面向对象编程

本节目标

- 包
- 继承
- 组合
- 多态
- 抽象类
- 接口

包

包 (package) 是组织类的一种方式.

使用包的主要目的是保证类的唯一性.

> 例如, 你在代码中写了一个 Test 类. 然后你的同事也可能写一个 Test 类. 如果出现两个同名的类, 就会冲突, 导致代码不能编译通过.

导入包中的类

Java 中已经提供了很多现成的类供我们使用. 例如

```
public class Test {
    public static void main(String[] args) {
        java.util.Date date = new java.util.Date();
        // 得到一个毫秒级别的时间戳
        System.out.println(date.getTime());
    }
}
```

可以使用 `java.util.Date` 这种方式引入 `java.util` 这个包中的 `Date` 类.

但是这种写法比较麻烦一些, 可以使用 `import` 语句导入包.

```java
import java.util.Date;
public class Test {
    public static void main(String[] args) {
        Date date = new Date();
        // 得到一个毫秒级别的时间戳
        System.out.println(date.getTime());
    }
}
```

如果需要使用 `java.util` 中的其他类, 可以使用 `import java.util.*`

```java
import java.util.*;
public class Test {
    public static void main(String[] args) {
        Date date = new Date();
        // 得到一个毫秒级别的时间戳
        System.out.println(date.getTime());
    }
}
```

但是我们更建议显式的指定要导入的类名. 否则还是容易出现冲突的情况.

```java
import java.util.*;
import java.sql.*;
public class Test {
    public static void main(String[] args) {
        // util 和 sql 中都存在一个 Date 这样的类, 此时就会出现歧义, 编译出错
        Date date = new Date();
        System.out.println(date.getTime());
    }
}

// 编译出错
Error:(5, 9) java: 对Date的引用不明确
  java.sql 中的类 java.sql.Date 和 java.util 中的类 java.util.Date 都匹配
```

在这种情况下需要使用完整的类名

```java
import java.util.*;
import java.sql.*;
public class Test {
    public static void main(String[] args) {
        java.util.Date date = new java.util.Date();
        System.out.println(date.getTime());
    }
}
```

注意事项: `import` 和 C++ 的 `#include` 差别很大. C++ 必须 `#include` 来引入其他文件内容, 但是 Java 不需要. import 只是为了写代码的时候更方便. `import` 更类似于 C++ 的 `namespace` 和 `using`

静态导入

使用 `import static` 可以导入包中的静态的方法和字段.

```java
import static java.lang.System.*;
public class Test {
    public static void main(String[] args) {
        out.println("hello");
    }
}
```

使用这种方式可以更方便的写一些代码, 例如

```java
import static java.lang.Math.*;

public class Test {
    public static void main(String[] args) {
        double x = 30;
        double y = 40;
        // 静态导入的方式写起来更方便一些.
        // double result = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
        double result = sqrt(pow(x, 2) + pow(y, 2));
        System.out.println(result);
    }
}
```

将类放到包中

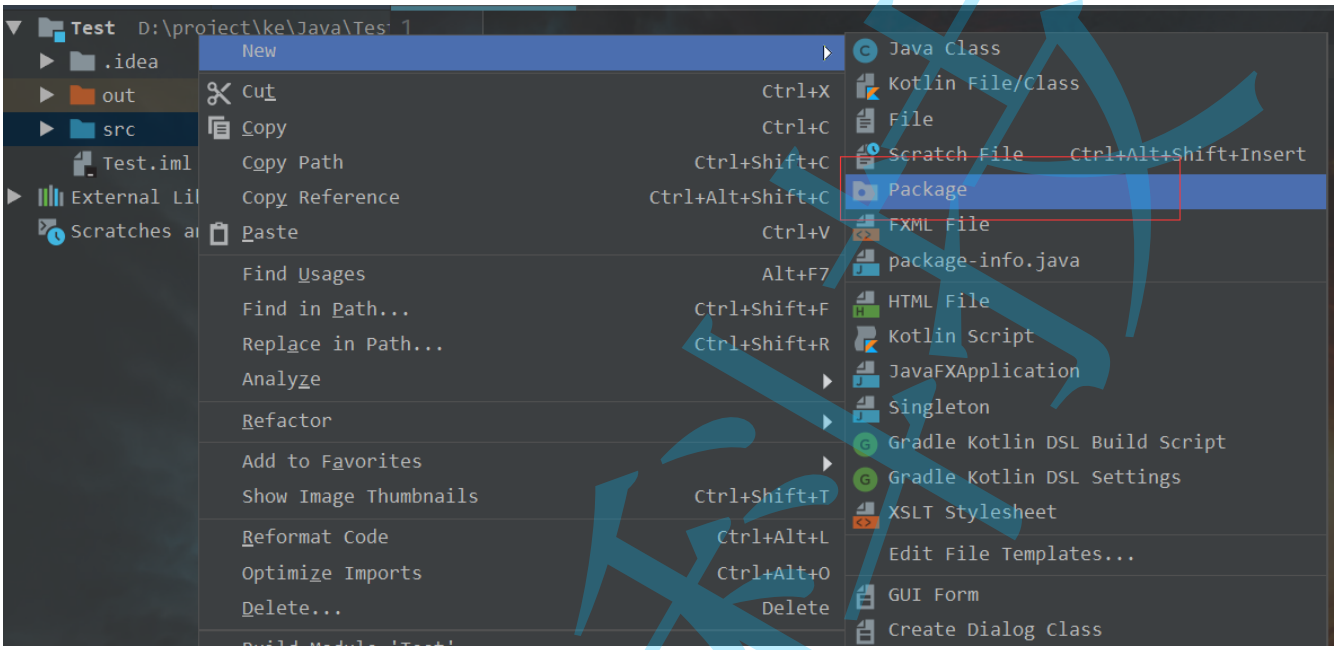
基本规则

- 在文件的最上方加上一个 package 语句指定该代码在哪个包中.
- 包名需要尽量指定成唯一的名字, 通常会用公司的域名的颠倒形式(例如 `com.bit.demo1`).

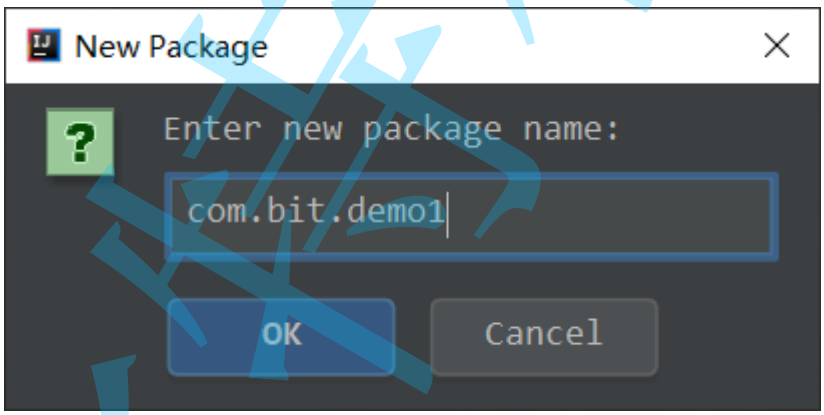
- 包名要和代码路径相匹配. 例如创建 `com.bit.demo1` 的包, 那么会存在一个对应的路径 `com/bit/demo1` 来存储代码.
- 如果一个类没有 package 语句, 则该类被放到一个默认包中.

操作步骤

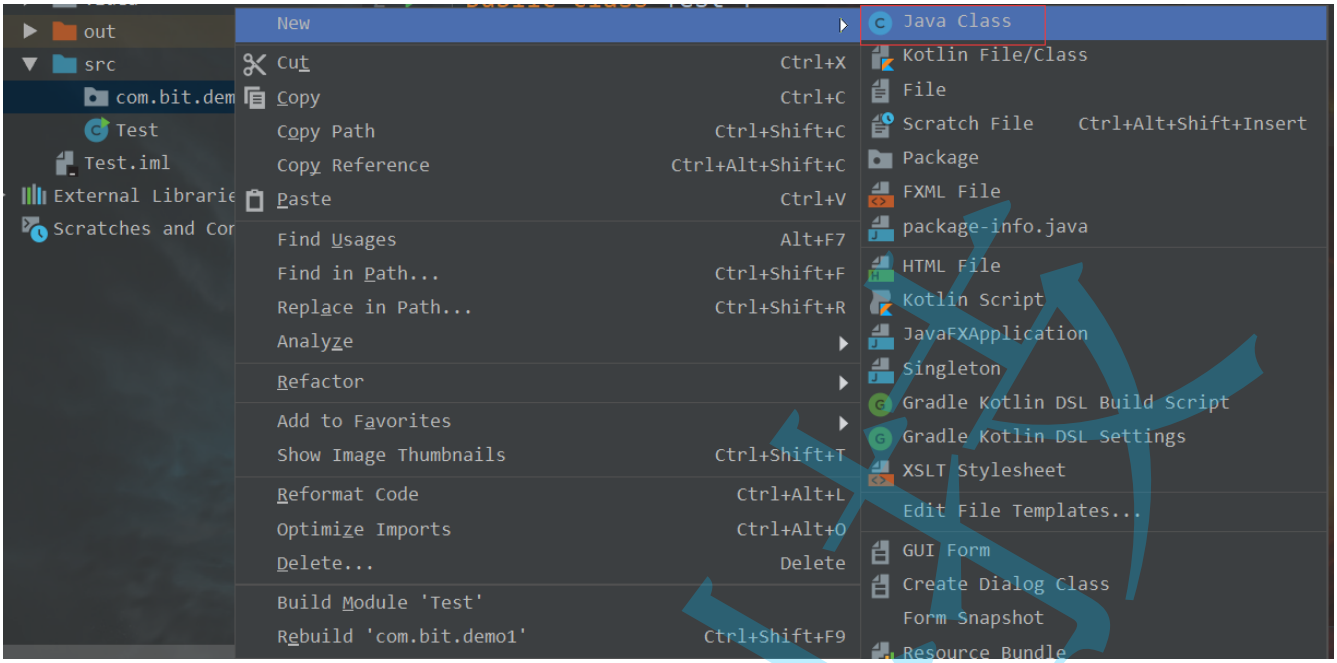
1) 在 IDEA 中先新建一个包: 右键 src -> 新建 -> 包



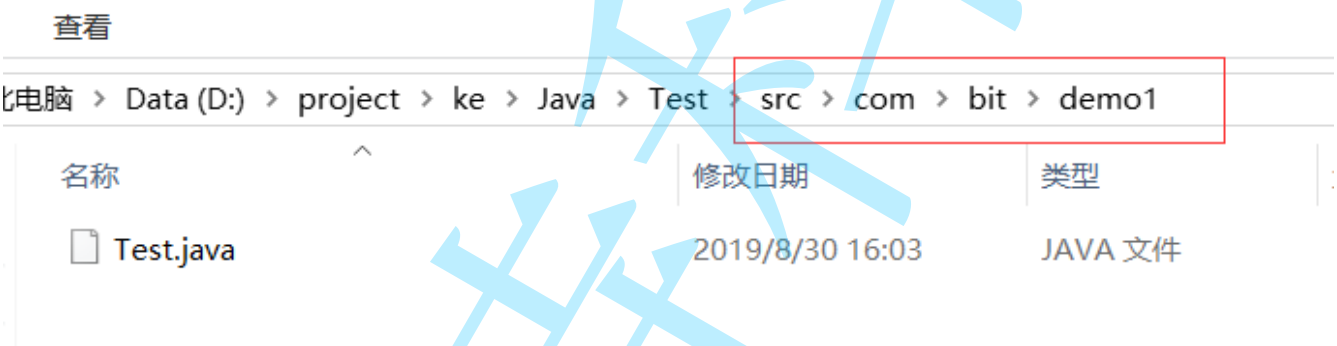
2) 在弹出的对话框中输入包名, 例如 `com.bit.demo1`



3) 在包中创建类, 右键包名 -> 新建 -> 类, 然后输入类名即可.



4) 此时可以看到我们的磁盘上的目录结构已经被 IDEA 自动创建出来了



5) 同时我们也看到了, 在新创建的 `Test.java` 文件的最上方, 就出现了一个 package 语句

包的访问权限控制

我们已经了解了类中的 public 和 private. private 中的成员只能被类的内部使用.

如果某个成员不包含 public 和 private 关键字, 此时这个成员可以在包内部的其他类使用, 但是不能在包外部的类使用.

下面的代码给了一个示例. Demo1 和 Demo2 是同一个包中, Test 是其他包中.

Demo1.java

```java
package com.bit.demo;

public class Demo1 {
    int value = 0;
}
```

Demo2.java

```java
package com.bit.demo;

public class Demo2 {
    public static void Main(String[] args) {
        Demo1 demo = new Demo1();
        System.out.println(demo.value);
    }
}

// 执行结果，能够访问到 value 变量
10
```

Test.java

```java
import com.bit.demo.Demo1;

public class Test {
    public static void main(String[] args) {
        Demo1 demo = new Demo1();
        System.out.println(demo.value);
    }
}

// 编译出错
Error:(6, 32) java: value在com.bit.demo.Demo1中不是公共的；无法从外部程序包中对其进行访问
```

常见的系统包

1. java.lang:系统常用基础类(String、Object),此包从JDK1.1后自动导入。
2. java.lang.reflect:java 反射编程包;
3. java.net:进行网络编程开发包。
4. java.sql:进行数据库开发的支持包。
5. java.util:是java提供的工具程序包。(集合类等) **非常重要**
6. java.io:I/O编程开发包。

继承

背景

代码中创建的类, 主要是为了抽象现实中的一些事物(包含属性和方法).

有的时候客观事物之间就存在一些关联关系, 那么在表示成类和对象的时候也会存在一定的关联.

例如, 设计一个类表示动物

> 注意, 我们可以给每个类创建一个单独的 java 文件. 类名必须和 .java 文件名匹配(大小写敏感).

```
// Animal.java
public class Animal {
    public String name;

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

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }
}

// Cat.java
class Cat {
    public String name;

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

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }
}

// Bird.java
class Bird {
    public String name;

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

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }

    public void fly() {
        System.out.println(this.name + "正在飞 ㄟ( ▔, ▔ )ㄏ");
    }
}
```

这个代码我们发现其中存在了大量的冗余代码.

仔细分析, 我们发现 `Animal` 和 `Cat` 以及 `Bird` 这几个类中存在一定的关联关系:

- 这三个类都具备一个相同的 eat 方法, 而且行为是完全一样的.
- 这三个类都具备一个相同的 name 属性, 而且意义是完全一样的.
- 从逻辑上讲, Cat 和 Bird 都是一种 Animal (is - a 语义).

此时我们就可以让 Cat 和 Bird 分别继承 Animal 类, 来达到代码重用的效果.

此时, Animal 这样被继承的类, 我们称为 **父类** , **基类** 或 **超类**, 对于像 Cat 和 Bird 这样的类, 我们称为 **子类**, **派生类**

和现实中的儿子继承父亲的财产类似, 子类也会继承父类的字段和方法, 以达到代码重用的效果.

语法规则

基本语法

```
class 子类 extends 父类 {

}
```

- 使用 extends 指定父类.
- Java 中一个子类只能继承一个父类 (而C++/Python等语言支持多继承).
- 子类会继承父类的所有 public 的字段和方法.
- 对于父类的 private 的字段和方法, 子类中是无法访问的.
- 子类的实例中, 也包含着父类的实例. 可以使用 super 关键字得到父类实例的引用.

对于上面的代码, 可以使用继承进行改进. 此时我们让 Cat 和 Bird 继承自 Animal 类, 那么 Cat 在定义的时候就不必再写 name 字段和 eat 方法.

```
class Animal {
    public String name;

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

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }
}

class Cat extends Animal {
    public Cat(String name) {
        // 使用 super 调用父类的构造方法.
        super(name);
    }
}

class Bird extends Animal {
    public Bird(String name) {
```

```
        super(name);
    }

    public void fly() {
        System.out.println(this.name + "正在飞 ㅅ(￣ᴗ￣)ㅅ");
    }
}

public class Test {
    public static void main(String[] args) {
        Cat cat = new Cat("小黑");
        cat.eat("猫粮");
        Bird bird = new Bird("圆圆");
        bird.fly();
    }
}
```

> extends 英文原意指 "扩展". 而我们所写的类的继承, 也可以理解成基于父类进行代码上的 "扩展".
>
> 例如我们写的 Bird 类, 就是在 Animal 的基础上扩展出了 fly 方法.

如果我们把 name 改成 private, 那么此时子类就不能访问了.

```
class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    public void fly() {
        System.out.println(this.name + "正在飞 ㅅ(￣ᴗ￣)ㅅ");
    }
}

// 编译出错
Error:(19, 32) java: name 在 Animal 中是 private 访问控制
```

protected 关键字

刚才我们发现, 如果把字段设为 private, 子类不能访问. 但是设成 public, 又违背了我们 "封装" 的初衷.

两全其美的办法就是 protected 关键字.

- 对于类的调用者来说, protected 修饰的字段和方法是不能访问的
- 对于类的 **子类** 和 **同一个包的其他类** 来说, protected 修饰的字段和方法是可以访问的

```
// Animal.java
public class Animal {
    protected String name;

    public Animal(String name) {
```

```
        this.name = name;
    }

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }
}

// Bird.java
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    public void fly() {
        // 对于父类的 protected 字段，子类可以正确访问
        System.out.println(this.name + "正在飞 ㅅ(￣◡￣)ㅅ");
    }
}

// Test.java 和 Animal.java 不在同一个 包 之中了.
public class Test {
    public static void main(String[] args) {
        Animal animal = new Animal("小动物");
        System.out.println(animal.name); // 此时编译出错，无法访问 name
    }
}
```

小结: Java 中对于字段和方法共有四种访问权限

- private: 类内部能访问, 类外部不能访问
- 默认(也叫包访问权限): 类内部能访问, 同一个包中的类可以访问, 其他类不能访问.
- protected: 类内部能访问, 子类和同一个包中的类可以访问, 其他类不能访问.
- public : 类内部和类的调用者都能访问

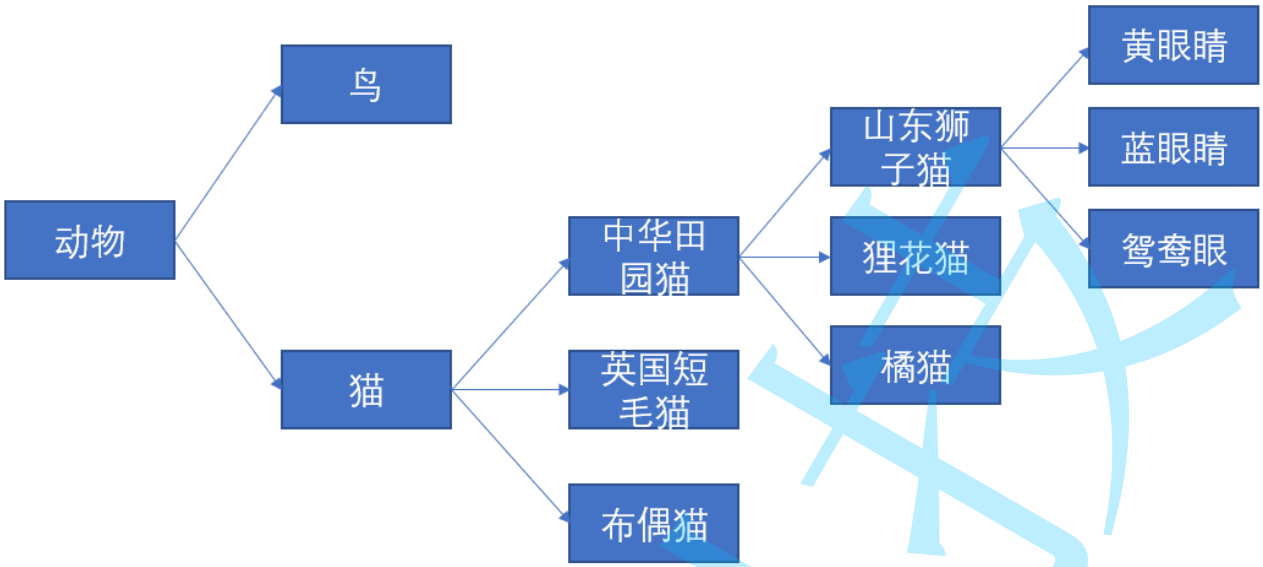
No	范围	private	default	protected	public
1	同一包中的同一类	✓	✓	✓	✓
2	同一包中的不同类		✓	✓	✓
3	不同包中的子类			✓	✓
4	不同包中的非子类				✓

> **什么时候下用哪一种呢?**
>
> 我们希望类要尽量做到 "封装", 即隐藏内部实现细节, 只暴露出 **必要** 的信息给类的调用者.
>
> 因此我们在使用的时候应该尽可能的使用 **比较严格** 的访问权限. 例如如果一个方法能用 private, 就尽量不要用 public.
>
> 另外, 还有一种 **简单粗暴** 的做法: 将所有的字段设为 private, 将所有的方法设为 public. 不过这种方式属于是对访问权限的滥用, 还是更希望同学们能写代码的时候认真思考, 该类提供的字段方法到底给 "谁" 使用(是类内部自己用, 还是类的调用者使用, 还是子类使用).

更复杂的继承关系

刚才我们的例子中, 只涉及到 Animal, Cat 和 Bird 三种类. 但是如果情况更复杂一些呢?

针对 Cat 这种情况, 我们可能还需要表示更多种类的猫~



这个时候使用继承方式来表示, 就会涉及到更复杂的体系.

```
// Animal.java
public Animal {
    ...
}

// Cat.java
public Cat extends Animal {
    ...
}

// ChineseGardenCat.java
public ChineseGardenCat extends Cat {
    ...
}

// OrangeCat.java
public Orange extends ChineseGardenCat {
    ...
}
......
```

如刚才这样的继承方式称为多层继承, 即子类还可以进一步的再派生出新的子类.

> 时刻牢记, 我们写的类是现实事物的抽象. 而我们真正在公司中所遇到的项目往往业务比较复杂, 可能会涉及到一系列复杂的概念, 都需要我们使用代码来表示, 所以我们真实项目中所写的类也会有很多. 类之间的关系也会更加复杂.
>
> 但是即使如此, 我们并不希望类之间的继承层次太复杂. 一般我们不希望出现超过三层的继承关系. 如果继承层次太多, 就需要考虑对代码进行重构了.
>
> 如果想从语法上进行限制继承, 就可以使用 final 关键字

final 关键字

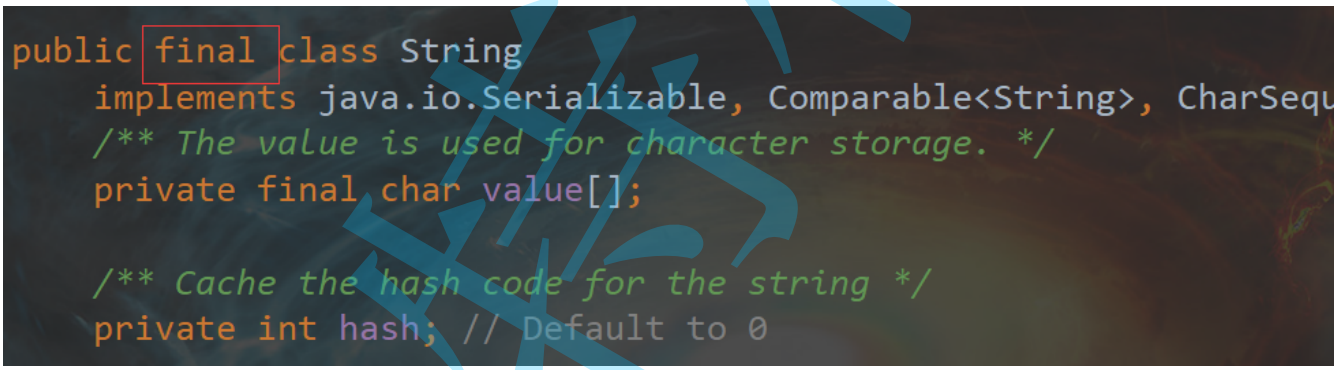
曾经我们学习过 final 关键字, 修饰一个变量或者字段的时候, 表示 **常量** (不能修改).

```
final int a = 10;
a = 20;  // 编译出错
```

final 关键字也能修饰类, 此时表示被修饰的类就不能被继承.

```
final public class Animal {
    ...
}

public class Bird extends Animal {
    ...
}

// 编译出错
Error:(3, 27) java: 无法从最终com.bit.Animal进行继承
```

> final 关键字的功能是 **限制** 类被继承
>
> "限制" 这件事情意味着 "不灵活". 在编程中, 灵活往往不见得是一件好事. 灵活可能意味着更容易出错.
>
> 是用 final 修饰的类被继承的时候, 就会编译报错, 此时就可以提示我们这样的继承是有悖这个类设计的初衷的.



我们平时是用的 String 字符串类, 就是用 final 修饰的, 不能被继承.

组合

和继承类似, 组合也是一种表达类之间关系的方式, 也是能够达到代码重用的效果.

例如表示一个学校:

```
public class Student {
    ...
}

public class Teacher {
    ...
}

public class School {
    public Student[] students;
    public Teacher[] teachers;
}
```

组合并没有涉及到特殊的语法(诸如 extends 这样的关键字), 仅仅是将一个类的实例作为另外一个类的字段.

这是我们设计类的一种常用方式之一.

组合表示 `has - a` 语义

在刚才的例子中, 我们可以理解成一个学校中 "包含" 若干学生和教师.

继承表示 `is - a` 语义

在上面的 "动物和猫" 的例子中, 我们可以理解成一只猫也 "是" 一种动物.

大家要注意体会两种语义的区别.

多态

向上转型

在刚才的例子中, 我们写了形如下面的代码

```
Bird bird = new Bird("圆圆");
```

这个代码也可以写成这个样子

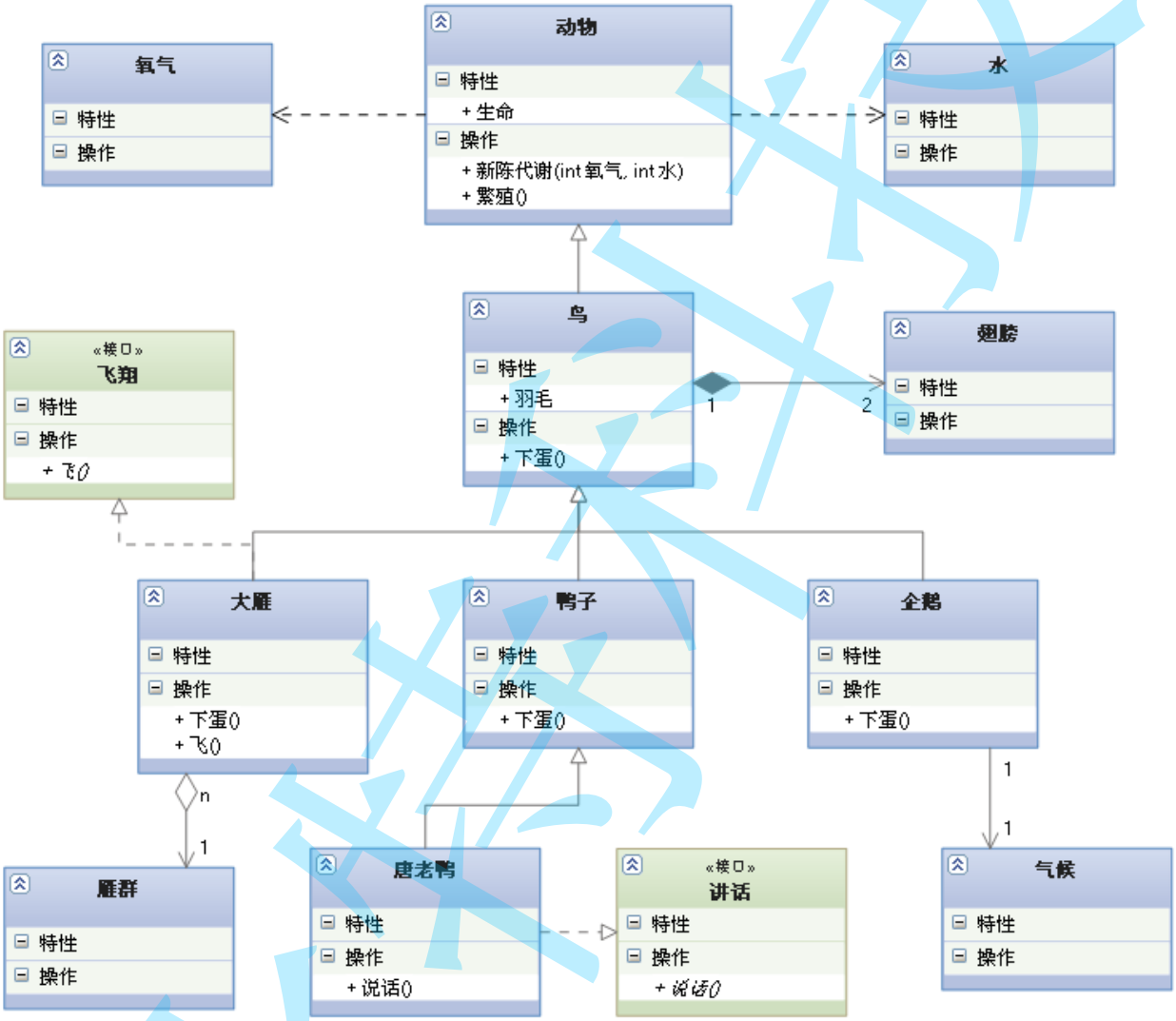
```
Bird bird = new Bird("圆圆");
Animal bird2 = bird;

// 或者写成下面的方式
Animal bird2 = new Bird("圆圆");
```

此时 bird2 是一个父类 (Animal) 的引用, 指向一个子类 (Bird) 的实例. 这种写法称为 **向上转型**.

> 向上转型这样的写法可以结合 `is - a` 语义来理解.
>
> 例如, 我让我媳妇去喂圆圆, 我就可以说, "媳妇你喂小鸟了没?", 或者 "媳妇你喂鹦鹉了没?"

> 因为圆圆确实是一只鹦鹉, 也确实是一只小鸟~~

> **为啥叫 "向上转型"?**
>
> 在面向对象程序设计中, 针对一些复杂的场景(很多类, 很复杂的继承关系), 程序猿会画一种 UML 图的方式来表示类之间的关系. 此时父类通常画在子类的上方. 所以我们就称为 "向上转型" , 表示往父类的方向转.
>
> 注意: 关于 UML 图的规则我们课堂上不详细讨论, 有兴趣的同学自己看看即可.



向上转型发生的时机:

- 直接赋值
- 方法传参
- 方法返回

直接赋值的方式我们已经演示了. 另外两种方式和直接赋值没有本质区别.

方法传参

```
public class Test {
    public static void main(String[] args) {
        Bird bird = new Bird("圆圆");
        feed(bird);
    }

    public static void feed(Animal animal) {
        animal.eat("谷子");
    }
}

// 执行结果
圆圆正在吃谷子
```

此时形参 animal 的类型是 Animal (基类), 实际上对应到 Bird (父类) 的实例.

方法返回

```
public class Test {
    public static void main(String[] args) {
        Animal animal = findMyAnimal();
    }

    public static Animal findMyAnimal() {
        Bird bird = new Bird("圆圆");
        return bird;
    }
}
```

此时方法 `findMyAnimal` 返回的是一个 Animal 类型的引用, 但是实际上对应到 Bird 的实例.

动态绑定

当子类和父类中出现同名方法的时候, 再去调用会出现什么情况呢?

对前面的代码稍加修改, 给 Bird 类也加上同名的 eat 方法, 并且在两个 eat 中分别加上不同的日志.

```
// Animal.java
public class Animal {
    protected String name;

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

    public void eat(String food) {
        System.out.println("我是一只小动物");
        System.out.println(this.name + "正在吃" + food);
    }
```

```
}

// Bird.java
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    public void eat(String food) {
        System.out.println("我是一只小鸟");
        System.out.println(this.name + "正在吃" + food);
    }
}

// Test.java
public class Test {
    public static void main(String[] args) {
        Animal animal1 = new Animal("圆圆");
        animal1.eat("谷子");
        Animal animal2 = new Bird("扁扁");
        animal2.eat("谷子");
    }
}

// 执行结果
我是一只小动物
圆圆正在吃谷子
我是一只小鸟
扁扁正在吃谷子
```

此时, 我们发现:

- animal1 和 animal2 虽然都是 `Animal` 类型的引用, 但是 animal1 指向 `Animal` 类型的实例, animal2 指向 `Bird` 类型的实例.
- 针对 animal1 和 animal2 分别调用 eat 方法, 发现 `animal1.eat()` 实际调用了父类的方法, 而 `animal2.eat()` 实际调用了子类的方法.

因此, 在 Java 中, 调用某个类的方法, 究竟执行了哪段代码 (是父类方法的代码还是子类方法的代码) , 要看究竟这个引用指向的是父类对象还是子类对象. 这个过程是程序运行时决定的(而不是编译期), 因此称为 **动态绑定**.

方法重写

针对刚才的 eat 方法来说:

子类实现父类的同名方法, 并且参数的类型和个数完全相同, 这种情况称为 **覆写/重写/覆盖(Override)**.

关于重写的注意事项

1. 重写和重载完全不一样. 不要混淆(思考一下, 重载的规则是啥?)
2. 普通方法可以重写, static 修饰的静态方法不能重写.

3. 重写中子类的方法的访问权限不能低于父类的方法访问权限.
4. 重写的方法返回值类型不一定和父类的方法相同(但是建议最好写成相同, 特殊情况除外).

方法权限示例: 将子类的 eat 改成 private

```
// Animal.java
public class Animal {
    public void eat(String food) {
        ...
    }
}

// Bird.java
public class Bird extends Animal {
    // 将子类的 eat 改成 private
    private void eat(String food) {
        ...
    }
}

// 编译出错
Error:(8, 10) java: com.bit.Bird中的eat(java.lang.String)无法覆盖com.bit.Animal中的
eat(java.lang.String)
  正在尝试分配更低的访问权限; 以前为public
```

另外, 针对重写的方法, 可以使用 `@Override` 注解来显式指定.

```
// Bird.java
public class Bird extends Animal {
    @Override
    private void eat(String food) {
        ...
    }
}
```

有了这个注解能帮我们进行一些合法性校验. 例如不小心将方法名字拼写错了 (比如写成 aet), 那么此时编译器就会发现父类中没有 aet 方法, 就会编译报错, 提示无法构成重写.

我们推荐在代码中进行重写方法时**显式加上** @Override 注解.

> 关于注解的详细内容, 我们会在后面的章节再详细介绍.

小结: 重载和重写的区别.

No	区别	重载（overload）	覆写(override)
1	概念	方法名称相同，参数的类型及个数不同	方法名称、返回值类型、参数的类型及个数完全相同
2	范围	一个类	继承关系
3	限制	没有权限要求	被覆写的方法不能拥有比父类更严格的访问控制权限

理解多态

有了面的向上转型, 动态绑定, 方法重写之后, 我们就可以使用 **多态(polypeptide)** 的形式来设计程序了.

我们可以写一些只关注父类的代码, 就能够同时兼容各种子类的情况.

代码示例: 打印多种形状

```
class Shape {
    public void draw() {
        // 啥都不用干
    }
}

class Cycle extends Shape {
    @Override
    public void draw() {
        System.out.println("○");
    }
}

class Rect extends Shape {
    @Override
    public void draw() {
        System.out.println("□");
```

```
    }
}

class Flower extends Shape {
    @Override
    public void draw() {
        System.out.println("♣");
    }
}

/////////////////////////我是分割线/////////////////////

// Test.java
public class Test {
    public static void main(String[] args) {
        Shape shape1 = new Flower();
        Shape shape2 = new Cycle();
        Shape shape3 = new Rect();
        drawMap(shape1);
        drawMap(shape2);
        drawMap(shape3);
    }
    // 打印单个图形
    public static void drawShape(Shape shape) {
        shape.draw();
    }
}
```

在这个代码中, 分割线上方的代码是 **类的实现者** 编写的, 分割线下方的代码是 **类的调用者** 编写的.

当类的调用者在编写 `drawMap` 这个方法的时候, 参数类型为 Shape (父类), 此时在该方法内部并**不知道, 也不关注**当前的 shape 引用指向的是哪个类型(哪个子类)的实例. 此时 shape 这个引用调用 draw 方法可能会有多种不同的表现(和 shape 对应的实例相关), 这种行为就称为 **多态**.

> 多态顾名思义, 就是 "一个引用, 能表现出多种不同形态"
>
> 举个具体的例子. 汤老湿家里养了两只鹦鹉(圆圆和扁扁)和一个小孩(核弹). 我媳妇管他们都叫 "儿子". 这时候我对我媳妇说, "你去喂喂你儿子去". 那么如果这里的 "儿子" 指的是鹦鹉, 我媳妇就要喂鸟粮; 如果这里的 "儿子" 指的是核弹, 我媳妇就要喂馒头.
>
> 那么如何确定这里的 "儿子" 具体指的是啥? 那就是根据我和媳妇对话之间的 "上下文".
>
> 代码中的多态也是如此. 一个引用到底是指向父类对象, 还是某个子类对象(可能有多个), 也是要根据上下文的代码来确定.
>
> PS: 大家可以根据汤老湿说话的语气推测一下在家里的家庭地位.

使用多态的好处是什么?

1) 类调用者对类的使用成本进一步降低.

- 封装是让类的调用者不需要知道类的实现细节.
- 多态能让类的调用者连这个类的类型是什么都不必知道, 只需要知道这个对象具有某个方法即可.

因此, 多态可以理解成是封装的更进一步, 让类调用者对类的使用成本进一步降低.

这也贴合了 <<代码大全>> 中关于 "管理代码复杂程度" 的初衷.

2) 能够降低代码的 "圈复杂度", 避免使用大量的 if - else

例如我们现在需要打印的不是一个形状了, 而是多个形状. 如果不基于多态, 实现代码如下:

```
public static void drawShapes() {
    Rect rect = new Rect();
    Cycle cycle = new Cycle();
    Flower flower = new Flower();
    String[] shapes = {"cycle", "rect", "cycle", "rect", "flower"};

    for (String shape : shapes) {
        if (shape.equals("cycle")) {
            cycle.draw();
        } else if (shape.equals("rect")) {
            rect.draw();
        } else if (shape.equals("flower")) {
            flower.draw();
        }
    }
}
```

如果使用使用多态, 则不必写这么多的 if - else 分支语句, 代码更简单.

```
public static void drawShapes() {
    // 我们创建了一个 Shape 对象的数组.
    Shape[] shapes = {new Cycle(), new Rect(), new Cycle(),
                      new Rect(), new Flower()};
    for (Shape shape : shapes) {
        shape.draw();
    }
}
```

> 什么叫 "圈复杂度" ?
>
> 圈复杂度是一种描述一段代码复杂程度的方式. 一段代码如果平铺直叙, 那么就比较简单容易理解. 而如果有很多的条件分支或者循环语句, 就认为理解起来更复杂.
>
> 因此我们可以简单粗暴的计算一段代码中条件语句和循环语句出现的个数, 这个个数就称为 "圈复杂度". 如果一个方法的圈复杂度太高, 就需要考虑重构.
>
> 不同公司对于代码的圈复杂度的规范不一样. 一般不会超过 10 .

3) 可扩展能力更强.

如果要新增一种新的形状, 使用多态的方式代码改动成本也比较低.

```
class Triangle extends Shape {
    @Override
    public void draw() {
        System.out.println("△");
    }
}
```

对于类的调用者来说(drawShapes方法), 只要创建一个新类的实例就可以了, 改动成本很低.

而对于不用多态的情况, 就要把 drawShapes 中的 if - else 进行一定的修改, 改动成本更高.

向下转型

向上转型是子类对象转成父类对象, 向下转型就是父类对象转成子类对象. 相比于向上转型来说, 向下转型没那么常见, 但是也有一定的用途.

```
// Animal.java
public class Animal {
    protected String name;

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

    public void eat(String food) {
        System.out.println("我是一只小动物");
        System.out.println(this.name + "正在吃" + food);
    }
}

// Bird.java
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    public void eat(String food) {
        System.out.println("我是一只小鸟");
        System.out.println(this.name + "正在吃" + food);
    }

    public void fly() {
        System.out.println(this.name + "正在飞");
    }
}
```

接下来是我们熟悉的操作

```
Animal animal = new Bird("圆圆");
animal.eat("谷子");

// 执行结果
圆圆正在吃谷子
```

接下来我们尝试让圆圆飞起来

```
animal.fly();

// 编译出错
找不到 fly 方法
```

注意事项

编译过程中, animal 的类型是 Animal, 此时编译器只知道这个类中有一个 eat 方法, 没有 fly 方法.

虽然 animal 实际引用的是一个 Bird 对象, 但是编译器是以 animal 的类型来查看有哪些方法的.

对于 `Animal animal = new Bird("圆圆")` 这样的代码,

- 编译器检查有哪些方法存在, 看的是 Animal 这个类型
- 执行时究竟执行父类的方法还是子类的方法, 看的是 Bird 这个类型.

那么想实现刚才的效果, 就需要向下转型.

```
// (Bird) 表示强制类型转换
Bird bird = (Bird)animal;
bird.fly();

// 执行结果
圆圆正在飞
```

但是这样的向下转型有时是不太可靠的. 例如

```
Animal animal = new Cat("小猫");
Bird bird = (Bird)animal;
bird.fly();

// 执行结果, 抛出异常
Exception in thread "main" java.lang.ClassCastException: Cat cannot be cast to Bird
    at Test.main(Test.java:35)
```

animal 本质上引用的是一个 Cat 对象, 是不能转成 Bird 对象的. 运行时就会抛出异常.

所以, 为了让向下转型更安全, 我们可以先判定一下看看 animal 本质上是不是一个 Bird 实例, 再来转换

```
Animal animal = new Cat("小猫");
if (animal instanceof Bird) {
    Bird bird = (Bird)animal;
    bird.fly();
}
```

`instanceof` 可以判定一个引用是否是某个类的实例. 如果是, 则返回 true. 这时再进行向下转型就比较安全了.

super 关键字

前面的代码中由于使用了重写机制, 调用到的是子类的方法. 如果需要在子类内部调用父类方法怎么办? 可以使用 super 关键字.

super 表示获取到父类实例的引用. 涉及到两种常见用法.

1) 使用了 super 来调用父类的构造器(这个代码前面已经写过了)

```
public Bird(String name) {
    super(name);
}
```

2) 使用 super 来调用父类的普通方法

```
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    @Override
    public void eat(String food) {
        // 修改代码, 让子调用父类的接口.
        super.eat(food);
        System.out.println("我是一只小鸟");
        System.out.println(this.name + "正在吃" + food);
    }
}
```

在这个代码中, 如果在子类的 eat 方法中直接调用 eat (不加super), 那么此时就认为是调用子类自己的 eat (也就是递归了). 而加上 super 关键字, 才是调用父类的方法.

注意 super 和 this 功能有些相似, 但是还是要注意其中的区别.

No	区别	this	super
1	概念	访问本类中的属性和方法	由子类访问父类中的属性、方法
2	查找范围	先查找本类，如果本类没有就调用父类	不查找本类而直接调用不累定义
3	特殊	表示当前对象	无

在构造方法中调用重写的方法(一个坑)

一段有坑的代码. 我们创建两个类, B 是父类, D 是子类. D 中重写 func 方法. 并且在 B 的构造方法中调用 func

```
class B {
    public B() {
        // do nothing
        func();
    }

    public void func() {
        System.out.println("B.func()");
    }
}

class D extends B {
    private int num = 1;
    @Override
    public void func() {
        System.out.println("D.func() " + num);
    }
}

public class Test {
    public static void main(String[] args) {
        D d = new D();
```

```
    }
}

// 执行结果
D.func() 0
```

- 构造 D 对象的同时, 会调用 B 的构造方法.
- B 的构造方法中调用了 func 方法, 此时会触发动态绑定, 会调用到 D 中的 func
- 此时 D 对象自身还没有构造, 此时 num 处在未初始化的状态, 值为 0.

结论: "用尽量简单的方式使对象进入可工作状态", 尽量不要在构造器中调用方法(如果这个方法被子类重写, 就会触发动态绑定, 但是此时子类对象还没构造完成), 可能会出现一些隐藏的但是又极难发现的问题.

总结

多态是面向对象程序设计中比较难理解的部分. 我们会在后面的抽象类和接口中进一步体会多态的使用. 重点是多态带来的编码上的好处.

另一方面, 如果抛开 Java, 多态其实是一个更广泛的概念, 和 "继承" 这样的语法并没有必然的联系.

- C++ 中的 "动态多态" 和 Java 的多态类似. 但是 C++ 还有一种 "静态多态"(模板), 就和继承体系没有关系了.
- Python 中的多态体现的是 "鸭子类型", 也和继承体系没有关系.
- Go 语言中没有 "继承" 这样的概念, 同样也能表示多态.

无论是哪种编程语言, 多态的核心都是让调用者**不必关注对象的具体类型**. 这是降低用户使用成本的一种重要方式.

抽象类

语法规则

在刚才的打印图形例子中, 我们发现, 父类 Shape 中的 draw 方法好像并没有什么实际工作, 主要的绘制图形都是由 Shape 的各种子类的 draw 方法来完成的. 像这种没有实际工作的方法, 我们可以把它设计成一个 **抽象方法(abstract method)**, 包含抽象方法的类我们称为 **抽象类(abstract class)**.

```
abstract class Shape {
    abstract public void draw();
}
```

- 在 draw 方法前加上 abstract 关键字, 表示这是一个抽象方法. 同时抽象方法没有方法体(没有 { }, 不能执行具体代码).
- 对于包含抽象方法的类, 必须加上 abstract 关键字表示这是一个抽象类.

注意事项

1) 抽象类不能直接实例化.

```
Shape shape = new Shape();

// 编译出错
Error:(30, 23) java: Shape是抽象的; 无法实例化
```

2) 抽象方法不能是 private 的

```
abstract class Shape {
    abstract private void draw();
}

// 编译出错
Error:(4, 27) java: 非法的修饰符组合: abstract和private
```

3) 抽象类中可以包含其他的非抽象方法, 也可以包含字段. 这个非抽象方法和普通方法的规则都是一样的, 可以被重写, 也可以被子类直接调用

```
abstract class Shape {
    abstract public void draw();

    void func() {
        System.out.println("func");
    }
}

class Rect extends Shape {
    ...
}

public class Test {
    public static void main(String[] args) {
        Shape shape = new Rect();
        shape.func();
    }
}

// 执行结果
func
```

抽象类的作用

抽象类存在的最大意义就是为了被继承.

抽象类本身不能被实例化, 要想使用, 只能创建该抽象类的子类. 然后让子类重写抽象类中的抽象方法.

> 有些同学可能会说了, 普通的类也可以被继承呀, 普通的方法也可以被重写呀, 为啥非得用抽象类和抽象方法呢?

确实如此. 但是使用抽象类相当于多了一重编译器的校验.

使用抽象类的场景就如上面的代码, 实际工作不应该由父类完成, 而应由子类完成. 那么此时如果不小心误用成父类了, 使用普通类编译器是不会报错的. 但是父类是抽象类就会在实例化的时候提示错误, 让我们尽早发现问题.

> 很多语法存在的意义都是为了 "预防出错", 例如我们曾经用过的 final 也是类似. 创建的变量用户不去修改, 不就相当于常量嘛? 但是加上 final 能够在不小心误修改的时候, 让编译器及时提醒我们.
>
> 充分利用编译器的校验, 在实际开发中是非常有意义的.

接口

接口是抽象类的更进一步. 抽象类中还可以包含非抽象方法, 和字段. 而接口中包含的方法都是抽象方法, 字段只能包含静态常量.

语法规则

在刚才的打印图形的示例中, 我们的父类 Shape 并没有包含别的非抽象方法, 也可以设计成一个接口

```
interface IShape {
    void draw();
}

class Cycle implements IShape {
    @Override
    public void draw() {
        System.out.println("○");
    }
}

public class Test {
    public static void main(String[] args) {
        IShape shape = new Rect();
        shape.draw();
    }
}
```

- 使用 interface 定义一个接口
- 接口中的方法一定是抽象方法, 因此可以省略 abstract
- 接口中的方法一定是 public, 因此可以省略 public
- Cycle 使用 implements 继承接口. 此时表达的含义不再是 "扩展", 而是 "实现"
- 在调用的时候同样可以创建一个接口的引用, 对应到一个子类的实例.
- 接口不能单独被实例化.

> 扩展(extends) vs 实现(implements)
>
> 扩展指的是当前已经有一定的功能了, 进一步扩充功能.
>
> 实现指的是当前啥都没有, 需要从头构造出来.

接口中只能包含抽象方法. 对于字段来说, 接口中只能包含静态常量(final static).

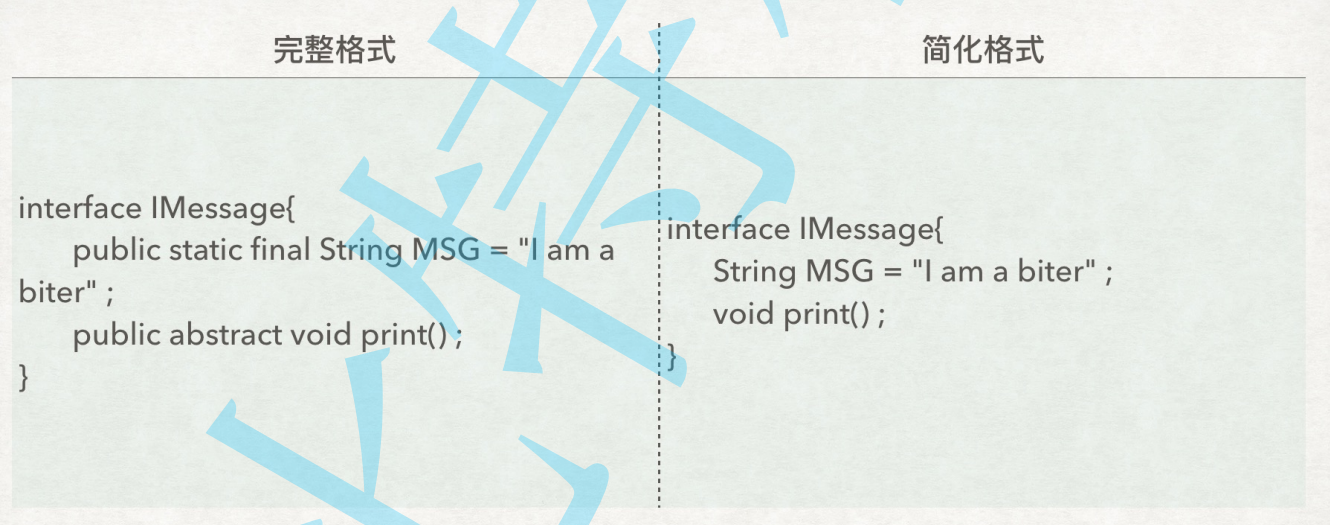
```
interface IShape {
    void draw();
    public static final int num = 10;
}
```

其中的 public, static, final 的关键字都可以省略. 省略后的 num 仍然表示 public 的静态常量.

> 提示:
>
> 1. 我们创建接口的时候, 接口的命名一般以大写字母 `I` 开头.
> 2. 接口的命名一般使用 "形容词" 词性的单词.
> 3. 阿里编码规范中约定, 接口中的方法和属性不要加任何修饰符号, 保持代码的简洁性.

一个错误的代码

```
interface IShape {
    abstract void draw() ; // 即便不写public, 也是public
}

class Rect implements IShape {
    void draw() {
        System.out.println("□") ; //权限更加严格了, 所以无法覆写。
    }
}
```



实现多个接口

有的时候我们需要让一个类同时继承自多个父类. 这件事情在有些编程语言通过 **多继承** 的方式来实现的.

然而 Java 中只支持单继承, 一个类只能 extends 一个父类. 但是可以同时实现多个接口, 也能达到多继承类似的效果.

现在我们通过类来表示一组动物.

```
class Animal {
    protected String name;

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

另外我们再提供一组接口, 分别表示 "会飞的", "会跑的", "会游泳的".

```
interface IFlying {
    void fly();
}

interface IRunning {
    void run();
}

interface ISwimming {
    void swim();
}
```

接下来我们创建几个具体的动物

猫, 是会跑的.

```
class Cat extends Animal implements IRunning {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println(this.name + "正在用四条腿跑");
    }
}
```

鱼, 是会游的.

```
class Fish extends Animal implements ISwimming {
    public Fish(String name) {
        super(name);
    }

    @Override
    public void swim() {
        System.out.println(this.name + "正在用尾巴游泳");
    }
}
```

青蛙, 既能跑, 又能游(两栖动物)

```
class Frog extends Animal implements IRunning, ISwimming {
    public Frog(String name) {
        super(name);
    }

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

    @Override
    public void swim() {
        System.out.println(this.name + "正在蹬腿游泳");
    }
}
```

> 提示, IDEA 中使用 ctrl + i 快速实现接口

还有一种神奇的动物, 水陆空三栖, 叫做 "鸭子"

```
class Duck extends Animal implements IRunning, ISwimming, IFlying {
    public Duck(String name) {
        super(name);
    }

    @Override
    public void fly() {
        System.out.println(this.name + "正在用翅膀飞");

    }

    @Override
    public void run() {
        System.out.println(this.name + "正在用两条腿跑");
    }

    @Override
    public void swim() {
        System.out.println(this.name + "正在漂在水上");
    }
}
```

上面的代码展示了 Java 面向对象编程中最常见的用法: 一个类继承一个父类, 同时实现多种接口.

继承表达的含义是 `is - a` 语义, 而接口表达的含义是 `具有 xxx 特性` .

> 猫是一种动物, 具有会跑的特性.
>
> 青蛙也是一种动物, 既能跑, 也能游泳
>
> 鸭子也是一种动物, 既能跑, 也能游, 还能飞

这样设计有什么好处呢? 时刻牢记多态的好处, 让程序猿**忘记类型**. 有了接口之后, 类的使用者就不必关注具体类型, 而只关注某个类是否具备某种能力.

例如, 现在实现一个方法, 叫 "散步"

```
public static void walk(IRunning running) {
    System.out.println("我带着伙伴去散步");
    running.run();
}
```

在这个 walk 方法内部, 我们并不关注到底是哪种动物, 只要参数是会跑的, 就行

```
Cat cat = new Cat("小猫");
walk(cat);

Frog frog = new Frog("小青蛙");
walk(frog);

// 执行结果
我带着伙伴去散步
小猫正在用四条腿跑
我带着伙伴去散步
小青蛙正在往前跳
```

甚至参数可以不是 "动物", 只要会跑!

```
class Robot implements IRunning {
    private String name;
    public Robot(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println(this.name + "正在用轮子跑");
    }
}

Robot robot = new Robot("机器人");
walk(robot);

// 执行结果
机器人正在用轮子跑
```

接口使用实例

刚才的例子比较抽象, 我们再来一个更能实际的例子.

给对象数组排序

给定一个学生类

```
class Student {
    private String name;
    private int score;

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

    @Override
    public String toString() {
        return "[" + this.name + ":" + this.score + "]";
    }
}
```

再给定一个学生对象数组, 对这个对象数组中的元素进行排序(按分数降序).

```
Student[] students = new Student[] {
    new Student("张三", 95),
    new Student("李四", 96),
    new Student("王五", 97),
    new Student("赵六", 92),
};
```

按照我们之前的理解, 数组我们有一个现成的 sort 方法, 能否直接使用这个方法呢?

```
Arrays.sort(students);
System.out.println(Arrays.toString(students));

// 运行出错, 抛出异常.
Exception in thread "main" java.lang.ClassCastException: Student cannot be cast to
java.lang.Comparable
```

仔细思考, 不难发现, 和普通的整数不一样, 两个整数是可以直接比较的, 大小关系明确. 而两个学生对象的大小关系怎么确定? 需要我们额外指定.

让我们的 Student 类实现 Comparable 接口, 并实现其中的 compareTo 方法

```
class Student implements Comparable {
    private String name;
    private int score;

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

    @Override
```

```
    public String toString() {
        return "[" + this.name + ":" + this.score + "]";
    }

    @Override
    public int compareTo(Object o) {
        Student s = (Student)o;
        if (this.score > s.score) {
            return -1;
        } else if (this.score < s.score) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

在 sort 方法中会自动调用 compareTo 方法. compareTo 的参数是 Object , 其实传入的就是 Student 类型的对象.

然后比较当前对象和参数对象的大小关系(按分数来算).

- 如果当前对象应排在参数对象之前, 返回小于 0 的数字;
- 如果当前对象应排在参数对象之后, 返回大于 0 的数字;
- 如果当前对象和参数对象不分先后, 返回 0;

再次执行程序, 结果就符合预期了.

```
// 执行结果
[[王五:97], [李四:96], [张三:95], [赵六:92]]
```

注意事项: 对于 sort 方法来说, 需要传入的数组的每个对象都是 "可比较" 的, 需要具备 compareTo 这样的能力. 通过重写 compareTo 方法的方式, 就可以定义比较规则.

为了进一步加深对接口的理解, 我们可以尝试自己实现一个 sort 方法来完成刚才的排序过程(使用冒泡排序)

```
public static void sort(Comparable[] array) {
    for (int bound = 0; bound < array.length; bound++) {
        for (int cur = array.length - 1; cur > bound; cur--) {
            if (array[cur - 1].compareTo(array[cur]) > 0) {
                // 说明顺序不符合要求, 交换两个变量的位置
                Comparable tmp = array[cur - 1];
                array[cur - 1] = array[cur];
                array[cur] = tmp;
            }
        }
    }
}
```

再次执行代码

```
sort(students);
System.out.println(Arrays.toString(students));

// 执行结果
[[王五:97], [李四:96], [张三:95], [赵六:92]]
```

接口间的继承

接口可以继承一个接口, 达到复用的效果. 使用 extends 关键字.

```
interface IRunning {
    void run();
}

interface ISwimming {
    void swim();
}

// 两栖的动物, 既能跑, 也能游
interface IAmphibious extends IRunning, ISwimming {

}

class Frog implements IAmphibious {
    ...
}
```

通过接口继承创建一个新的接口 IAmphibious 表示 "两栖的". 此时实现接口创建的 Frog 类, 就继续要实现 run 方法, 也需要实现 swim 方法.

> 接口间的继承相当于把多个接口合并在一起.

Clonable 接口和深拷贝

Java 中内置了一些很有用的接口, Clonable 就是其中之一.

Object 类中存在一个 clone 方法, 调用这个方法可以创建一个对象的 "深拷贝". 但是要想合法调用 clone 方法, 必须要先实现 Clonable 接口, 否则就会抛出 CloneNotSupportedException 异常.

```
class Animal implements Cloneable {
    private String name;

    @Override
    public Animal clone() {
        Animal o = null;
        try {
            o = (Animal)super.clone();
        } catch (CloneNotSupportedException e) {
```

```
            e.printStackTrace();
        }
        return o;
    }
}

public class Test {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Animal animal2 = animal.clone();
        System.out.println(animal == animal2);
    }
}

// 输出结果
// false
```

说明调用 clone 方法之后, 得到了一个全新的 Animal 对象. 这个过程称为 "深拷贝".

> 理解深拷贝
>
> 回忆我们曾经写的链表相关笔试题(例如根据指定值把链表分成两部分).
>
> 此时我们既可以直接在原链表上修改, 也可以不动原链表, 而是创建全新的链表. 这个创建全新链表的过程就是 "深拷贝".

总结

抽象类和接口都是 Java 中多态的常见使用方式. 都需要重点掌握. 同时又要认清两者的区别(重要!!! 常见面试题).

核心区别: 抽象类中可以包含普通方法和普通字段, 这样的普通方法和字段可以被子类直接使用(不必重写), 而接口中不能包含普通方法, 子类必须重写所有的抽象方法.

如之前写的 Animal 例子. 此处的 Animal 中包含一个 name 这样的属性, 这个属性在任何子类中都是存在的. 因此此处的 Animal 只能作为一个抽象类, 而不应该成为一个接口.

```
class Animal {
    protected String name;

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

> 再次提醒:
>
> 抽象类存在的意义是为了让编译器更好的校验, 像 Animal 这样的类我们并不会直接使用, 而是使用它的子类. 万一不小心创建了 Animal 的实例, 编译器会及时提醒我们.

No	区别	抽象类(abstract)	接口(interface)
1	结构组成	普通类+抽象方法	抽象方法+全局常量
2	权限	各种权限	public
3	子类使用	使用extends关键字继承抽象类	使用implements关键字实现接口
4	关系	一个抽象类可以实现若干接口	接口不能继承抽象类，但是接口可以使用extends关键字继承多个父接口
5	子类限制	一个子类只能继承一个抽象类	一个子类可以实现多个接口

作业

- 编写课堂上的代码.
- 写博客总结, 抽象类和接口的区别