例方法（§8.4.8）。

枚举常量的类体声明抽象方法是编译时错误。

由于每个枚举常量只有一个实例，因此在比较两个对象引用时，如果已知其中至少一个引用了枚举常量，则允许使用==操作符代替 equals 方法。

枚举中的 equals 方法是仅调用 super.equals 的 final 方法，并返回结果，从而执行相等比较。

8.9.2 枚举体声明

除了枚举常量之外，枚举声明的主体还可以包含构造函数和成员声明，以及实例和静态初始化器。

EnumBodyDeclarations:

；{ClassBodyDeclaration}

为了方便起见，以下是§8.1.7 的产品:

ClassBodyDeclaration:
ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration
ClassDeclaration
InterfaceDeclaration

枚举声明体中的任何构造函数或成员声明都适用于枚举类，就像它们出现在普通类声明体中一样，除非另有明确说明。

如果枚举声明中的构造函数声明是公共的或受保护的，则为编译时错误（§6.6）。

如果枚举声明中的构造函数声明包含超类构造函数调用语句（§8.8.7.1），则为编译时错误。

从类的枚举声明中的构造函数、实例初始化器或实例变量初始化器引用枚举类的静态字段是编译时错误，除非该字段是常量变量（§4.12.4）。

在枚举声明中，没有访问修饰符的构造函数声明是私有的。

在没有构造函数声明的枚举声明中，默认构造函数是隐式声明的。默认构造函数是私有的，没有形式参数，也没有 throws 子句。

> 实际上，编译器可能会通过在枚举类的默认构造函数中声明字符串和 int 参数来反映枚举类。但是，这些参数没有被指定为“隐式声明”，因为不同的编译器不需要就默认构造函数的形式达成一致。只有枚举声明的编译器知道如何实例化枚举常量;其他编译器可以简单地依赖枚举类的隐式声明的公共静态字段(§8.9.3)，而不考虑这些字段是如何初始化的。

如果枚举声明 E 有一个抽象方法 m 作为成员，则会在编译时出错，除非 E 至少有一个枚举常量，而且 E 的所有枚举常量都有提供 m 具体实现的类体。

枚举声明声明终结器是一个编译时错误(§12.6)。枚举类的实例可能永远不会完成。

例子 8.9.2-1. 枚举体声明

```
enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);
    Coin(int value) { this.value = value; }

    private final int value;
    public int value() { return value; }
}
```

每个枚举常量通过构造函数传入的值在 value 字段中安排不同的值。该字段表示美国硬币的价值，单位为美分。请注意，枚举类的构造函数可以声明的参数没有限制。

例子 8.9.2-2. 枚举常量自引用的限制

如果没有静态字段访问规则，由于枚举类固有的初始化循环，显然合理的代码将在运行时失败。(循环存在于任何具有“自类型”静态字段的类中。) 下面是一个可能失败的代码示例:

```
import java.util.HashMap;
import java.util.Map;

enum Color {
    RED, GREEN, BLUE;
    Color() { colorMap.put(toString(), this); }

    static final Map<String,Color> colorMap = new HashMap<String,Color>();
}
```

这个枚举类的静态初始化将抛出 NullPointerException，因为当枚举常量的构造函数运行时，静态变量 colorMap 没有初始化。上面的限制确保这样的代码不能编译。然而，代码可以很容易地重构以正常工作:

```
import java.util.HashMap;
import java.util.Map;

enum Color {
    RED, GREEN, BLUE;

    static final Map<String,Color> colorMap =
        new HashMap<String,Color>();
    static {
        for (Color c : Color.values())
          colorMap.put(c.toString(), c);
    }
}
```


重构后的版本显然是正确的，因为静态初始化是从上到下进行的。

8.9.3 枚举类成员

枚举类 E 的成员如下所示:

- 在 E 的声明体中声明的成员。
- 从 Enum<E>继承的成员。
- 对于 E 声明体中声明的每个枚举常量 c, E 都有一个隐式声明的公共静态 final 字段，类型为 E，名称与 c 相同。该字段有一个变量初始化器，用于实例化 E，并将 c 的任何参数传递给为 E 选择的构造函数。该字段具有与 c 相同的注解(如果有的话)。

  这些字段隐式声明的顺序与对应的枚举常量相同，位于 E 声明体中显式声明的任何静态字段之前。

  当初始化相应隐式声明的字段时，就称创建了枚举常量。
- 一个隐式声明的方法 public static E[] values()，它返回一个包含 E 的枚举常量的数组，其顺序与 E 声明体中出现的顺序相同。

- 隐式声明的方法 public static E valueOf(String name)，它返回具有指定名称的枚举常量 E。

由此可见，枚举类 E 的声明不能包含与 E 的枚举常量对应的隐式声明字段冲突的字段，也不能包含与隐式声明的方法冲突的方法或重写 Enum <E>类的 final 方法的方法。

例子 8.9.3-1. 使用增强的 for 循环迭代枚举常量

```
public class Test {
    enum Season { WINTER, SPRING, SUMMER, FALL }

    public static void main(String[] args) {
       for (Season s : Season.values())
          System.out.println(s);
    }
}
```

程序产生输出：

```
WINTER
SPRING
SUMMER
FALL
```

例子 8.9.3-2. 在枚举常量间切换

switch 语句(§14.11)用于模拟从枚举类外部向枚举类添加方法的过程。这个例子向§8.9.2 中的 Coin 类“添加”了一个 color 方法，并打印出一个包含硬币、它们的值和颜色的表格。

```
class Test {
   enum CoinColor { COPPER, NICKEL, SILVER }

   static CoinColor color(Coin c) {
      switch (c) {
      case PENNY:
         return CoinColor.COPPER;
      case NICKEL:
         return CoinColor.NICKEL;
      case DIME: case QUARTER:
         return CoinColor.SILVER;
      default:
         throw new AssertionError("Unknown coin: " + c);
      }
   }

   public static void main(String[] args) {
      for (Coin c : Coin.values())
         System.out.println(c + "\t\t" +
            c.value() + "\t" + color(c));
      }
}
```

程序产生输出：

```
PENNY           1    COPPER
NICKEL          5    NICKEL
DIME            10   SILVER
```

例子 8.9.3-3. 带类体的枚举常量

与其使用 switch 语句从外部向枚举类“添加”行为，还不如使用类主体直接将行为附加到枚举常量。

```
enum Operation {
     PLUS {
        double eval(double x, double y) { return x + y; }
     },
     MINUS {
        double eval(double x, double y) { return x - y; }
     },
     TIMES {
        double eval(double x, double y) { return x * y; }
     },
     DIVIDED_BY {
        double eval(double x, double y) { return x / y; }
     };

     // Each constant supports an arithmetic operation
     abstract double eval(double x, double y);

     public static void main(String[] args) {
         double x = Double.parseDouble(args[0]);
         double y = Double.parseDouble(args[1]);
         for (Operation op : Operation.values())
            System.out.println(x + " " + op + " " + y +
                               " = " + op.eval(x, y));
     }
}
```

程序产生输出：

```
java Operation 2.0 4.0
2.0 PLUS 4.0 = 6.0
2.0 MINUS 4.0 = -2.0
2.0 TIMES 4.0 = 8.0
2.0 DIVIDED_BY 4.0 = 0.5
```

此模式比使用 switch 语句安全得多，因为该模式排除了忘记为新常量添加行为的可能性(因为枚举声明将导致编译时错误)。

例子 8.9.3-4. 多个枚举类

在下面的程序中，扑克牌类构建在两个简单枚举之上。

```
import java.util.ArrayList;
import java.util.List;

class Card implements Comparable<Card>,
                      java.io.Serializable {
     public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE,
                        TEN,JACK, QUEEN, KING, ACE }

     public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

```
    private final Rank rank;
    private final Suit suit;
    public Rank rank() { return rank; }
    public Suit suit() { return suit; }

    private Card(Rank rank, Suit suit) {
      if (rank == null || suit == null)
        throw new NullPointerException(rank + ", " + suit);
      this.rank = rank;
      this.suit = suit;
    }

    public String toString() { return rank + " of " + suit; }

    // Primary sort on suit, secondary sort on rank
    public int compareTo(Card c) {
        int suitCompare = suit.compareTo(c.suit);
        return (suitCompare != 0 ?
                    suitCompare :
                    rank.compareTo(c.rank));
    }

    private static final List<Card> prototypeDeck =
      new ArrayList<Card>(52);

    static {
      for (Suit suit : Suit.values())
        for (Rank rank : Rank.values())
          prototypeDeck.add(new Card(rank, suit));
    }
    // Returns a new deck
    public static List<Card> newDeck() {
      return new ArrayList<Card>(prototypeDeck);
    }
}
```

下面的程序练习 Card 类。它在命令行上采用两个整数参数，表示要处理的手数和每手牌数:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Deal {
    public static void main(String[] args) {
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);
        List<Card> deck = Card.newDeck();
        Collections.shuffle(deck);
        for (int i=0; i < numHands; i++)
          System.out.println(dealHand(deck, cardsPerHand));
    }

    /**
     *       Returns a new ArrayList consisting of the last n
     *       elements of deck, which are removed from deck.
     *       The returned list is sorted using the elements'
     *       natural ordering.
     */
```

```
    public static <E extends Comparable<E>>
    ArrayList<E> dealHand(List<E> deck, int n) {
      int deckSize = deck.size();
      List<E> handView = deck.subList(deckSize - n, deckSize); ArrayList<E> hand
      = new ArrayList<E>(handView);
      handView.clear();
      Collections.sort(hand);
      return hand;
    }
}
```

程序产生输出:

```
java Deal 4 3
[DEUCE of CLUBS, SEVEN of CLUBS, QUEEN of DIAMONDS]
[NINE of HEARTS, FIVE of SPADES, ACE of SPADES]
[THREE of HEARTS, SIX of HEARTS, TEN of SPADES]
[TEN of CLUBS, NINE of DIAMONDS, THREE of SPADES]
```

8.10 记录类

记录声明指定了一个新的记录类，这是一种定义简单值聚合的受限类。

RecordDeclaration:
  {ClassModifier} `record` *TypeIdentifier [TypeParameters] RecordHeader*
    [ClassImplements] RecordBody

记录声明可以指定顶级记录类别（§7.6）、成员记录类别（§8.5、§9.5）或局部记录类别（§14.3）。

记录声明中的 TypeIdentifier 指定记录类的名称。

如果记录声明的修饰符为 abstract、sealed 或 non-sealed，则这是编译时错误。

记录类是隐式 final 的。允许记录类的声明冗余地指定 final 修饰符。

嵌套记录类是隐式静态的。也就是说，每个成员记录类和局部记录类都是静态的。允许成员记录类的声明冗余指定静态修饰符，但对局部记录类的声明是不允许的（§14.3）。

如果同一关键字作为记录声明的修饰符出现多次，或者记录声明具有多个访问修饰符 public、protected 和 private（§6.6），则这是编译时错误。

记录类的直接超类类型是 Record (§8.1.4)。

记录声明没有 extends 子句，因此不可能显式声明直接超类类型，甚至 Record。

序列化机制处理记录类实例的方式不同于普通的可序列化或可外部化对象。特别地，一个记录对象使用规范构造函数反序列化(§8.10.4)。

8.10.1 记录组件

记录类的记录组件(如果有的话)在记录声明的头中指定。每个记录组件由一个类型(可选地

前面有一个或多个注解)和一个指定记录组件名称的标识符组成。一个记录组件对应于记录类的两个成员:一个隐式声明的私有字段，以及一个显式或隐式声明的公共访问方法(§8.10.3)。

如果一个记录类没有记录组件，那么在记录声明的头中将出现一对空圆括号。

RecordHeader:
（ *[RecordComponentList]* ）

RecordComponentList:
RecordComponent {， RecordComponent}

RecordComponent:
{RecordComponentModifier} UnannType Identifier
VariableArityRecordComponent

VariableArityRecordComponent:
{RecordComponentModifier} UnannType {Annotation} ... Identifier

RecordComponentModifier:
Annotation

记录组件可以是可变记录组件，由类型后面的省略号表示。一个记录类最多允许一个可变记录组件。如果可变记录组件出现在记录组件列表中除最后一个位置以外的任何位置，则为编译时错误。

关于记录组件的注解修饰符的规则见§9.7.4 和§9.7.5。

> 如果在记录组件上下文中注解接口是可用的，那么记录组件上的注解通过反射是可用的(§9.6.4.1)。如果记录组件的注解接口适用于其他上下文（§8.10.3、§8.10.4），则记录组件上的注解将独立地传播到记录类的成员和构造函数的声明中。

如果记录声明拥有名为 clone、finalize、getClass、hashCode、notify、notifyAll、toString 或 wait 的记录组件，则这是一个编译时错误。

> 这些是 Object 中无参数公共和受保护方法的名称。不允许它们作为记录组件的名称可以在许多方面避免混淆。首先，每个记录类都提供 hashCode 和 toString 的实现，这些实现返回记录对象作为一个整体的表示；它们不能用作名为 hashCode 或 toString 的记录组件的访问器方法（§8.10.3），并且无法从记录类之外访问此类记录组件。类似地，一些记录类可能提供 clone 和（遗憾的是）finalize 的实现，因此称为 clone 或 finalize 的记录组件无法通过访问器方法访问。最后，Object 中的 getClass、notify、notifyAll 和 wait 方法是 final 的，因此同名的记录组件不能有访问器方法。(访问器方法将具有与 final 方法相同的签名，因此将尝试重写它们，但未成功。)

对于一个记录声明来说，有两个名称相同的记录组件是一个编译时错误。

记录组件的声明类型取决于它是否是可变记录组件:

- 如果记录组件不是可变记录组件，则声明的类型由 UnannType 表示。

· 如果记录组件是可变记录组件，则声明的类型是§10.2 规定的数组类型。

如果可变记录组件的声明类型具有不可具体化的元素类型（§4.7），则对于可变记录组件的声明会出现编译时未检查警告，除非规范构造函数（§8.10.4）用@SafeVarargs（§9.6.4.7）注解，或者该警告被@SuppressWarnings（§9.6.4.5）抑制。

8.10.2 记录体声明

记录声明的主体可以包含构造函数和成员声明以及静态初始化器。

RecordBody:
{ *{RecordBodyDeclaration}* }

RecordBodyDeclaration:
ClassBodyDeclaration
CompactConstructorDeclaration

为了方便起见，以下是§8.1.7 的产品:

ClassBodyDeclaration:
ClassMemberDeclaration
InstanceInitializer
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration:
FieldDeclaration
MethodDeclaration
ClassDeclaration
InterfaceDeclaration ;

§8.10.4.2 描述了 CompactConstructorDeclaration 子句。

如果记录声明的主体包含一个非静态字段声明，这是编译时错误(§8.3.1.1)。

如果记录声明的主体包含了 abstract 或 native 的方法声明，这是编译时错误(§8.4.3.1，§8.4.3.4)。

如果记录声明体中包含实例初始化器，这是一个编译时错误(§8.6)。

8.10.3 记录成员

对于每个记录组件，记录类都有一个字段，该字段的名称与记录组件相同，类型与声明的记录组件类型相同。这个隐式声明的字段被称为组件字段。

组件字段是私有的、final 的和非静态的。

组件字段的注解(如果有的话)出现在相应的记录组件上，并且其注解接口适用于字段声明上下文，或类型上下文，或两者都适用(§9.7.4)。

此外，对于每个记录组件，记录类有一个与记录组件同名的方法和一个空的形式参数列表。

这种显式或隐式声明的方法称为访问器方法。

如果一个记录组件的访问器方法被显式声明，那么下面所有的都必须为真，否则会发生编译时错误:

- 访问器方法的返回类型(§8.4.5)必须与记录组件声明的类型相同。
- 访问器方法不能是泛型的 (§8.4.4)。
- 访问器方法必须是没有形式参数和 throws 子句的公共实例方法。

如果一个记录类有一个没有显式声明访问器方法的记录组件，那么该记录组件的访问器方法将隐式声明，具有以下属性:

- 它的名称与记录组件的名称相同。
- 它的返回类型与记录组件声明的类型相同。
- 它不是泛型的。
- 它是公共的实例方法，没有形式参数和 throws 子句。
- 它被注解为相应记录组件上的注解(如果有的话)，并且其注解接口适用于方法声明上下文中，或适用于类型上下文中，或两者都适用(§9.7.4)。
- 它的主体返回相应组件字段的值。

> 对记录组件名称的限制(§8.10.1)意味着没有隐式声明的访问器方法具有与类 Object 的非私有方法重写等价的签名。一个显式的方法声明，它接受其中一个受限制的名称，例如 public void wait() {...}不是一个访问器方法，因为 wait 从来不是一个记录组件名。
>
> 出现在记录组件上的注解不会传播到该记录组件的显式声明的访问器方法。在某些情况下，程序员可能需要在显式声明的访问器方法上复制记录组件的注解，但这通常不是必要的。
>
> 传播到隐式声明的访问器方法的注解必须产生合法注解的方法。例如，在以下记录声明中，隐式声明的访问器方法 x()将使用@SafeVarargs 进行注解，但这种注解在固定参数方法上是非法的（§9.6.4.7）：
>
> ```
> record BadRecord(@SafeVarargs int x) {}  // Error
> ```

组件字段和访问器方法的作用域和遮蔽在§6.3 和§6.4.1 中指定。(它们对应的记录组件不是一个声明，因此没有自己的作用域。)

记录类可以显式声明访问器方法以外的实例方法，但不能显式声明实例变量（§8.10.2）。允许类方法和类变量的显式声明。

记录类的所有成员，包括隐式声明的成员，都要遵守类中成员声明的常规规则（§8.3、§8.4、§8.5）。

所有适用于普通类的继承规则都适用于记录类。特别是，记录类可能从超接口继承成员，尽管超接口方法永远不会作为访问器方法被继承，因为记录类总是显式或隐式地声明重写超接口方法的访问器方法。

例如，一个记录类可以从它的直接超接口继承默认方法，尽管默认方法体不知道记录类的组件字段。以下程序打印 Logged:

```
public class Test {
    interface Logging {
        default void logAction() {
            System.out.println("Logged");
        }

    }

    record Point(int i, int j) implements Logging {}

    public static void main(String[] args) {
        Point p = new Point(10, 20);
        p.logAction();
    }
}
```

一个记录类提供了在类 Record 中声明的所有抽象方法的实现。对于下面的每一种方法，如果一个记录类 R 没有显式地声明一个具有相同修饰符、名称和签名的方法(§8.4.2)，那么该方法隐式声明如下:

- 一个方法 public final boolean equals(Object)返回 true 当且仅当参数是 R 的一个实例，当前实例等于 R 的每个记录组件上的参数实例; 否则返回 false。

  记录类 R 的一个实例 a 与记录组件 c 的同一个记录类的另一个实例 b 相等，如下所示:

  - 如果记录组件 c 的类型是引用类型，则等式如下所示: 如果 a 和 b 的组件字段 c 的值都是空引用，那么返回 true; 如果 a 或 b 的组件字段 c 的值（但不是两者）是空引用，则返回 false; 否则，通过调用 a 的组件字段 c 的值的 equals 方法来确定相等性，参数是 b 的组件字段 c 的值。
  - 如果记录组件 c 的类型是原生类型 T，则通过调用对应于 T 的包装类的静态方法 compare（§5.1.7）确定相等性，第一个参数由 a 的组件字段 c 的值给出，第二个参数由 b 的组件字段 c 的值给出；如果该方法将返回 0，则返回 true，否则返回 false。

    在包装类中使用 compare 可确保隐式声明的 equals 方法是自反的，并与具有浮点组件的记录类的隐式声明方法 hashCode 保持一致。

- 一个方法 public final int hashCode()，它返回一个从 R 的每个记录组件上的哈希码值派生出来的哈希码值。

  记录组件 c 的记录类实例 a 的哈希码值如下:

  - 如果记录组件 c 的类型是引用类型，那么哈希代码值就像用 a 的组件字段 c 的值调用方法 hashCode 来确定一样。
  - 如果记录组件 c 的类型是原生类型 T，那么哈希码值的确定就好像是将 a 的组件字段 c 的值进行装箱转换(§5.1.7)，然后在结果对象上调用与 T 对应的包装类的 hashCode 方法。

• 一个方法 public final String toString()，返回一个字符串，该字符串派生自记录类的名称以及 R 的每个记录组件的名称和字符串表示。

记录类实例 a 的记录组件 c 的字符串表示如下：

- 如果记录组件 c 的类型是引用类型，则在 a 的组件字段 c 的值上调用 toString 方法来确定字符串表示。
- 如果记录组件 c 的类型是原生类型 T，则字符串表示通过对 a 的组件字段 c 的值进行装箱转换（§5.1.7），然后在结果对象上调用对应于 T 的包装类的 toString 方法来确定。

请注意，相等性、哈希代码值和字符串表示是通过直接查看组件字段的值而不是调用访问器方法来确定的。

考虑一个记录类 R，它有组件 C_1、…、C_n，每个组件都有一个隐式声明的访问器方法，以及一个隐式声明的 equals 方法。如果用以下方式复制 R 的实例 r1：

```
R r2 = new R(r1.c1(), r1.c2(), ..., r1.cn());
```

然后，假设 r1 不是空引用，则表达式 r1.equals(r2)的计算结果总是 true。显式声明的访问器方法和 equals 方法应遵守此不变量。编译器通常不可能检查显式声明的方法是否尊重此不变量。下面的记录声明是糟糕的风格，因为它的访问器方法修剪了 x 和 y 组件，因此阻止 p3 等于 p1：

```
record SmallPoint(int x, int y) {
    public int x() { return  this.x < 100 ? this.x :  100;}
    public int y() { return  this.y < 100 ? this.y :  100;}

    public static void main(String[] args) {
        SmallPoint p1 = new  SmallPoint(200,300);
        SmallPoint p2 = new  SmallPoint(200,300);
        System.out.println(p1.equals(p2)); // prints true

        SmallPoint p3 = new SmallPoint(p1.x(), p1.y());
        System.out.println(p1.equals(p3)); // prints false
    }
}
```

8.10.4 记录构造函数声明

为确保记录组件的正确初始化，记录类不会隐式声明默认构造函数（§8.8.9）。相反，记录类有一个显式或隐式声明的规范构造函数，用于初始化记录类的所有组件字段。

在记录声明中显式声明规范构造函数有两种方法：通过声明具有适当签名的普通构造函数（§8.10.4.1）或通过声明紧凑构造函数（§4.2）。

给定符合规范的普通构造函数的签名，以及为紧凑构造函数派生的签名，构造函数签名规则（§8.8.2）意味着如果记录声明同时具有符合规范的正常构造函数和紧凑构造函数，则这是编译时错误。

无论哪种方式，显式声明的规范构造函数必须至少提供与记录类一样多的访问，如下所示：

• 如果记录类是公共的，那么规范构造函数必须是公共的；否则将发生编译错误。

- 如果记录类是受保护的，那么规范构造函数必须是受保护的或公共的；否则，将发生编译时错误。
- 如果记录类具有包访问权限，那么规范构造函数必须不是私有的;否则，将发生编译时错误。
- 如果记录类是私有的，那么规范构造函数可以声明为任何可访问级别。

显式声明的规范构造函数可以是固定参数构造函数或可变参数构造函数（§8.8.1）。

如果在记录类 R 的声明中未显式声明规范构造函数，则在 R 中隐式声明具有以下属性的规范构造函数 r:

- r 的签名没有类型参数，并且具有由 R 的导出形式参数列表给出的形式参数，定义如下。
- r 具有与 R 相同的访问修饰符，除非 R 缺少访问修饰词，在这种情况下，r 具有包访问权限。
- r 没有 throws 子句。
- r 的主体使用 r 的相应形式参数初始化记录类的每个组件字段，顺序为记录组件（对应于组件字段）出现在记录头中。

记录类的派生形式参数列表是通过从记录头中的每个记录组件派生的形式参数形成的，顺序如下:

- 如果记录组件不是可变记录组件，则派生的形式参数与记录组件具有相同的名称和声明的类型。

  如果记录组件是可变记录组件，则导出的形式参数是与记录组件具有相同名称和声明类型的可变参数（§8.4.1）。
- 派生的形式参数使用记录组件上的注解(如果有的话)进行注解，其注解接口适用于形式参数上下文或类型上下文，或两者兼有（§9.7.4）。

记录声明可能包含非规范构造函数的构造函数声明。记录声明中每个非规范构造函数的主体必须以备用构造函数调用（§8.8.7.1）开始，否则会发生编译时错误。

8.10.4.1 普通规范构造函数

记录类 R 声明中的(非紧凑)构造函数是 R 的规范构造函数，如果它的签名与 R 的派生构造函数签名是重写等价的(§8.4.2)。

记录类 R 的派生构造函数签名是由名称 R、无类型参数和形式参数类型组成的签名，这些参数类型是通过按顺序获取每个记录组件的声明类型从 R 的记录头派生的。

> 由于规范构造函数的签名重写等价于记录类的派生构造函数签名，因此记录类中只能显式声明一个规范构造函数。

（非紧凑）规范构造函数的声明必须满足以下所有条件，否则会发生编译时错误：

- 形式参数列表中的每个形式参数必须与相应的记录组件具有相同的名称和声明的类型。当且仅当相应的记录组件是可变记录组件时，形式参数必须是可变参数。
- 构造函数不能是泛型的 (§8.8.4)。
- 构造函数不能有 throws 子句。
- 构造函数体不得包含显式构造函数调用语句（§8.8.7.1）。
- 必须满足普通类声明中构造函数声明的所有其他规则（§8.8）。

这些规则的结果是，记录组件上的注解可能不同于显式声明的规范构造函数的相应形式参数上的注解。例如，以下记录声明有效：

```
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@interface Foo {}
@interface Bar {}

record Person(@Foo String name) {
    Person(@Bar String name) {
       this.name = name;
    }
}
```


8.10.4.2 紧凑的规范构造函数

紧凑构造函数声明是构造函数声明的一种简洁形式，仅在记录声明中可用。它声明了记录类的规范构造函数，而不需要手动重复类的记录组件作为构造函数的形式参数。

CompactConstructorDeclaration:
{ConstructorModifier} SimpleTypeName ConstructorBody

为方便起见，此处显示了§8.8、§8.8.3 和§8.7 中的以下产品：

ConstructorModifier:
(one of)
Annotation `public protected private`

SimpleTypeName:
TypeIdentifier

ConstructorBody:
`{` *[ExplicitConstructorInvocation] [BlockStatements]* `}`

记录声明具有多个紧凑构造函数声明是编译时错误。

记录类的紧凑构造函数的形式参数是隐式声明的。它们由记录类的派生的形式参数列表给出（§8.10.4）。

如果记录类具有可变记录组件，则记录类的紧凑构造函数是可变构造函数（§8.8.1）。

紧凑构造函数声明的签名等于记录类的派生构造函数签名（§8.10.4.1）。

紧凑构造函数声明的主体必须满足以下所有条件，否则会发生编译时错误：

- 主体不能包含 return 语句(§14.17)。
- 主体不能包含显式构造函数调用语句 (§8.8.7.1)。
- 主体不能包含对记录类的组件字段的赋值。
- 普通类声明中构造函数的所有其他规则都必须满足(§8.8)，除了记录类的组件字段必须明确赋值，而且在紧凑构造函数结束时不能明确不赋值(§8.3.1.2)。

> 如果一个记录声明有一个名为 c 的记录组件，那么紧凑构造函数体中的简单名称 c 表示隐式形式参数 c，而不是组件字段 c。

在紧凑构造函数体中的最后一条语句(如果有的话)正常完成后(§14.1)，record 类的所有组件字段隐式初始化为相应形参的值。组件字段按照记录头中声明相应记录组件的顺序初始化。

> 紧凑构造函数声明的目的是在构造函数体中只需要给出验证或规范化参数的代码;剩下的初始化代码由编译器提供。例如，下面的记录类有一个简化有理数的紧凑构造函数:

```
record Rational(int num, int denom) {
  private static int gcd(int a, int b) {
    if (b == 0) return Math.abs(a);
    else return gcd(b, a % b);
  }

  Rational {
    int gcd = gcd(num, denom);
    num /= gcd;
    denom /= gcd;
  }
}
```

> 紧凑的构造函数 Rational{…}的行为与普通构造函数相同:

```
Rational(int num, int denom) {
  int gcd = gcd(num, denom);
  num /= gcd;
  denom /= gcd;
  this.num = num;
  this.denom = denom;
}
```