OOP 目录

面向对象基础

面向对象编程，是一种通过对象的方式，把现实世界映射到计算机模型的一种编程方法。

现实世界中，我们定义了“人”这种抽象概念，而具体的人则是“小明”、“小红”、“小军”等一个个具体的人。所以，“人”可以定义为一个类（class），而具体的人则是实例（instance）：

现实世界	计算机模型	Java代码
人 类	class	class Person { }
小明	实例 / ming	Person ming = new Person()
小红	实例 / hong	Person hong = new Person()
小军	实例 / jun	Person jun = new Person()

class和instance

1. ***class*** 是一种对象模板，它定义了如何创建实例。因此，它是一种数据类型。//class定义了抽象的数据类型和方法函数。

2. ***instance***是对象实例，instance是根据class创建的实例，可以创建多个instance，每个instance类型相同，但各自属性可能不相同。

定义class

在Java中，创建一个类，例如，给这个类命名为Person，就是定义一个class：

```
class Person {
    public String name;
    public int age;
}
```

一个class可以包含多个字段（field），字段用来描述一个类的特征。上面的Person类，我们定义了两个字段，一个是String类型的字段，命名为name，一个是int类型的字段，命名为age。因此，通过class，把一组数据汇集到一个对象上，实现了数据封装。

*public*是用来修*饰*字段的，它表示*这*个字段可以被外部*访问*。

创建实例

定义了class，只是定义了对象模版，而要根据对象模版创建出真正的对象实例，必须用new操作符。

new操作符可以创建一个实例，然后，我们需要定义一个引用类型的变量来指向这个实例：

```
Person ming = new Person();
```

上述代码创建了一个Person类型的实例，并通过变量ming指向它。

步骤：

1. 注意区分Person ming是定义Person类型的变量ming，
2. 而new Person()是创建Person实例。

有了指向这个实例的变量，我们就可以通过这个变量来操作实例。访问实例变量可以用变量.字段

小结

- **在OOP中，class和instance是“模版”和“实例”的关系；**
- **定义class就是定义了一种数据类型，对应的instance是这种数据类型的实例；**
- **class定义的field，在每个instance都会拥有各自的field，且互不干扰；**
- **通过new操作符创建新的instance，然后用变量指向它，即可通过变量来引用这个instance；**
- **访问实例字段的方法是变量名.字段名；**
- **指向instance的变量都是引用变量。**

方法

一个类通过定义方法，就可以给外部代码暴露一些操作的接口，同时，内部自己保证逻辑一致性。

外部代码不可以直接访问private的参数，但可以通过公开的public接口去修改和获取参数的值。而类中定义的方法可以通过设置一些条件，筛选调不合适的操作。

调用方法的语法是`实例变量.方法名(参数);`。一个方法调用就是一个语句，所以不要忘了在末尾加`;`。

定义方法

从上面的代码可以看出，定义方法的语法是：

```
修饰符 方法返回类型 方法名(方法参数列表) {
    若干方法语句;
    return 方法返回值;
}
```

方法返回值通过`return`语句实现，如果没有返回值，返回类型设置为`void`，可以省略`return`。

private方法

定义private方法的理由是内部方法是可以调用private方法的。例如：

```
// private method
public class Main {
    public static void main(String[] args) {        Person ming = new Person();
        ming.setBirth(2008);
        System.out.println(ming.getAge());
    }
}

class Person {
    private String name;
    private int birth;

    public void setBirth(int birth) {
        this.birth = birth;
    }

    public int getAge() {
        return calcAge(2019); // 调用private方法
    }

    // private方法:
    private int calcAge(int currentYear) {
        return currentYear - this.birth;
    }
}
```

```
//Output
11
```

观察上述代码，calcAge()是一个private方法，外部代码无法调用，但是，内部方法getAge()可以调用它。

此外，我们还注意到，这个Person类只定义了birth字段，没有定义age字段，获取age时，通过方法getAge()返回的是一个实时计算的值，并非存储在某个字段的值。这说明方法可以封装一个类的对外接口，调用方不需要知道也不关心Person实例在内部到底有没有age字段。

this变量

在方法内部，可以使用一个隐含的变量this，它始终指向当前实例。

如果没有命名冲突，可以省略this。例如：

```
class Person {
    private String name;

    public String getName() {
        return name; // 相当于this.name
    }
}
```

但是，**如果有局部变量和字段重名，那么局部变量优先级更高，就必须加上this**：

```
class Person {
    private String name;

    public void setName(String name) {
        this.name = name; // 前面的this不可少，少了就变成局部变量name了
    }
}
```

方法参数

方法可以包含0个或任意个参数。方法参数用于接收传递给方法的变量值。调用方法时，必须严格按照参数的定义一一传递。**调用时，也必须严格按照定义传递对应的参数。**

可变参数

可变参数用类型...定义，可变参数相当于数组类型：

```
class Group {
    private String[] names;

    public void setNames(String... names) {
        this.names = names;
```

```
    }
}
```

上面的`setNames()`就定义了一个可变参数。调用时，可以这么写：

```
Group g = new Group();
g.setNames("Xiao Ming", "Xiao Hong", "Xiao Jun"); // 传入3个String
g.setNames("Xiao Ming", "Xiao Hong"); // 传入2个String
g.setNames("Xiao Ming"); // 传入1个String
g.setNames(); // 传入0个String
```

参数绑定

调用方把参数传递给实例方法时，调用时传递的值会按参数位置一一绑定。

我们先观察一个基本类型参数的传递：

```
// 基本类型参数绑定
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        int n = 15; // n的值为15
        p.setAge(n); // 传入n的值
        System.out.println(p.getAge()); // 15
        n = 20; // n的值改为20
        System.out.println(p.getAge()); // 15还是20?
    }
}

class Person {
    private int age;

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

运行代码，从结果可知，修改外部的局部变量`n`，不影响实例`p`的`age`字段，原因是`setAge()`方法获得的参数，复制了`n`的值，因此，`p.age`和局部变量`n`互不影响。

结论：基本类型参数的传递，是调用方值的复制。双方各自的后续修改，互不影响。

我们再看一个传递引用参数的例子：

```
// 引用类型参数绑定,参数是一个数组
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        String[] fullname = new String[] { "Homer", "Simpson" };
        p.setName(fullname); // 传入fullname数组
        System.out.println(p.getName()); // "Homer Simpson"
        fullname[0] = "Bart"; // fullname数组的第一个元素修改为"Bart"
        System.out.println(p.getName()); // "Homer Simpson"还是"Bart Simpson"?
    }
}

class Person {
    private String[] name;

    public String getName() {
        return this.name[0] + " " + this.name[1];
    }

    public void setName(String[] name) {
        this.name = name;
    }
}
```

注意到`setName()`的参数现在是一个数组。一开始，把`fullname`数组传进去，然后，修改`fullname`数组的内容，结果发现，实例`p`的字段`p.name`也被修改了！

结论：引用类型参数的传递，调用方的变量，和接收方的参数变量，指向的是同一个对象。双方任意一方对这个对象的修改，都会影响对方（因为指向同一个对象嘛）。传入的参数是指向数组首地址的一个指针，而改动的是数组中指针的指向，因此改动任意一方都会相互影响。

有了上面的结论，我们再看一个例子：

```
// 引用类型参数绑定
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        String bob = "Bob";
        p.setName(bob); // 传入bob变量
        System.out.println(p.getName()); // "Bob"
        bob = "Alice"; // bob改名为Alice
        System.out.println(p.getName()); // "Bob"还是"Alice"?
    }
}

class Person {
    private String name;

    public String getName() {
```

```
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

辨析：

前一个例子参数是数组，传入的是指向数组的首地址。因此，fullname 和 name指向的都是创建的数组首地址。而无论是对于fullname[0], 还是对于name[0]的修改都是对于数组内某一个内存单元内指针指向的修改。因此，改动任意一方都会影响另一方的值。name指针的指向没有改变。

第二个例子的参数是一个指向字符串的指针，传的是一个字符串的首地址。因此，通过setName()修改后,使得name和bob两个指针都指向“Bob”这个字符串储存的内存单元。而bob="Alice"语句所做的修改只是让bob这个指针指向Alice，并不会改变name指针的指向。

小结

方法可以让外部代码安全地访问实例字段；

方法是一组执行语句，并且可以执行任意逻辑；

方法内部遇到return时返回，void表示不返回任何值（注意和返回null不同）；

外部代码通过public方法操作实例，内部代码可以调用private方法；

理解方法的参数绑定。

构造方法

创建实例的时候，实际上是通过构造方法来初始化实例的。

构造方法的名称就是类名。构造方法的参数没有限制，在方法内部，也可以编写任意语句。但是，和普通方法相比，构造方法没有返回值（也没有void），调用构造方法，必须用new操作符。

默认构造方法

任何class都有构造方法。

那前面我们并没有为Person类编写构造方法，为什么可以调用new Person()？

原因是如果一个类没有定义构造方法，编译器会自动为我们生成一个默认构造方法，它没有参数，也没有执行语句，类似这样：

```
class Person {
    public Person() {
    }
}
```

要特别注意的是，**如果我们自定义了一个构造方法，那么，编译器就不再自动创建默认构造方法。**

如果既要能使用带参数的构造方法，又想保留不带参数的构造方法，那么只能把两个构造方法都定义出来：

```
// 构造方法
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Xiao Ming", 15); // 既可以调用带参数的构造方法
        Person p2 = new Person(); // 也可以调用无参数构造方法
    }
}

class Person {
    private String name;
    private int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

没有在构造方法中初始化字段时，引用类型的字段默认是`null`，数值类型的字段用默认值，`int`类型默认值是`0`，布尔类型默认值是`false`：

```
class Person {
    private String name; // 默认初始化为null
    private int age; // 默认初始化为0

    public Person() {
    }
}
```

也可以对字段直接进行初始化：

```
class Person {
    private String name = "Unamed";
```

```
    private int age = 10;
}
```

多构造方法

1. 可以定义多个构造方法，在通过new操作符调用的时候，编译器通过构造方法的参数数量、位置和类型自动区分：

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        this.name = name;
        this.age = 12;
    }

    public Person() {
    }
}
```

如果调用`new Person("Xiao Ming", 20);`，会自动匹配到构造方法`public Person(String, int)`。

如果调用`new Person("Xiao Ming");`，会自动匹配到构造方法`public Person(String)`。

如果调用`new Person();`，会自动匹配到构造方法`public Person()`。

2. 一个构造方法可以调用其他构造方法，这样做的目的是便于代码复用。**调用其他构造方法的语法是`this(…)`**：

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        this(name, 18); // 调用另一个构造方法Person(String, int)
    }

    public Person() {
```

```
        this("Unnamed"); // 调用另一个构造方法Person(String)
    }
}
```

小结

实例在创建时通过new操作符会调用其对应的构造方法，构造方法用于初始化实例；

没有定义构造方法时，编译器会自动创建一个默认的无参数构造方法；

可以定义多个构造方法，编译器根据参数自动判断；

可以在一个构造方法内部调用另一个构造方法，便于代码复用。

方法重载

在一个类中，我们可以定义多个方法。如果有一系列方法，它们的功能都是类似的，只有参数有所不同，那么，可以把这一组方法名做成同名方法。例如，在`Hello`类中，定义多个`hello()`方法：

```
class Hello {
    public void hello() {
        System.out.println("Hello, world!");
    }

    public void hello(String name) {
        System.out.println("Hello, " + name + "!");
    }

    public void hello(String name, int age) {
        if (age < 18) {
            System.out.println("Hi, " + name + "!");
        } else {
            System.out.println("Hello, " + name + "!");
        }
    }
}
```

这种方法名相同，但各自的参数不同，称为**方法重*载***（***Overload***）。

注意：方法重载的返回值类型通常都是相同的。

方法重载的目的是，功能类似的方法使用同一名字，更容易记住，因此，调用起来更简单。

举个例子，`String`类提供了多个重载方法`indexOf()`，可以查找子串：

`int indexOf(int ch)`：根据字符的`Unicode`码查找；

`int indexOf(String str)`：根据字符串查找；

`int indexOf(int ch, int fromIndex)`：根据字符查找，但指定起始位置；

`int indexOf(String str, int fromIndex)`根据字符串查找，但指定起始位置。

小结

方法重载是指多个方法的方法名相同，但各自的参数不同；

重载方法应该完成类似的功能，参考`String`的`indexOf()`；

重载方法返回值类型应该相同。

继承

继承是面向对象编程中非常强大的一种机制，它首先可以复用代码。当我们让`Student`从`Person`继承时，`Student`就获得了`Person`的所有功能，我们只需要为`Student`编写新增的功能。

Java使用`extends`关键字来实现继承：

```
class Person {
    private String name;
    private int age;

    public String getName() {...}
    public void setName(String name) {...}
    public int getAge() {...}
    public void setAge(int age) {...}
}

class Student extends Person {
    // 不要重复name和age字段/方法，
    // 只需要定义新增score字段/方法:
    private int score;

    public int getScore() { … }
    public void setScore(int score) { … }
}
```

可见，通过继承，Student只需要编写额外的功能，不再需要重复代码。

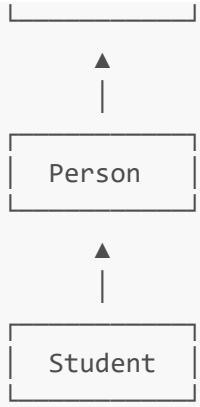
注意：子类自*动获*得了父类的所有字段，*严*禁定*义*与父类重名的字段！

在OOP的术语中，我们把`Person`称为超类（super class），父类（parent class），基类（base class），把`Student`称为子类（subclass），扩展类（extended class）。

继承树

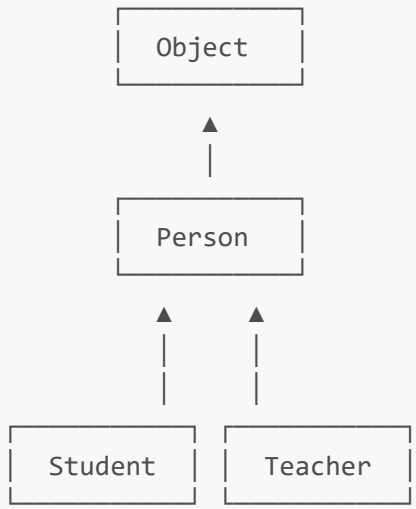
注意到我们在定义`Person`的时候，没有写`extends`。在Java中，没有明确写`extends`的类，编译器会自动加上`extends Object`。所以，任何类，除了`Object`，都会继承自某个类。下图是`Person`、`Student`的继承树：

```
┌──────────┐
│  Object  │
```



Java只允许一个class继承自一个类，因此，一个类有且仅有一个父类。只有Object特殊，它没有父类。

类似的，如果我们定义一个继承自Person的Teacher，它们的继承树关系如下：



protected

继承有个特点，就是子类无法访问父类的private字段或者private方法。

这使得继承的作用被削弱了。为了让子类可以访问父类的字段，我们需要把private改为protected。

用protected修饰的字段可以被子类访问：

```
class Person {
    protected String name;
    protected int age;
}

class Student extends Person {
    public String hello() {
        return "Hello, " + name; // OK!
    }
}
```

因此，protected关键字可以把字段和方法的访问权限控制在继承树内部，一个protected字段和方法可以被其子类，以及子类的子类所访问。

super

super关键字表示父类（超类）。子类引用父类的字段时，可以用super.fieldName。例如：

```
class Student extends Person {
    public String hello() {
        return "Hello, " + super.name;
    }
}
```

实际上，这里使用super.name，或者this.name，或者name，效果都是一样的。编译器会自动定位到父类的name字段。

但是，**在某些时候，就必须使用super**。我们来看一个例子：

```
// super
public class Main {
    public static void main(String[] args) {
        Student s = new Student("Xiao Ming", 12, 89);
    }
}

class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        this.score = score;
    }
}
```

运行上面的代码，会得到一个编译错误，大意是在Student的构造方法中，无法调用Person的构造方法。

这是因为在Java中，任何class的构造方法，第一行语句必须是调用父类的构造方法。如果没有明确地调用父类的构造方法，编译器会帮我们自动加一句super();，所以，Student类的构造方法实际上是这样：

```
class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        super(); // 自动调用父类的构造方法
        this.score = score;
    }
}
```

但是，Person类并没有无参数的构造方法，因此，编译失败。

解决方法是调用Person类存在的某个构造方法。例如：

```
class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        super(name, age); // 调用父类的构造方法Person(String, int)
        this.score = score;
    }
}
```

这样就可以正常编译了！

因此我们得出结论：**如果父类没有默认的构造方法，子类就必须显式调用super()并给出参数以便让编译器定位到父类的一个合适的构造方法。**

这里还顺带引出了另一个问题：**即子类不会*继*承任何父类的构造方法。子类默*认*的构造方法是*编译*器自*动*生成的，不是*继*承的。**(子类可以继承父类的所有字段。)

阻止继承

正常情况下，只要某个class没有final修饰符，那么任何类都可以从该class继承。

从Java 15开始，允许使用sealed修饰class，并通过permits明确写出能够从该class继承的子类名称。

例如，定义一个Shape类：

```
public sealed class Shape permits Rect, Circle, Triangle {
    ...
}
```

上述Shape类就是一个sealed类，它只允许指定的3个类继承它。如果写：

```
public final class Rect extends Shape {...}
```

是没问题的，因为Rect出现在Shape的permits列表中。但是，如果定义一个Ellipse就会报错：

```
public final class Ellipse extends Shape {...}
// Compile error: class is not allowed to extend sealed class: Shape
```

原因是Ellipse并未出现在Shape的permits列表中。这种sealed类主要用于一些框架，防止继承被滥用。

sealed类在Java 15中目前是预览状态，要启用它，必须使用参数--enable-preview和--source 15。

向上转型

如果一个引用变量的类型是Student，那么它可以指向一个Student类型的实例：

```
Student s = new Student();
```

如果一个引用类型的变量是Person，那么它可以指向一个Person类型的实例：

```
Person p = new Person();
```

现在问题来了：如果Student是从Person继承下来的，那么，一个引用类型为Person的变量，能否指向Student类型的实例？

```
Person p = new Student(); // ???
```

测试一下就可以发现，这种指向是允许的！

这是因为Student继承自Person，因此，它拥有Person的全部功能。Person类型的变量，如果指向Student类型的实例，对它进行操作，是没有问题的！

这种**把一个子类类型安全地变为父类类型的赋值，被称为向上转型（upcasting）**。

向上转型实际上是把一个子类型安全地变为更加抽象的父类型：

```
Student s = new Student();
Person p = s; // upcasting, ok
Object o1 = p; // upcasting, ok
Object o2 = s; // upcasting, ok
```

注意到继承树是Student > Person > Object，所以，可以把Student类型转型为Person，或者更高层次的Object。**即可以把更具体的类转变成更抽象的类。**

向下转型

和向上转型相反，**如果把一个父类类型强制转型为子类类型，就是向下转型（downcasting）**。例如：

```
Person p1 = new Student(); // upcasting, ok
Person p2 = new Person();
Student s1 = (Student) p1; // ok
Student s2 = (Student) p2; // runtime error! ClassCastException!
```

如果测试上面的代码，可以发现：

Person类型p1实际指向Student实例，Person类型变量p2实际指向Person实例。在向下转型的时候，把p1转型为Student会成功，因为p1确实指向Student实例，把p2转型为Student会失败，因为p2的实际类型是Person，不能把父类变为子类，因为子类功能比父类多，多的功能无法凭空变出来。 **只有当抽象类原来指向具体类的时候，才能将抽象类的指向强制转换成具体类。**

因此，向下转型很可能会失败。失败的时候，Java虚拟机会报ClassCastException。

为了避免向下转型出错，Java提供了instanceof操作符，可以先判断一个实例究竟是不是某种类型：

```
Person p = new Person();
System.out.println(p instanceof Person); // true
System.out.println(p instanceof Student); // false

Student s = new Student();
System.out.println(s instanceof Person); // true
System.out.println(s instanceof Student); // true

Student n = null;
System.out.println(n instanceof Student); // false
```

instanceof实际上判断一个变量所指向的实例是否是指定类型，或者这个类型的子类。如果一个引用变量为null，那么对任何instanceof的判断都为false。

利用instanceof，在向下转型前可以先判断：

```
Person p = new Student();
if (p instanceof Student) {
    // 只有判断成功才会向下转型:
    Student s = (Student) p; // 一定会成功
}
```

从Java 14开始，判断instanceof后，可以直接转型为指定变量，避免再次强制转型。例如，对于以下代码：

```
Object obj = "hello";
if (obj instanceof String) {
    String s = (String) obj;
```

```
        System.out.println(s.toUpperCase());
    }
```

可以改写如下：

```
// instanceof variable:
public class Main {
    public static void main(String[] args) {
        Object obj = "hello";
        if (obj instanceof String s) {
            // 可以直接使用变量s:
            System.out.println(s.toUpperCase());
        }
    }
}
```

这种使用instanceof的写法更加简洁。

区分继承和组合

在使用继承时，我们要注意逻辑一致性。

考察下面的Book类：

```
class Book {
    protected String name;
    public String getName() {...}
    public void setName(String name) {...}
}
```

这个Book类也有name字段，那么，我们能不能让Student继承自Book呢？

```
class Student extends Book {
    protected int score;
}
```

显然，从逻辑上讲，这是不合理的，Student不应该从Book继承，而应该从Person继承。

究其原因，是因为Student是Person的一种，它们是is关系，而Student并不是Book。实际上Student和Book的关系是has关系。

具有has关系不应该使用继承，而是使用组合，即Student可以持有一个Book实例：

```
class Student extends Person {
    protected Book book;
```

```
    protected int score;
}
```

因此，继承是is关系，组合是has关系。

小结

继承是面向对象编程的一种强大的代码复用方式；

Java只允许单继承，所有类最终的根类是Object；

protected允许子类访问父类的字段和方法；

子类的构造方法可以通过super()调用父类的构造方法；

可以安全地向上转型为更抽象的类型；

可以强制向下转型，最好借助instanceof判断；

子类和父类的关系是is，has关系不能用继承。

多态

在继承关系中，子类如果定义了一个与父类方法签名完全相同的方法，被称为覆写（Override）。

例如，在Person类中，我们定义了run()方法：

```
class Person {
    public void run() {
        System.out.println("Person.run");
    }
}
```

在子类Student中，覆写这个run()方法：

```
class Student extends Person {
    @Override
    public void run() {
        System.out.println("Student.run");
    }
}
```

Override和Overload不同的是，如果方法签名如果不同，就是Overload，Overload方法是一个新方法；如果方法签名相同，并且返回值也相同，就是Override。

注意：方法名相同，方法参数相同，但方法返回值不同，也是不同的方法。在Java程序中，出现这种情况，编译器会报错。

```
class Person {
    public void run() { … }
}

class Student extends Person {
    @Override   //Compile error
    // 不是Override，因为参数不同:
    public void run(String s) { … }
    @Override   //Compile error
    // 不是Override，因为返回值不同:
    public int run() { … }
}
```

加上@Override可以让编译器帮助检查是否进行了正确的覆写。希望进行覆写，但是不小心写错了方法签名，编译器会报错。

多态是指，针对某个类型的方法调用，其真正执行的方法取决于运行时期实际类型的方法。

例如：

```
Person p = new Student();
p.run(); // 无法确定运行时究竟调用哪个run()方法
```

实际调用的是Student的run()方法。

如果我们编另外一个方法：

```
public void runTwice(Person p) {
    p.run();
    p.run();
}
```

它传入的参数类型是Person，我们是无法知道传入的参数实际类型究竟是Person，还是Student，还是Person的其他子类，因此，也无法确定调用的是不是Person类定义的run()方法。

所以，**多*态*的特性就是，运行期才能*动态*决定*调*用的子类方法。*对*某个类型*调*用某个方法，*执*行的*实际*方法可能是某个子类的覆写方法。**

> **这种不确定性的方法调用，究竟有什么作用？**

我们还是来举栗子。

假设我们定义一种收入，需要给它报税，那么先定义一个Income类：

```
class Income {
    protected double income;
    public double getTax() {
```

```
        return income * 0.1; // 税率10%
    }
}
```

对于工资收入，可以减去一个基数，那么我们可以从Income派生出SalaryIncome，并覆写getTax()：

```
class Salary extends Income {
    @Override
    public double getTax() {
        if (income <= 5000) {
            return 0;
        }
        return (income - 5000) * 0.2;
    }
}
```

如果你享受国务院特殊津贴，那么按照规定，可以全部免税：

```
class StateCouncilSpecialAllowance extends Income {
    @Override
    public double getTax() {
        return 0;
    }
}
```

现在，我们要编写一个报税的财务软件，对于一个人的所有收入进行报税，可以这么写：

```
public double totalTax(Income... incomes) {
    double total = 0;
    for (Income income: incomes) {
        total = total + income.getTax();
    }
    return total;
}
```

来试一下：

```
// Polymorphic
public class Main {
    public static void main(String[] args) {
        // 给一个有普通收入、工资收入和享受国务院特殊津贴的小伙伴算税:
        Income[] incomes = new Income[] {
            new Income(3000),
            new Salary(7500),
            new StateCouncilSpecialAllowance(15000)
```

```
        };
        System.out.println(totalTax(incomes));
    }

    public static double totalTax(Income... incomes) {
        double total = 0;
        for (Income income: incomes) {
            total = total + income.getTax();
        }
        return total;
    }
}

class Income {
    protected double income;

    public Income(double income) {
        this.income = income;
    }

    public double getTax() {
        return income * 0.1; // 税率10%
    }
}

class Salary extends Income {
    public Salary(double income) {
        super(income);
    }

    @Override
    public double getTax() {
        if (income <= 5000) {
            return 0;
        }
        return (income - 5000) * 0.2;
    }
}

class StateCouncilSpecialAllowance extends Income {
    public StateCouncilSpecialAllowance(double income) {
        super(income);
    }

    @Override
    public double getTax() {
        return 0;
    }
}

// Output
  800.0
```

观察`totalTax()`方法：利用多态，`totalTax()`方法只需要和`Income`打交道，它完全不需要知道`Salary`和`StateCouncilSpecialAllowance`的存在，就可以正确计算出总的税。如果我们要新增一种稿费收入，只需要从`Income`派生，然后正确覆写`getTax()`方法就可以。把新的类型传入`totalTax()`，不需要修改任何代码。

可见，多态具有一个非常强大的功能，就是允许添加更多类型的子类实现功能扩展，却不需要修改基于父类的代码。

覆写Object方法

因为所有的class最终都继承自Object，而Object定义了几个重要的方法：

`toString()`：把instance输出为String；

`equals()`：判断两个instance是否逻辑相等；

`hashCode()`：计算一个instance的哈希值。

在必要的情况下，我们可以覆写Object的这几个方法。例如：

```
class Person {
    ...
    // 显示更有意义的字符串:
    @Override
    public String toString() {
        return "Person:name=" + name;
    }

    // 比较是否相等:
    @Override
    public boolean equals(Object o) {
        // 当且仅当o为Person类型:
        if (o instanceof Person) {
            Person p = (Person) o;
            // 并且name字段相同时，返回true:
            return this.name.equals(p.name);
        }
        return false;
    }

    // 计算hash:
    @Override
    public int hashCode() {
        return this.name.hashCode();
    }
}
```

调用super

在子类的覆写方法中，如果要调用父类的被覆写的方法，可以通过`super`来调用。例如：

```
class Person {
    protected String name;
    public String hello() {
        return "Hello, " + name;
    }
}

Student extends Person {
    @Override
    public String hello() {
        // 调用父类的hello()方法:
        return super.hello() + "!";
    }
}
```

final

继承可以允许子类覆写父类的方法。**如果一个父类不允许子类对它的某个方法进行覆写，可以把该方法标记为`final`。****用`final`修饰的方法不能被`Override`**：(类似于C++中的const)

```
class Person {
    protected String name;
    public final String hello() {
        return "Hello, " + name;
    }
}

Student extends Person {
    // compile error: 不允许覆写
    @Override
    public String hello() {
    }
}
```

如果一个类不希望任何其他类继承自它，那么可以把这个类本身标记为`final`。

用`final`修饰的类不能被继承：

```
final class Person {
    protected String name;
}

// compile error: 不允许继承自Person
Student extends Person {
}
```

对于一个类的实例字段，同样可以用`final`修饰。

用final修饰的字段在初始化后不能被修改。

例如：

```
class Person {
    public final String name = "Unamed";
}
```

对final字段重新赋值会报错：

```
Person p = new Person();
p.name = "New Name"; // compile error!
```

可以在构造方法中初始化final字段：

```
class Person {
    public final String name;
    public Person(String name) {
        this.name = name;
    }
}
```

这种方法更为常用，因为可以保证实例一旦创建，其final字段就不可修改。（在构造函数，*创*建类*时*就*给*类*赋值*，之后不可更改。）

小结

子类可以覆写父类的方法（Override），覆写在子类中改变了父类方法的行为；

Java的方法调用总是作用于运行期对象的实际类型，这种行为称为多态；

final修饰符有多种作用：

final修饰的方法可以阻止被覆写；

final修饰的class可以阻止被继承；

final修饰的field必须在创建对象时初始化，随后不可修改。

抽象类

如果一个class定义了方法，但没有具体执行代码，这个方法就是抽象方法，抽象方法用abstract修饰。

因为无法执行抽象方法，因此这个类也必须申明为抽象类（abstract class）。

使用abstract修饰的类就是抽象类。

我们无法实例化一个抽象类：(因为抽象类中，没有具体的实现的方法，需要被子类实例化)

```
Person p = new Person(); // 编译错误
```

无法实例化的抽象类有什么用？

因为抽象类本身被设计成只能用于被继承，因此，抽象类可以强迫子类实现其定义的抽象方法，否则编译会报错。因此，**抽象方法实际上相当于定义了“规范”。**

面向抽象编程

当我们定义了抽象类`Person`，以及具体的`Student`、`Teacher`子类的时候，我们可以通过抽象类`Person`类型去引用具体的子类的实例：

```
Person s = new Student();
Person t = new Teacher();
```

这种引用抽象类的好处在于，我们对其进行方法调用，并不关心`Person`类型变量的具体子类型：

```
// 不关心Person变量的具体子类型:
s.run();
t.run();
```

同样的代码，如果引用的是一个新的子类，我们仍然不关心具体类型：

```
// 同样不关心新的子类是如何实现run()方法的:
Person e = new Employee();
e.run();
```

这种尽量引用高层类型，避免引用实际子类型的方式，称之为面向抽象编程。

面向抽象编程的本质就是：

上层代码只定义规范（例如：`abstract class Person`）；

不需要子类就可以实现业务逻辑（正常编译）；

具体的业务逻辑由不同的子类实现，调用者并不关心。

小结

通过abstract定义的方法是抽象方法，它只有定义，没有实现。抽象方法定义了子类必须实现的接口规范；

定义了抽象方法的class必须被定义为抽象类，从抽象类继承的子类必须实现抽象方法；

如果不实现抽象方法，则该子类仍是一个抽象类；

面向抽象编程使得调用者只关心抽象方法的定义，不关心子类的具体实现。

接口

在抽象类中，抽象方法本质上是定义接口规范：即规定高层类的接口，从而保证所有子类都有相同的接口实现，这样，多态就能发挥出威力。

如果一个抽象类没有字段，所有方法全部都是抽象方法：

```
abstract class Person {
    public abstract void run();
    public abstract String getName();
}
```

就可以把该抽象类改写为接口：`interface`。

在Java中，使用`interface`可以声明一个接口：

```
interface Person {
    void run();
    String getName();
}
```

所谓`interface`，就是比抽象类还要抽象的纯抽象接口，因为它连字段都不能有。**因*为*接口定*义*的所有方法默*认*都是*`public abstract`*的，所以*这*两个修*饰*符不需要写出来（写不写效果都一*样*）。**

当一个具体的`class`去实现一个interface时，需要使用`implements`关键字。举个例子：

```
class Student implements Person {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println(this.name + " run");
    }

    @Override
    public String getName() {
        return this.name;
    }
}
```

我们知道，在Java中，一个类只能继承自另一个类，不能从多个类继承。但是，**一个类可以实现多个interface**，例如：

```
class Student implements Person, Hello { // 实现了两个interface
    ...
}
```

术语

> 注意区分术语：

> **Java的接口特指interface的定义，表示一个接口类型和一组方法签名，而编程接口泛指接口规范，如方法签名，数据格式，网络协议等。**

抽象类和接口的对比如下：

abstract	class	interface
继承	只能extends一个class	可以implements多个interface
字段	可以定义实例字段	不能定义实例字段
抽象方法	可以定义抽象方法	可以定义抽象方法
非抽象方法	可以定义非抽象方法	可以定义default方法

接口继承

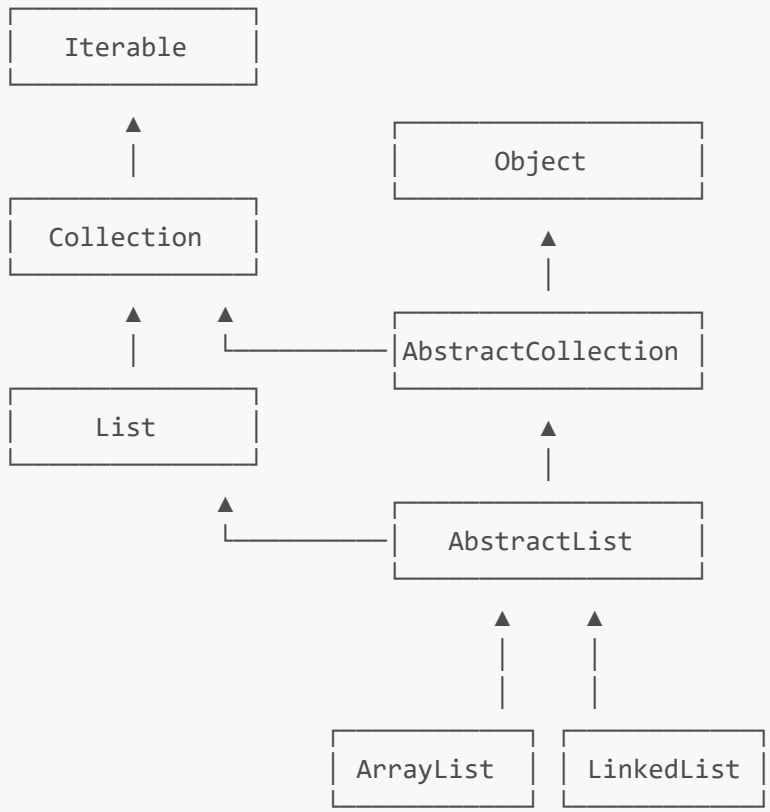
一个interface可以继承自另一个interface。**interface继承自interface使用extends，它相当于扩展了接口的方法**。例如：

```
interface Hello {
    void hello();
}

interface Person extends Hello {
    void run();
    String getName();
}
```

此时，Person接口继承自Hello接口，因此，Person接口现在实际上有3个抽象方法签名，其中一个来自继承的Hello接口。

继承关系

合理设计interface和abstract class的继承关系，可以充分复用代码。一般来说，公共逻辑适合放在abstract class中，具体逻辑放到各个子类，而接口层次代表抽象程度。可以参考Java的集合类定义的一组接口、抽象类以及具体子类的继承关系：



在使用的时候，实例化的对象永远只能是某个具体的子类，但总是通过接口去引用它，因为接口比抽象类更抽象：

```
List list = new ArrayList(); // 用List接口引用具体子类的实例
Collection coll = list; // 向上转型为Collection接口
Iterable it = coll; // 向上转型为Iterable接口
```

default方法

在接口中，可以定义default方法。

> default关键字修饰的方法就是初始化的抽象方法。或者说是一个已经实现了的抽象方法，不需要再在其他implement接口位置进行实现。比如定义了一个接口，有大量的类实现了这个接口，但是新需求来了，需要在原有的基础上添加一个方法，而使用default关键字的话就不用每个实现类都实现一次。

```
//interface1
public interface ExtendInterface1 {

    default String getName(String str) {
        return str;
    }

    default int getNumber(int i) {
        return i;
```

```
    }
}
```

```
public interface ExtendInterface2 {

    default int getNumber(int i) {
        return i * i;
    }
}
```

由于两个接口中有重名的方法，要手动覆盖。

```
public interface TestInterface extends ExtendInterface1, ExtendInterface2 {

    @Override
    default int getNumber(int i) {
        // TODO Auto-generated method stub
        return ExtendInterface2.super.getNumber(i);
    }
}
```

```
public class TestClass implements TestInterface{

    @Test
    public void Test() {
        System.out.println(this.getNumber(100));
        System.out.println(this.getName("TestInterface"));
    }
}
```

实现类可以不必覆写default方法。**default方法的目的是，当我们需要给接口新增一个方法时，会涉及到修改全部子类。如果新增的是default方法，那么子类就不必全部修改，只需要在需要覆写的地方去覆写新增方法。**

default方法和抽象类的普通方法是有所不同的。因为interface没有字段，default方法无法访问字段，而抽象类的普通方法可以访问实例字段。

小结

- **Java的接口（interface）定义了纯抽象规范，一个类可以实现多个接口；**
- **接口也是数据类型，适用于向上转型和向下转型；**
- **接口的所有方法都是抽象方法，接口不能定义实例字段；**
- **接口可以定义default方法（JDK>=1.8）。**

静态字段和静态方法

在一个`class`中定义的字段，我们称之为实例字段。实例字段的特点是，每个实例都有独立的字段，各个实例的同名字段互不影响。

还有一种字段，是**用`static`修饰的字段，称为静态字段：`static field`。**

实例字段在每个实例中都有自己的一个独立“空间”，但是静态字段只有一个共享“空间”，所有实例都会共享该字段。

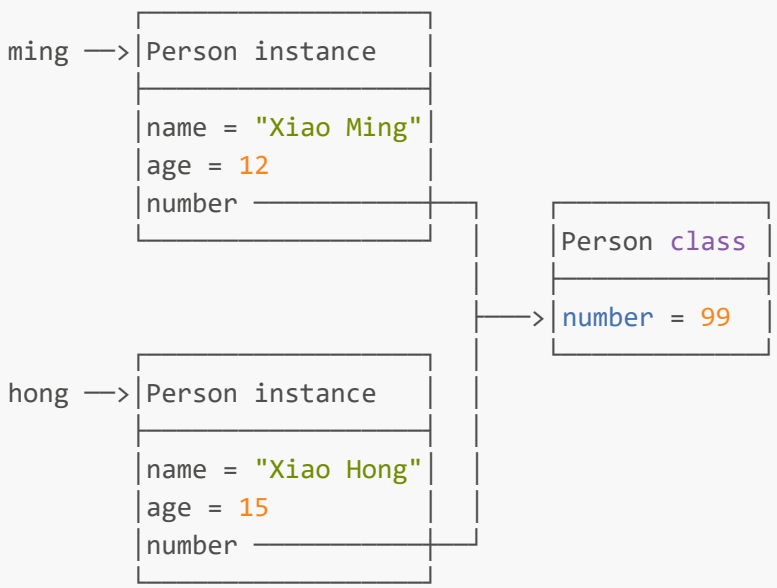
举个例子：

```
class Person {
    public String name;
    public int age;
    // 定义静态字段number:
    public static int number;
}
```

我们来看看下面的代码：

```
// static field
public class Main {
    public static void main(String[] args) {
        Person ming = new Person("Xiao Ming", 12);
        Person hong = new Person("Xiao Hong", 15);
        ming.number = 88;
        System.out.println(hong.number);
        hong.number = 99;
        System.out.println(ming.number);
    }
}

class Person {
    public String name;
    public int age;

    public static int number;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

//Output
88
99
```

对于静态字段，无论修改哪个实例的静态字段，效果都是一样的：所有实例的静态字段都被修改了，原因是静态字段并不属于实例：



虽然实例可以访问静态字段，但是它们指向的其实都是`Person class`的静态字段。所以，所有实例共享一个静态字段。

因此，不推荐用`实例变量.静态字段`去访问静态字段，因为在Java程序中，实例对象并没有静态字段。在代码中，实例对象能访问静态字段只是因为编译器可以根据实例类型自动转换为`类名.静态字段`来访问静态对象。

推荐用类名来访问静态字段。可以把静态字段理解为描述class本身的字段（非实例字段）。对于上面的代码，更好的写法是：

```
//用实例.静态字段访问
Person.number = 99;
System.out.println(Person.number);
```

静态方法

有静态字段，就有静态方法。用`static`修饰的方法称为静态方法。

调用实例方法必须通过一个实例变量，而调用静态方法则不需要实例变量，通过类名就可以调用。静态方法类似其它编程语言的函数。例如：

```
// static method
public class Main {
    public static void main(String[] args) {
        Person.setNumber(99);
        System.out.println(Person.number);
    }
}
```

```
class Person {
    public static int number;

    public static void setNumber(int value) {
        number = value;
    }
}

//Output
99
```

因为静态方法属于class而不属于实例，因此，静态方法内部，无法访问this变量，也无法访问实例字段，它只能访问静态字段。

通过实例变量也可以调用静态方法，但这只是编译器自动帮我们把实例改写成类名而已。

通常情况下，通过实例变量访问静态字段和静态方法，会得到一个编译警告。

静态方法经常用于工具类。例如：

- Arrays.sort()
- Math.random()

静态方法也经常用于辅助方法。注意到Java程序的入口`main()`也是静态方法。

接口的静态字段

因为`interface`是一个纯抽象类，所以它不能定义实例字段。但是，`interface`是可以有静态字段的，并且静态字段必须为`final`类型：

```
//接口的静态字段
public interface Person {
    public static final int MALE = 1;
    public static final int FEMALE = 2;
}
```

实际上，因为`interface`的字段只能是`public static final`类型，所以我们可以把这些修饰符都去掉，上述代码可以简写为：

```
//简写版
public interface Person {
    // 编译器会自动加上public statc final:
    int MALE = 1;
    int FEMALE = 2;
}
```

编译器会自动把该字段变为`public static final`类型。

小结

- **静态字段属于所有实例“共享”的字段，实际上是属于class的字段；**
- **调用静态方法不需要实例，无法访问this，但可以访问静态字段和其他静态方法；**
- **静态方法常用于工具类和辅助方法。**

包

包

在Java中，我们使用package来解决名字冲突。

> 例如：小明既想用gitHub上找来的Arrays类，又想用系统自带的Arrays类。

Java定义了一种名字空间，称之为包：package。一个类总是属于某个包，类名（比如Person）只是一个简写，真正的完整类名是包名.类名。

> 例如：
>
> 小明的Person类存放在包ming下面，因此，完整类名是ming.Person；
>
> 小红的Person类存放在包hong下面，因此，完整类名是hong.Person；
>
> 小军的Arrays类存放在包mr.jun下面，因此，完整类名是mr.jun.Arrays；
>
> JDK的Arrays类存放在包java.util下面，因此，完整类名是java.util.Arrays。

在定义class的时候，我们需要在第一行声明这个class属于哪个包。

> 小明的Person.java文件：

```
package ming; // 申明包名ming

public class Person {
}
```

在Java虚拟机执行的时候，JVM只看完整类名，因此，只要包名不同，类就不同。

包可以是多层结构，用.隔开。

> 例如：java.util。

要特别注意：包没有父子关系。*java.util*和*java.util.zip*是不同的包，两者没有任何*继*承关系。

没有定义包名的class，它使用的是默认包，非常容易引起名字冲突，因此，不推荐不写包名的做法。

我们还需要按照包结构把上面的Java文件组织起来。

> 假设以package_sample作为根目录，src作为源码目录，
>
> 那么所有文件结构就是：

```
package_sample
└─ src
   ├─ hong
   │  └─ Person.java
   │  ming
   │  └─ Person.java
   └─ mr
      └─ jun
         └─ Arrays.java
```

即所有Java文件对应的目录层次要和包的层次一致。

> 编译后的.class文件也需要按照包结构存放。如果使用IDE，把编译后的.class文件放到bin目录下，
>
> 那么，编译的文件结构就是：

```
package_sample
└─ bin
   ├─ hong
   │  └─ Person.class
   │  ming
   │  └─ Person.class
   └─ mr
      └─ jun
         └─ Arrays.class
```

包的作用域

位于同一个包的类，可以访问包作用域的字段和方法。不用`public`、`protected`、`private`修饰的字段和方法就是包作用域。

> 例如，`Person`类定义在`hello`包下面：

```
package hello;

public class Person {
    // 包作用域:
    void hello() {
        System.out.println("Hello!");
    }
}
```

> `Main`类也定义在`hello`包下面：

```
package hello;
```

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.hello(); // 可以调用，因为Main和Person在同一个包
    }
}
```

import

在一个class中，我们总会引用其他的class。

> 例如，小明的ming.Person类，如果要引用小军的mr.jun.Arrays类，他有三种写法：

第一种，直接写出完整类名，

> 例如：

```
// Person.java
package ming;

public class Person {
    public void run() {
        mr.jun.Arrays arrays = new mr.jun.Arrays();
    }
}
```

很显然，每次写完整类名比较痛苦。

第二种写法是用import语句，导入小军的Arrays，然后写简单类名：

```
// Person.java
package ming;

// 导入完整类名:
import mr.jun.Arrays;

public class Person {
    public void run() {
        Arrays arrays = new Arrays();
    }
}
```

在写import的时候，可以使用*，表示把这个包下面的所有class都导入进来（但不包括子包的class）：

```
// Person.java
package ming;
```

```
// 导入mr.jun包的所有class:
import mr.jun.*;

public class Person {
    public void run() {
        Arrays arrays = new Arrays();
    }
}
```

我们一般不推荐这种写法，因为在导入了多个包后，很难看出Arrays类属于哪个包。

还有一种import static的语法，它可以导入可以导入一个类的静态字段和静态方法：

```
package main;

// 导入System类的所有静态字段和静态方法:
import static java.lang.System.*;

public class Main {
    public static void main(String[] args) {
        // 相当于调用System.out.println(…)
        out.println("Hello, world!");
    }
}
```

import static很少使用。

Java编译器最终编译出的.class文件只使用完整类名，因此，在代码中，当编译器遇到一个class名称时：

- **如果是完整类名，就直接根据完整类名查找这个class；**
- **如果是简单类名，按下面的顺序依次查找：**
    - 查找当前package是否存在这个class；
    - 查找import的包是否包含这个class；
    - 查找java.lang包是否包含这个class。

如果按照上面的规则还无法确定类名，则编译报错。

> 例如下面这个例子：

```
// Main.java
package test;

import java.text.Format;

public class Main {
    public static void main(String[] args) {
        java.util.List list; // ok, 使用完整类名 -> java.util.List
        Format format = null; // ok, 使用import的类 -> java.text.Format
```

```
        String s = "hi"; // ok, 使用java.lang包的String -> java.lang.String
        System.out.println(s); // ok, 使用java.lang包的System -> java.lang.System
        MessageFormat mf = null; // 编译错误: 无法找到MessageFormat: MessageFormat
cannot be resolved to a type
    }
}
```

因此，**编写class的时候，编译器会自动帮我们做两个import动作：**

- 默认自动import当前package的其他class；
- 默认自动import java.lang.*。

如果有两个class名称相同，例如，mr.jun.Arrays和java.util.Arrays，(其中类名Arrays相同)那么只能import其中一个，另一个必须写完整类名。

最佳实践

为了避免名字冲突，我们需要确定唯一的包名。**推荐的做法是使用倒置的域名来确保唯一性。**

> 例如：

```
org.apache
org.apache.commons.log
com.liaoxuefeng.sample
```

> 子包就可以根据功能自行命名。

要注意不要和java.lang包的类重名，即自己的类不要使用这些名字：

```
String
System
Runtime
...
```

要注意也不要和JDK常用类重名：

```
java.util.List
java.text.Format
java.math.BigInteger
...
```

小结

- Java内建的package机制是为了避免class命名冲突；

- JDK的核心类使用java.lang包，编译器会自动导入；
- JDK的其它常用类定义在java.util.，*java.math.*，java.text.*，……；
- 包名推荐使用倒置的域名，例如org.apache。

作用域

在Java中，我们经常看到public、protected、private这些修饰符。在Java中，这些修饰符可以用来限定访问作用域。

public

定义为public的class、interface可以被其他任何类访问：

```
package abc;

public class Hello {
    public void hi() {
    }
}
```

> 上面的Hello是public，因此，可以被其他包的类访问：

```
package xyz;

class Main {
    void foo() {
        // Main可以访问Hello
        Hello h = new Hello();
    }
}
```

定义为public的field、method可以被其他类访问，前提是首先有访问class的权限：

```
package abc;

public class Hello {
    public void hi() {
    }
}
```

> 上面的hi()方法是public，可以被其他类调用，前提是首先要能访问Hello类：

```
package xyz;
```

```
class Main {
    void foo() {
        Hello h = new Hello();
        h.hi();
    }
}
```

private

定义为private的field、method无法被其他类访问：

```
package abc;

public class Hello {
    // 不能被其他类调用:
    private void hi() {
    }

    public void hello() {
        this.hi();
    }
}
```

实际上，确切地说，**private访问权限被限定在class的内部，而且与方法声明顺序无关。**

推荐把private方法放到后面，因为public方法定义了类对外提供的功能，阅读代码的时候，应该先关注public方法：

```
package abc;

public class Hello {
    public void hello() {
        this.hi();
    }

    private void hi() {
    }
}
```

由于Java支持嵌套类，如果一个类内部还定义了嵌套类，那么，嵌套类拥有访问private的权限：

```
// private
public class Main {
    public static void main(String[] args) {
        Inner i = new Inner();
        i.hi();
    }
```

```
    // private方法:
    private static void hello() {
        System.out.println("private hello!");
    }

    // 静态内部类:
    static class Inner {
        public void hi() {
            Main.hello();
        }
    }
}

 Run
private hello!
```

定义在一个class内部的class称为嵌套类（nested class），Java支持好几种嵌套类。

protected

protected作用于继承关系。定义为protected的字段和方法可以被子类访问，以及子类的子类：

```
package abc;

public class Hello {
    // protected方法:
    protected void hi() {
    }
}
```

> 上面的protected方法可以被继承的类访问：

```
package xyz;

class Main extends Hello {
    void foo() {
        // 可以访问protected方法:
        hi();
    }
}
```

package

最后，包作用域是指一个类允许访问同一个package的没有public、private修饰的class，以及没有public、protected、private修饰的字段和方法。

```
package abc;
// package权限的类:
class Hello {
    // package权限的方法:
    void hi() {
    }
}
```

只要在同一个包，就可以访问package权限的class、field和method：

```
package abc;

class Main {
    void foo() {
        // 可以访问package权限的类:
        Hello h = new Hello();
        // 可以调用package权限的方法:
        h.hi();
    }
}
```

注意，包名必须完全一致，包没有父子关系，com.apache和com.apache.abc是不同的包。

局部变量

在方法内部定义的变量称为局部变量，局部变量作用域从变量声明处开始到对应的块结束。方法参数也是局部变量。

```
package abc;

public class Hello {
    void hi(String name) { // ①
        String s = name.toLowerCase(); // ②
        int len = s.length(); // ③
        if (len < 10) { // ④
            int p = 10 - len; // ⑤
            for (int i=0; i<10; i++) { // ⑥
                System.out.println(); // ⑦
            } // ⑧
        } // ⑨
    } // ⑩
}
```

> 我们观察上面的hi()方法代码：
>
> 方法参数name是局部变量，它的作用域是整个方法，即①～⑩；

> 变量s的作用域是定义处到方法结束，即②～⑩；
>
> 变量len的作用域是定义处到方法结束，即③～⑩；
>
> 变量p的作用域是定义处到if块结束，即⑤～⑨；
>
> 变量i的作用域是for循环，即⑥～⑧。
>
> 使用局部变量时，应该尽可能把局部变量的作用域缩小，尽可能延后声明局部变量。

final

Java还提供了一个final修饰符。`final`与访问权限不冲突，它有很多作用。

用`final`修饰`class`可以阻止被继承：

```
package abc;

// 无法被继承:
public final class Hello {
    private int n = 0;
    protected void hi(int t) {
        long i = t;
    }
}
```

用`final`修饰`method`可以阻止被子类覆写：

```
package abc;

public class Hello {
    // 无法被覆写:
    protected final void hi() {
    }
}
```

用`final`修饰`field`可以阻止被重新赋值：

```
package abc;

public class Hello {
    private final int n = 0;
    protected void hi() {
        this.n = 1; // error!
    }
}
```

用`final`修饰局部变量可以阻止被重新赋值：

```
package abc;

public class Hello {
    protected void hi(final int t) {
        t = 1; // error!
    }
}
```

最佳实践

如果不确定是否需要public，就不声明为public，即尽可能少地暴露对外的字段和方法。

把方法定义为package权限有助于测试，因为测试类和被测试类只要位于同一个package，测试代码就可以访问被测试类的package权限方法。

一个.java文件只能包含一个public类，但可以包含多个非public类。如果有public类，文件名必须和public类的名字相同。

小结

- Java内建的访问权限包括public、protected、private和package权限；
- Java在方法内部定义的变量是局部变量，局部变量的作用域从变量声明开始，到一个块结束；
- final修饰符不是访问权限，它可以修饰class、field和method；
    - 修饰class：阻止被继承；
    - 修饰method：阻止被子类覆写；
    - 修饰field：阻止被重新赋值；
    - 修饰局部变量：阻止被重新���值；
- 一个.java文件只能包含一个public类，但可以包含多个非public类。
- Java内建的访问权限：

权限类型	block	作用域
public	class method,field	任何类； 任何类，(前提class可以被访问。)
private	class,method,field	本类;嵌套类
public	class,method,field	继承类
package	无public,private,protected修饰的字段	本包内

内部类

Inner Class

如果一个类定义在另一个类的内部，这个类就是Inner Class：

```
class Outer {
    class Inner {
        // 定义了一个Inner Class
    }
}
```

上述定义的Outer是一个普通类，而Inner是一个Inner Class，它与普通类有个最大的不同，就是**Inner Class的实例不能单独存在，必须依附于一个Outer Class的实例**。

> 示例代码如下：

```
// inner class
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer("Nested"); // 实例化一个Outer
        Outer.Inner inner = outer.new Inner(); // 实例化一个Inner
        inner.hello();
    }
}

class Outer {
    private String name;

    Outer(String name) {
        this.name = name;
    }

    class Inner {
        void hello() {
            System.out.println("Hello, " + Outer.this.name);
        }
    }
}
//
 Run
Hello, Nested
```

观察上述代码，要实例化一个Inner，我们必须首先创建一个Outer的实例，然后，调用Outer实例的new来创建Inner实例：

```
Outer.Inner inner = outer.new Inner();
```

这是因为Inner Class除了有一个this指向它自己，还隐含地持有一个Outer Class实例，可以用Outer.this访问这个实例。所以，实例化一个Inner Class不能脱离Outer实例。

Inner Class和普通Class相比，除了能引用Outer实例外，还有一个额外的“特权”，就是可以修改Outer Class的private字段，因为Inner Class的作用域在Outer Class内部，所以能访问Outer Class的private字段和方法。

观察Java编译器编译后的.class文件可以发现，Outer类被编译为Outer.class，而Inner类被编译为Outer$Inner.class。

Anonymous Class

还有一种定义Inner Class的方法，它不需要在Outer Class中明确地定义这个Class，而是**在方法内部，通过匿名类（Anonymous Class）来定义**。

> 示例代码如下：

```
// Anonymous Class
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer("Nested");
        outer.asyncHello();
    }
}

class Outer {
    private String name;

    Outer(String name) {
        this.name = name;
    }

    void asyncHello() {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello, " + Outer.this.name);
            }
        };
        new Thread(r).start();
    }
}

// Run
Hello, Nested
```

观察asyncHello()方法，我们在方法内部实例化了一个Runnable。Runnable本身是接口，接口是不能实例化的，所以这里实际上是定义了一个实现了Runnable接口的匿名类，并且通过new实例化该匿名类，然后转型为Runnable。

在定义匿名类的时候就必须实例化它，定义匿名类的写法如下：

```
Runnable r = new Runnable() {
    // 实现必要的抽象方法...
};
```

匿名类和Inner Class一样，可以访问Outer Class的private字段和方法。之所以我们要定义匿名类，是因为在这里我们通常不关心类名，比直接定义Inner Class可以少写很多代码。

观察Java编译器编译后的.class文件可以发现，Outer类被编译为Outer.class，而匿名类被编译为Outer$1.class。如果有多个匿名类，Java编译器会将每个匿名类依次命名为Outer$1、Outer$2、Outer$3……Static Nested Class

最后一种内部类和Inner Class类似，但是使用static修饰，称为静态内部类（Static Nested Class）：

```
// Static Nested Class
public class Main {
    public static void main(String[] args) {
        Outer.StaticNested sn = new Outer.StaticNested();
        sn.hello();
    }
}

class Outer {
    private static String NAME = "OUTER";

    private String name;

    Outer(String name) {
        this.name = name;
    }

    static class StaticNested {
        void hello() {
            System.out.println("Hello, " + Outer.NAME);
        }
    }
}

//Run
Hello, OUTER
```

用static修饰的内部类和Inner Class有很大的不同，它不再依附于Outer的实例，而是一个完全独立的类，因此无法引用Outer.this，但它可以访问Outer的private静态字段和静态方法。如果把StaticNested移到Outer之外，就失去了访问private的权限。

小结

- Java的内部类可分为Inner Class、Anonymous Class和Static Nested Class三种：
- Inner Class和Anonymous Class本质上是相同的，都必须依附于Outer Class的实例，即隐含地持有Outer.this实例，并拥有Outer Class的private访问权限；
- Static Nested Class是独立类，但拥有Outer Class的private访问权限。

classpath 和 jar

classpath是JVM用到的一个环境变量，它用来指示JVM如何搜索class。

所以，classpath就是一组目录的集合，它设置的搜索路径与操作系统相关。例如，在Windows系统上，用;分隔，带空格的目录用""括起来，可能长这样：

```
C:\work\project1\bin;C:\shared;"D:\My Documents\project1\bin"
```

jar包

如果有很多.class文件，散落在各层目录中，肯定不便于管理。如果能把目录打一个包，变成一个文件，就方便多了。

jar包就是用来干这个事的，**它可以把package组织的目录层级，以及各个目录下的所有文件（包括.class文件和其他文件）都打成一个jar文件**，这样一来，无论是备份，还是发给客户，就简单多了。

jar包实际上就是一个zip格式的压缩文件，而jar包相当于目录。如果我们要执行一个jar包的class，就可以把jar包放到classpath中：

```
java -cp ./hello.jar abc.xyz.Hello
```

这样JVM会自动在hello.jar文件里去搜索某个类。

创建jar包

因为jar包就是zip包，

所以，

- 直接在资源管理器中，找到正确的目录，
- 点击右键，
- 在弹出的快捷菜单中选择“发送到”，
- “压缩(zipped)文件夹”，就制作了一个zip文件。
- 然后，把后缀从.zip改为.jar，
- 一个jar包就创建成功。

> 假设编译输出的目录结构是这样：

```
package_sample
└─ bin
   ├─ hong
   │  └─ Person.class
   │  ming
   │  └─ Person.class
   └─ mr
      └─ jun
         └─ Arrays.class
```

这里需要特别注意的是，`jar`**包里的第一层目录，不能是**`bin`**，而应该是**`hong`、`ming`、`mr`。

模块

小结

- Java 9引入的模块目的是为了管理依赖；
- 使用模块可以按需打包JRE；
- 使用模块对类的访问权限有了进一步限制。