1495 UNIVERSITY OF ABERDEEN

# JC2002 Java 程序设计

## 第 3 天：面向对象编程基础（人工智能、计算机科学与技术）

11 月 1 日星期三/11 月 2 日星期四

1495 UNIVERSITY OF ABERDEEN

# JC2002 Java 程序设计

## 第 3 天，第 1 课时：对象和类别

# 面向对象编程（OOP）

- 今天，我们将介绍面向对象的基础知识。编程（OOP）
  - 类和对象的基本概念
  - 实例变量、设置和获取方法
  - 范围和访问修改器
  - 枚举类型
  - 继承、组成和多态性

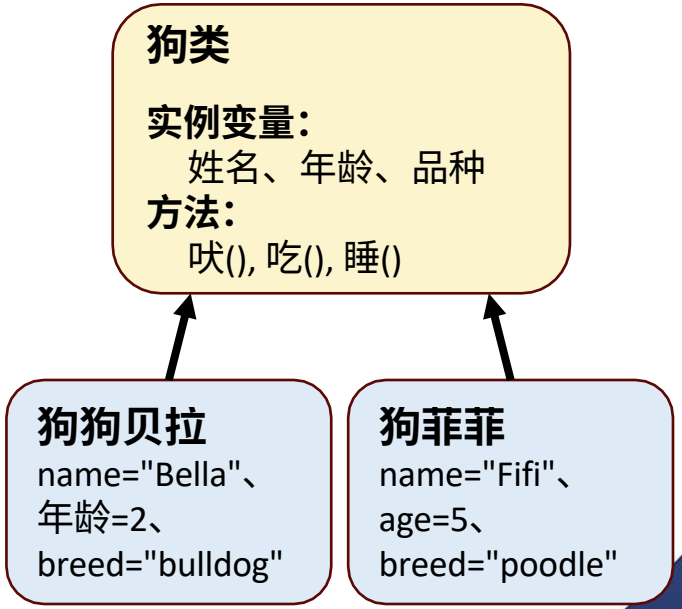
- 大部分材料基于《***Java：如何编程》的幻灯片***、第 7 章，可通过 MyAberdeen 获取

# 学习目标

- 今天的理论课程结束后，您应该能够
  - 解释类和对象的基本概念
  - 在 Java 程序中声明类和实例化对象
  - 为您的班级选择适当的访问修改器
  - 在 Java 程序中使用继承和组合

# 类和对象的概念

- ***类***是一种数据结构，代表一类具有某些共同特征的对象
  - 类可以包含定义其状态的*实例变量*以及实现其行为的*方法*
- ***对象***是一个类的实例
  - 例如，类 "人 "代表人，对象 "约翰 "是类 "人 "的一个实例，代表一个具体的



人

# Java 中的类和对象

- 在 Java 中，您可以根据需要声明新的类。
  被称为 *可扩展*语言
- 您创建的每个类都会成为一个新类型，可用于声明变量和创建对象
  - 按照惯例，类名、方法名和变量名都是
    标识符，并且全部使用驼峰字母命名方案
  - 此外，按照惯例，类名以大写字母开头，方法名和变量名以小写字母开头
  - 请注意，这些约定*并不是* Java 语法所强制要求的；然而，它是

强烈建议关注他们

# 实例变量

- 对象具有作为实例变量实现的属性
  并伴随其一生
- 该类的每个对象（实例）都有各自的
  类的实例变量
- 实例变量在类声明中声明，但在
  类方法声明的主体
- 一个类通常包含一个或多个方法，用于操作
  属于类的特定对象的实例变量

# 获取和设置方法

- 按照惯例，我们使用 *set* 和 *get* 方法来存储/获取实例对象中的变量值（即属性
  - 如果变量被定义为私有变量，则无法直接访问
  - 如果变量被定义为 public，则可以直接访问，但即使如此，最好还是使用 set 和 get 方法来修改变量
- *集合*方法通常被称为*突变方法*
- *获取*方法通常被称为*访问器方法*或*查询方法*

# 获取和设置示例

Account.java

```java
public class Account {
  private String name; // instance variable
  // 设置名称的方法
  public void setName(String name) {
    this.name = name;
  }
  // 检索名称的方法
  public String getName() {
     return name; // 返回名称值
  }
}
```

AccountTest.java

```java
import java.util.Scanner;
public class AccountTest {
  public static void main(String[] args) {
    // 为输入创建扫描仪对象
    Scanner input = new Scanner(System.in);
    // 创建账户对象 myAccount
    账户 myAccount = new Account();
    // 显示名称的初始值（空）
    System.out.printf ("初始名称为：%s%n%n",
      myAccount.getName()) ;
    // 提示并读取姓名 System.out.println("Please
    enter the name:"); String theName =
    input.nextLine(); myAccount.setName(theName);
    System.out.println(); // 输出空行
    // 显示存储在对象 myAccount 中的名称
  } System.out.printf("Name in myAccount is:%n%s%n",
}     myAccount.getName());
```

```
初始名称为：空

请输入姓名：简-格林

对象 myAccount 中的名称是：
简-格林
```

# 获取和设置示例

Account.java

```
1   public class Account {
2     private String name; // instance varia
3     // 设置名称的方法
4     public void setName(String name) {
5
6       this.name = name;
7
8
9     }
10    // 检索名称的方法
11  }
      public String getName() {
          return name; // 返回名称值
      }
```

设置器需要一个参数、返回无效

Getter 不带参数、return is String

```
13                字符串 theName = input.nextLine();
20      }
21
    }
```

```
的名称（空）
初始名称为：%s%n%n"、
);
d 名称
请输入名称："）；
名称）；
// 在对象 myAccount 中输出空
行 tored me in myAccount
is:%n%s%n"、
);
```

AccountTest.java

```java
import java.util.Scanner;
public class AccountTest {
  public static void main(String[] args) {
    // 为输入创建扫描仪对象

    Scanner input = new Scanner(System.in);

    // 创建账户对象 myAccount

    账户 myAccount = new Account();
```

```
初始名称为：空

请输入姓名：简-格林

对象 myAccount 中的名称是：
简-格林
```

# 访问修改器（公共和私人）

- 大多数实例变量声明前都有关键字
  私有，这是一个访问修饰符
- 使用访问修饰符 private 声明的变量或方法只能被声明它们的类中的方法访问
- 使用访问修饰符 private 声明实例变量被称为
  *信息隐藏*
  - 当程序创建（实例化）一个 *Account* 类对象时，*变量名*被封装（隐藏）在对象中，只能通过对象类的方法访问

# 方法的局部变量

- 方法的参数是方法的*局部变量*
  - 在特定方法正文中声明的局部变量可用于*只有*在这种情况下
  - 方法终止时，其局部变量的值将丢失
  - 本地变量不会自动初始化
- 如果方法包含同名的局部变量和实例变量，方法的主体将引用局部变量，而不是实例变量

- 局部变量是方法主体中实例变量的*影子*。
- 关键字 **this** 可用于明确引用阴影实例变量

# 使用关键字 *this*

```
1   public class Account {
2       private String name; // instance variable
3
4       // 在对象中设置名称的方法
5       public void setName(String name) {
6           this.name = name; // 存储
7       }
8
9       // 从
10      public String getName() {
11          return name; // 将名称值返回给调用者
12      }
13  }
```

我们本可以通过在行中选择不同的参数名来避免使用**这个**关键字。

# 关于*此*关键字的更多信息

- 每个对象都可以使用关键字 **this** 访问自身的引用 (有时也称为**本**参考文献)
- 当调用某个对象的实例方法时，该对象的方法的主体隐式地使用关键字 **this** 来引用对象的实例变量和其他方法
  - 因此，类的代码知道应该操作哪个对象
  - 每个方法在每个类中只有一个副本；每个对象的同类共享该方法的代码
- 另一方面，每个对象都有自己的类实例变量副本，非静态方

法会隐式地使用**该**副本来确定要操作的特定对象

# 实例化对象

- 使用关键字 **new** 创建类实例（对象
- *构造函数*类似于方法，但它是由 **new** 操作符隐式调用的，用于在创建对象时初始化对象的实例变量。
  - 如果一个类没有定义构造函数，编译器会提供一个不带参数的默认构造函数，类的实例变量会被初始化为默认值
  - 每个实例变量都有一个默认初始值（提供的值为如果您没有指定初始值
  - 字符串类型实例变量的默认值为空

# 构造函数示例

- 在本例中，使用构造函数设置了实例变量名、因此我们不需要在创建对象后调用 setName

Account.java

```
1   public class Account {
2     private String name; // instance variable
3
4     // 构造函数初始化名称
5     public Account(String name) {
6       this.name = name;
7     }
8
9     // 设置名称的方法
10    public void setName(String name) {
11      this.name = name;
12    }
13    // 检索名称的方法
14    public String getName() {
15      return name; // 返回名称值
      }
    }
```

AccountTest.java

```
...
9
10
11
...
    System.out.println ("请输入姓名: ") ;
    String theName = input.nextLine();
    Account myAccount = new Account(theName)
    ;
```

构造函数是一种与类同名的方法。当使用关键字 new 来实例化一个对象时，它就会被调用。

# 构造函数重载示例

- *重载构造函数*允许以不同方式初始化对象
  - 只有构造函数的参数不同

Account.java

```
1  public class Account {
2    private String name; // instance variable
3
4    // 使用全名作为输入的构造函数
5    public Account(String name) {
6      this.name = name;
7    }
8
9    // 带有姓和名的构造函数
10   // 作为输入
11   public Account(String first, String last) {
12     this.name = first + " " + name;
     }
   }
```

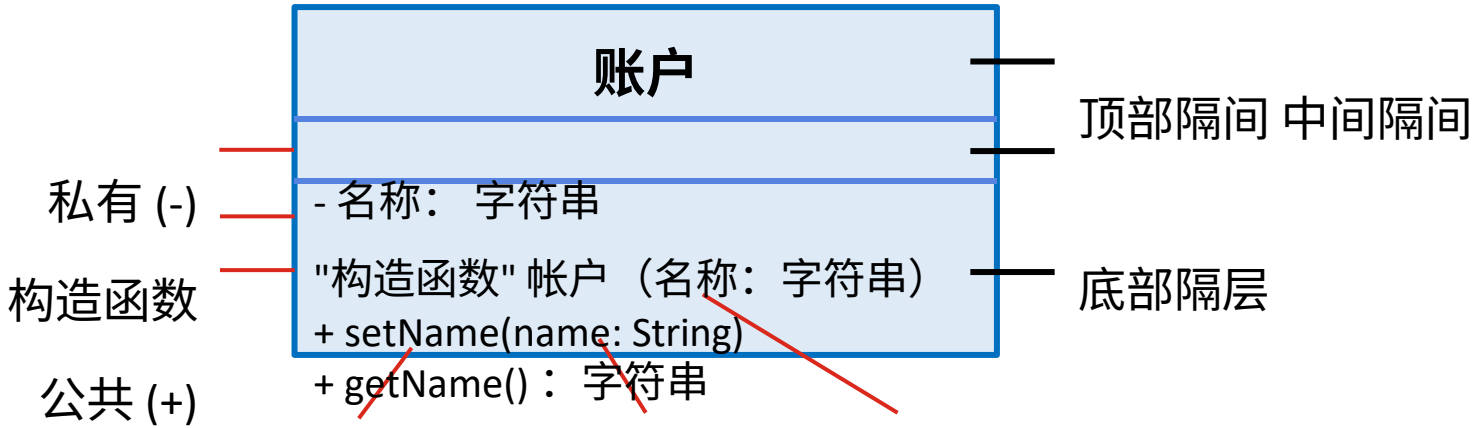
AccountTest.java

```
...
9   账户 lisasAccount = 新账户 ("Lisa Brown") ; 账户
10  bobsAccount = 新账户 ("Bob"、"Blue") ;
...
```

创建 lisasAccount 时会调用带一个参数的构造函数，创建 bobsAccount 时会调用带两个参数的构造函数。

# UML 类图

- UML 类图通常用于说明类



方法　　名称返回类型　　参数类型

# 有问题或意见?

1495 UNIVERSITY OF ABERDEEN

# JC2002 Java 程序设计

## 第 3 天，第 2 课时：枚举类型、静态和最终类型

# 枚举类型和关键字 static 和 final

- 什么是枚举类型?
    - 枚举声明
- 关键字 静态
    - 静态类成员
    - 静态输入
- 最终关键词

- 最小特权原则
- 最终实例变量

- 大部分材料基于 ***Java*** 的幻灯片：***如何编程***》第 8 章，可通过 MyAberdeen 获取

# 什么是枚举类型?

- 与类一样，所有枚举类型都是引用类型
- 基本枚举类型定义了一组以唯一标识符表示的常量
- 对于每个枚举，编译器都会生成静态方法 **values()** 返回枚举常量数组
- 枚举常量可以在任何可以使用常量的地方使用，例如开关语句的大小写标签，以及控制增强语句

# 枚举声明

- 枚举类型是通过*枚举声明来声明*的，*枚举声明*是一个*逗号分隔*的枚举*常量*列表
- 声明可选择包含传统类的其他组成部分，例如构造函数、字段和方法
  - 枚举构造函数可以指定任意数量的参数，并且可以重载
- 每个枚举声明都声明了一个枚举类，并有以下限制：
  - 枚举常量是隐式最终和静态的
  - 使用操作符 new 创建枚举类型对象的尝试

会导致编译错误

# 枚举声明示例

Book.java

```
1   公共枚举图书 {
2     // 声明枚举类型的常量
3     JHTP ("Java 如何编程", "2018") 、
4     CHTP ("C 如何编程", "2016") 、
5     IW3HTP("Internet & World Wide Web How to Program", "2012")、
6     CPPHTP ("C++ 如何编程", "2017") 、
7     VBHTP ("Visual Basic 如何编程", "2014") 、
8     CSHARPHTP("Visual C# 如何编程", "2017"); 9
10    // 实例字段
11    private final String title;
12    private final String copyrightYear;
13
15      B          tring title, String copyrightYear)
        o          {
        o
        k  16        this.title = title;
        (  17        this.copyrightYear =
        S            copyrightYear;
```



// 枚举构造函数

```
18      }
```

```
18      // 字段标题访问器
19      public String getTitle() {
20        return title;
21      }
22      // 版权年字段的访问器
23      公共字符串 getCopyrightYear() {
24        return copyrightYear;
25      }
26  }
```

# 枚举方法

- 增强的 for 语句可与 EnumSet 一起使用，只需数组一样
- 类 **EnumSet** 的方法 **range**()（在包 java.util) 可用于访问枚举常量的范围
  - 方法 range 需要两个参数：第一个和最后一个枚举范围内的常数
  - 返回一个 EnumSet，其中包含这两个常量之间的所有常量（包括这两个常量）。
- 类 EnumSet 还提供了其他几个静态方法

# 枚举使用示例

EnumTest.java

```
1   import java.util.EnumSet;
2
3   公共类 EnumTest {
4     public static void main(String[] args) {
5       System.out.println("All books:");
6       // 打印枚举图书中的所有图书
7       for (Book book : Book.values()) {
8         System.out.printf("%-10s%-45s%s%n", book,
9           book.getTitle(), book.getCopyrightYear());
10      }
11      System.out.printf ("%n显示一系列枚举常量：%n") ;
12      // 印刷前四册
13      for (Book book : EnumSet.range(Book.JHTP, Book.CPPHTP)){
14        System.out.printf("%-10s%-45s%s%n", book,
15          book.getTitle(), book.getCopyrightYear());
```

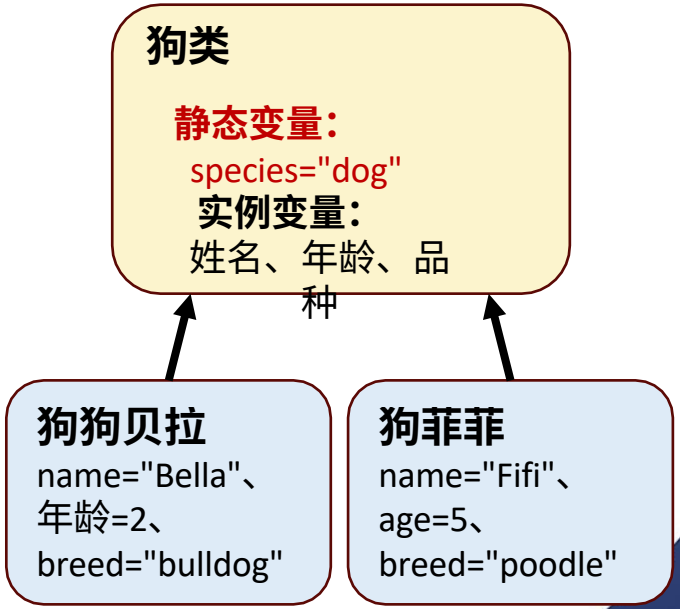
```
All books:
JHTP      Java How to Program                           2018
CHTP      C How to Program                              2016
IW3HTP    Internet & World Wide Web How to Program      2012
CPPHTP    C++ How to Program                            2017
VBHTP     Visual Basic How to Program                   2014
CSHARPHTP Visual C# How to Program                      2017

Display a range of enum constants:
JHTP      Java How to Program                           2018
CHTP      C How to Program                              2016
IW3HTP    Internet & World Wide Web How to Program      2012
CPPHTP    C++ How to Program                            2017
```

```
16          }
17        }
18      }
```

# 静态类成员

- 静态字段（称为*类变量*）用于一个类的所有对象只*共享*一个特定变量副本的情况
- 静态变量具有*类作用域*，它代表*整个类的*信息：该类的所有对象共享*同*一段数据，它也可以在该类的所有方法中使用
- 静态变量的声明以关键字



# static 开始

# 静态类成员的特点

- 静态类成员在类加载到
  执行时的内存
  - 声明为私有静态的类成员只能通过类的方法被客户端代码访问
  - 类的公共静态成员可以通过对该类任何对象的引用来访问，也可以用类名和点（.）来限定成员名，如 Math.random()
- 当类中不存在对象时：
  - 要访问公共静态成员，请在类名前加上一个点 （.）
    静态成员，如 Math.PI
  - 要访问私有静态成员，请提供一个公共静态方法，并在调用该方法时在其名称前加上类名和一个点

# 静态方法的特点

- 由于静态方法即使在没有实例化类对象的情况下也能被调用，因此静态方法*不能*访问类的实例变量和实例方法
  - this 引用*不能*在静态方法中使用：this 引用必须指向一个特定的类对象，但当调用静态方法时，内存中可能没有该类的任何对象。
- 如果静态变量未被初始化，编译器会给它分配一个默认值（例如，int 类型的默认值为 0）

# 静态类成员示例 (1)

雇员.java

```
1   public class Employee {
2     private static int count =
3     0; private String firstName;
4
5     private String lastName;
6
7     // 构造函数
8     public Employee(String firstName,
9         String lastName) {
10      this.firstName = firstName;
11
12      this.lastName = lastName;
13
14      ++count; // 增加静态计数
15
16      System.out.printf("Name %s %s; count = %d%n",
17          名，姓，计数）；
18    }
19    public String getFirstName() {
20      return firstName;
21    }
22    public String getLastName() {
23      return lastName;
```

EmployeeTest.java

```
1   公共类 EmployeeTest {
2     public static void main(String[] args) {
3     System.out.printf("employees before: %d\n",
4         Employee.getCount());
5     // 创建两个雇员；计数应为 2
6     员工 e1 = 新员工 ("Susan", "Baker") ;
7     员工 e2 = 新员工 ("Bob", "Blue") ;
8
9     // 显示计数现在是 2
10    System.out.printf("ënEmployees after:\n");
11    System.out.printf("via e1.getCount(): %d\n",
12        e1.getCount());
13    System.out.printf("via e2.getCount(): %d\n",
15        e2.getCount());
16    System.out.printf("via Employee.getCount(): %d\n",
17        Employee.getCount());
18    // 获取员工姓名
19    System.out.printf("\nEmployee 1: %s %s%n",
20        e1.getFirstName(), e1.getLastName());
21    System.out.printf("\nEmployee 2: %s %s%n",
22        e2.getFirstName(), e2.getLastName());
23    }
24  }
```

```
  }
  public static int getCount() {
    return count;
  }
}
```

# 静态类成员示例 (2)

雇员.java

```
1   公共类 雇员 {
2     私有静态 int count = 0;
3     private String firstName;
4     private String lastName;
5     // 构造函数
6     public Employee(String firstName、
13    }
14    公共字符串 getFirstName() {
15      return firstName;
16    }
17    公共字符串 getLastName() {
18      return lastName;
```

计数器变量 count 是 Employee 类所有实例共享的静态变量。

%d%n"、

EmployeeTest.java

```
1
2   公共类 EmployeeTest {
3     public static void main(String[] args) {
4     System.out.printf("Employees before: %d\n"、
5         Employee.getCount());
6     // 雇员 e1 = new Employee("Susan", "Baker");
7
8     雇员 e2 = new Employee("Bob", "Blue");
9     // 显示计数为 2 System.out.printf("\nEmployees
10
11    after:\n");System.out.printf("via e1.getCount():
12
13    %d\n"、
15        e1.getCount());
16    System.out.printf("via e2.getCount(): %d\n",
          e2.getCount());
      System.out.printf("via Employee.getCount(): %d\n"、
17        Employee.getCount());
18    // 获取员工姓名
19    System.out.printf("\nEmployee 1: %s %s%n"
```

```
    }
    公共静态 int getCount()        {

      return count;
    }
  }
```

```
    `
        e1.getFirstName(), e1.getLastName());
    System.out.printf("\nEmployee 2: %s %s%n"
    `
        e2.getFirstName(), e2.getLastName());
    }
}
```

# 静态类成员示例 (3)

雇员.java

```
1   公共类 雇员 {
2     私有静态 int count = 0;
3     private String firstName;
4     private String lastName;
5     // 构造函数
6     public Employee(String firstName、
7         字符串 lastName) {
8       this firstName = firstName;
9       this lastName =
10      ++count; // 增加静态计数
11      System.out.printf("Name %s %s; count = %d%n"、
12          名，姓，计数）；
13    }
```

EmployeeTest.java

```
1   公共类 EmployeeTest {
2     public static void main(String[] args) {
3   System.out.printf("employees before: %d\n"、
4
5         Employee.getCount());
6     // 创建两个雇员；计数应为 2 雇员 e1 = new
7
8     Employee("Susan", "Baker"); 雇员 e2 = new
9     // 显示计数为 2 System.out.printf("\nEmployees
10    Employee("Bob", "Blue");
11    after:\n");System.out.printf("via e1.getCount():
12
13    %d\n"、
15
16        e1.getCount());
17    System.out.printf("via e2.getCount(): %d\n",
          e2.getCount());
      System.out.printf("via Employee.getCount(): %d\n",
          Employee.getCount());
```

```
17      公共字符串 getLastName() {
18        return lastName;

19      }
20      公共静态 int getCount()        {

21        return count;
22      }
23    }
```

```
18    // 获取员工姓名
19    System.out.printf("\nEmployee 1: %s %s%n"
      、
20        e1.getFirstName(), e1.getLastName());
21    System.out.printf("\nEmployee 2: %s %s%n"
      、
22        e2.getFirstName(), e2.getLastName());
23    }
24  }
```

# 静态类成员示例 (4)

雇员.java

```
1   公共类 雇员 {
2     私有静态 int count = 0;
3     private String firstName;
4     private String lastName;
5     // 构造函数
6     public Employee(String firstName、
7         字符串 lastName) {
8       this firstName = firstName;
9       this lastName =
10      ++count  // 增加静态计数
11      System.out.printf("Name %s %s; count = %d%n"、
12          名，姓，计数）；
13    }
14
15      return firstName;
16    }
```

EmployeeTest.java

```
1
2   公共类 EmployeeTest {
3     public static void main(String[] args) {
4     System.out.printf("Employees before: %d\n",
5         Employee.getCount());
6
7     // 雇员 e1 = new Employee("Susan", "Baker");
8
9     雇员 e2 = new Employee("Bob", "Blue");
10    // 显示计数为 2 System.out.printf("\nEmployees
11    after:\n");System.out.printf("via e1.getCount()：
12
13    %d\n"、
15        e1.getCount());
16
17    System.out.printf("via e2.getCount()：%d\n",
18
19        e2.getCount());
20
21    System.out.printf("via Employee.getCount()：%d\n",
22
23        Employee.getCount());
24    // 获取员工姓名
}     System.out.printf("\nEmployee 1: %s %s%n"、
          e1.getFirstName(), e1.getLastName());
```

```
姓名:  苏珊-贝克Susan Baker;
count = 1 Name: Bob Blue;
count = 2
```

```
19      }
20      public static int getCount() {
21        return count;
22      }
23    }
```

# 静态类成员示例 (5)

雇员.java

```
1   公共类 雇员 {
2     私有静态 int count = 0;
3     private String firstName;
4     private String lastName;
5     // 构造函数
6     public Employee(String firstName、
7         字符串 lastName) {
8       this.firstName = firstName;
9       this.lastName = lastName;
10      ++count; // 增加静态计数
11      System.out.printf("Name %s %s; count = %d%n"、
12          名，姓，计数）；
13    }
```

```
之前的员工人数：0
姓名： 苏珊-贝克Susan Baker;
```

EmployeeTest.java

```
1   公共类 EmployeeTest {
2     public static void main(String[] args) {
3     System.out.printf("employees before: %d\n"、
4         Employee.getCount());
5     // 创建两个雇员；计数应为 2
6     员工 e1 = 新员工 ("Susan"、"Baker")；
7     员工 e2 = 新员工 ("Bob"、"Blue")；
8
9     // 显示计数现在是 2
10    System.out.printf("ënEmployees after:\n");
11    System.out.printf("via e1.getCount()：%d\n"、d\n",
12        e1.getCount());
13    System.out.printf("via e2.getCount()：%d\n"、
20    // 获取员工姓名
21
22    System.out.printf("\nEmployee 1: %s %s%n"、
23        e1.getFirstName(), e1.getLastName());
24  } System.out.printf("\nEmployee 2: %s %s%n"、
        e2.getFirstName(), e2.getLastName());
    }
```

```
count = 2
```

之后的员工

通过 e1.getCount()：2

通过 e2.getCount()：2

通过 Employee.getCount()：2

# 静态类成员示例 (6)

雇员.java

```
1   公共类 雇员 {
2     私有静态 int count = 0;
3     private String firstName;
4     private String lastName;
5     // 构造函数
6     public Employee(String firstName、
7                     lastName
8           this firstName =
9           this lastName =
10          ++count  // 增加静态计数
```

```
之前的员工人数: 0
姓名:  苏珊-贝克Susan Baker;
count = 1 Name: Bob Blue;
count = 2
```

EmployeeTest.java

```
1
2   公共类 EmployeeTest {
3     public static void main(String[] args) {
4     System.out.printf("Employees before: %d\n",
5         Employee.getCount());
6
7     // 雇员 e1 = new Employee("Susan", "Baker");
8
9     雇员 e2 = new Employee("Bob", "Blue");
10    // 显示计数为 2 System.out.printf("\nEmployees
11    after:\n");System.out.printf("via e1.getCount():
12
13    %d\n"、
15        e1.getCount());
16
17    System.out.printf("via e2.getCount(): %d\n",
18
19        e2.getCount());
20
21    System.out.printf("via Employee.getCount(): %d\n",
22
23        Employee.getCount());
24    // 获取员工姓名
}
      System.out.printf("\nEmployee 1: %s %s%n"、
          e1.getFirstName(), e1.getLastName());
```

之后的员工

通过 e1.getCount(): 2

通过 e2.getCount(): 2

通过 Employee.getCount(): 2

员工 1: 苏珊-贝克 员工 2: 鲍勃-布鲁

# 静态输入

- *静态导入*声明使您可以导入类或接口的静态成员，这样您就可以通过类中*未限定的名称来*访问它们。
- 两种形式的静态导入：
  - 一个导入特定静态成员（称为*单一静态导入）*
  - 一种是导入类的所有静态成员（称为*静态按需导入*)

# 静态导入语法

- 以下语法导入了一个特定的静态成员：

  **导入静态 *packageName.ClassName.staticMemberName;***

- 以下语法导入了类的*所有*静态成员：

  **import static *packageName.ClassName.*;***

  - 其中，*packageName* 是类的包，*ClassName* 是类的名称，*staticMemberName* 是静态字段或方法的名称。
  - 通配符 * 表示应导入指定类的*所有*静态成员

- 请注意，静态导入声明只导入静态类成员：应使用常规导入声明来指定程序中使用的类

# 静态导入示例

```
1   // 静态导入数学类方法。
2   import static java.lang.Math.*;
3
4   公共类 静态导入测试 {
5       public static void main(String[] args) {
6           System.out.printf("sqrt(900.0) = %.1f\n", sqrt(900.0));
7           System.out.printf("ceil(-9.8) = %.1f\n", ceil(-9.8));
8           System.out.printf("E = %f\n", E);
9           System.out.printf("PI = %f\n", PI);
10      }
11  }
```

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

# 最终实例变量

- 关键字 final 指定变量不可修改（即它是一个常量），任何修改它的尝试都会导致错误
  - 最终变量在初始化后不能通过赋值修改
  - 可以在声明终变量时对其进行初始化，例如，声明一个最终（常量）实例变量 INCREMENT，类型为 int，用途是：

```
私用最终 INCREMENT.D.C;
```

  - 如果在类的不同构造函数中用不同的值初始化了最终变量，那么类的不同对象可能会有不同的最终变量值

# 为什么要使用最终变量?

- *最小特权原则*是良好软件的基础工程学
  - 只应授予代码完成其指定任务所需的权限和访问量，不得超出这一范围。
  - 这一原则可防止代码意外（或恶意）修改变量值和调用不应访问的方法，从而使程序更加稳健

# 有问题或意见?

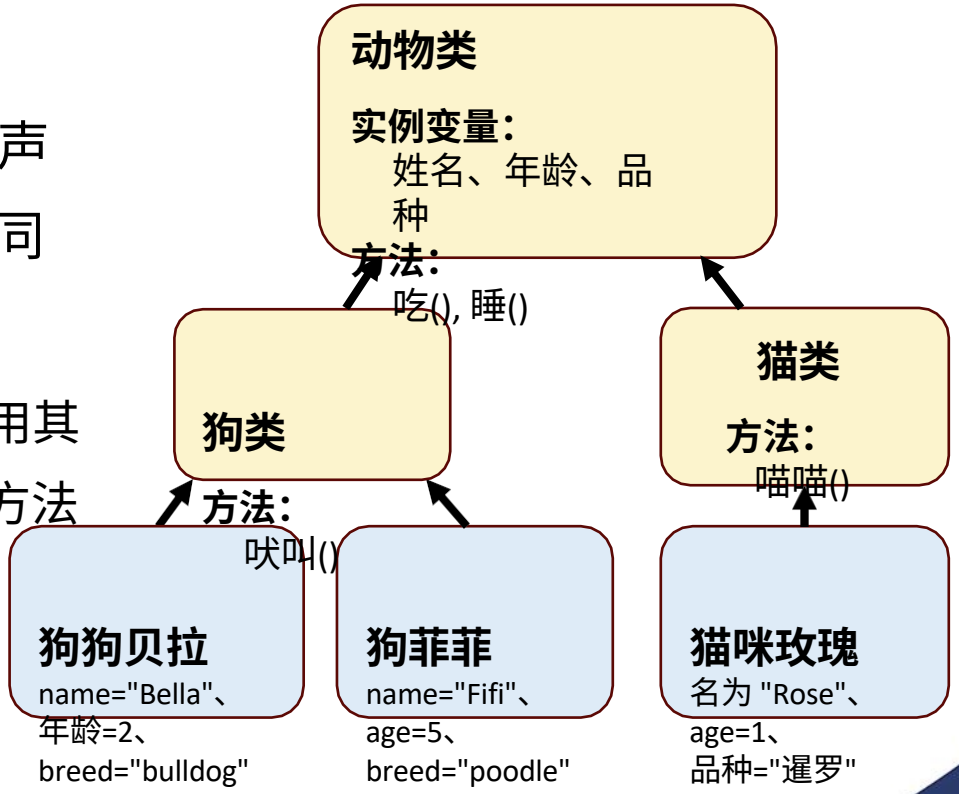
1495 UNIVERSITY OF ABERDEEN

# JC2002 Java 程序设计

## 第 3 天，第 3 课：类继承和访问修饰符

# 类继承

- ***通过类继承***，我们可以声明从更高级类继承共同结构的类
  - 继承类的对象可以使用其继承类的成员变量和方法



# 类继承的好处

- DRY：不要重复
  - 通过继承，我们可以将通用结构和信息传递给类似物
- 类继承允许 "重复使用 "对象的某些部分
  - 我们可以找出共同属性，并将它们提升到更高的位置级对象，然后在下级对象中加以区分
  - 减少重复，简化代码维护和可重用性

# 超类和子类

- 继承自另一个类的类是***子类***（子类）
  - Java 并不直接支持多重继承：您只能继承一类
- 被继承的类是***超类***（父类）
  - 扩展共同超类的所有类的对象都可被视为该超类的对象/成员
- 例如，要从一个类继承，请使用 **extends** 关键字：

```
class Dog extends Animal { ... }
```

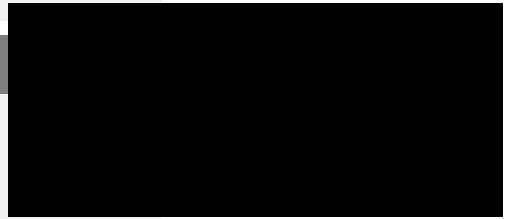
# 继承示例

车辆.java

```
1  类 车辆 {
2    protected String brand = "Ford";
3    public void honk() {
4      System.out.println("Tuut tuut!");
5    }
6  }
```

Car

```
1  类 Car 扩展车辆 {
2    private String modelName = "Mustang";
3    public static void main(String[] args) {
4      Car myCar = new Car();
5      myCar.honk();
```

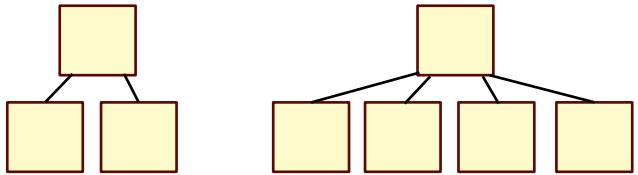


```
$ javac Car.java
$ java Car
Tuut tuut! 福
特野马
$
```

```
6         System.out.println(myCar.brand + " " + myCar.model);
7       }
8     }
```

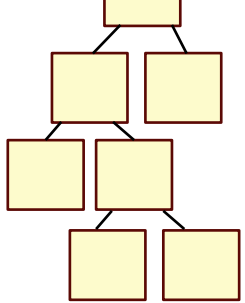
# 继承等级

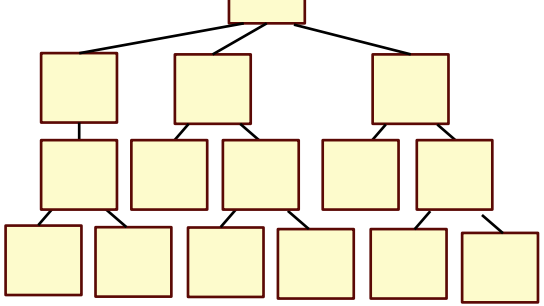
- 可通过继承构建不同的类层次结构
  - 深层次的层次结构非常复杂，而且往往会随着时间的推移而变得越来越宽，从而使其更难维护和使用
  - 为简单起见，更推荐使用浅层次结构



浅，窄

浅，宽





深、窄

深、宽

# 使用子类的构造函数

- 子类构造函数的首要任务是调用它的直接*显式*或*隐式地使用*超类的构造函数
    - 确保从超类继承的实例变量被正确初始化。
- 如果代码中没有明确调用超类的构造函数，Java 会隐式调用超类的默认或无参数构造函数

# 构造函数示例

TestCar.java

```
1  类 车辆 {
2    公共车辆( ) {
3      System.out.println("this is Vehicle constructor");
4    }
5  }
6  类 Car 扩展车辆 {
7    公共汽车( ) {
8      System.out.println("this is Car constructor"
9    }
10 }
11 公共类 TestCar {
12   public static void main(String[] arg) {
13                     new
14     }
15 }
```

```
$ javac TestCar.java
$ java TestCar
这是车辆构造函数  这是汽车构造函数
$
```

# 重新定义（覆盖）方法

- 即使超类方法适用于子类，但子类的子类通常需要一个定制版本的方法
- 子类可以*覆盖*（即重新定义）超类方法，并采用适当的实现方式
    - 在 Java 中，您可以使用可选的 **@Override** 注解来告诉编译器该方法应该覆盖另一个方法；这有助于在编译时发现错误
- 如果某个方法使用了关键字 **final**，就不能被重写；

试图覆盖最终方法时会出现编译错误

# 覆盖示例

```
1   类 车辆 {
2     void engine() {
3       System.out.println("this is vehicle engine");
4     }
5   }
6   类 Car 扩展车辆 {
7     void engine() {
8       System.out.println("this is car engine");
9     }
10  }
11  类 MotorBike 扩展车辆 {
12    void engine() {
13      System.out.println("this is motorbike engine")
        ;
14    }
15  }
```

```
16  公共类 TestEngines {
17    public static void main(String[] arg) {
18      MotorBike honda = new MotorBike ();
19      honda.engine();
20      Car ford = new Car ();
21      ford.engine () ;
22
23
```

```
$ javac TestEngines.java
$ java TestEngines
这是摩托车发动机 这是汽车发
动机
$
```

# 使用 @Override 重写示例

@Override 注解揭示了一个方法名称中的键入错误

```
3       System.out.println("this is vehicle engine")
4     } ;
5   }
6   类 Car 扩展车辆 {
7     @Override
8     void engne() {
9
10      System.out.println("this is car engine");
11    }
12  }
13
14  类 MotorBike 扩展车辆 {
15    @Override
16    void engine() {
17      System.out.println("this is motorbike engine")
        ;
      }
    }
```

```
18
19  公共类 TestEngines {
20    public static void main(String[] arg) {
21      MotorBike honda = new MotorBike ();
22      honda.engine();
23      Car ford = new Car ();
24      ford.engine () ;
25    }
```

```
$ javac TestEngines.java
错误：方法未覆盖或
实现超类中的方法
  @Override
  ^
1 个错误
$
```

# 用最终

```
1   类 车辆 {
2     final void engine() {
3       System.out.println("this is vehicle engine")
        ;
4     }
5   }
6   类 Car 扩展 Ve
7     @Override
8     void engine() {
9
10      System.out.println("this is car engine");
11    }
12  类 MotorBike 扩展车辆 {
13    @Override
14    void engine() {
15      System.out.println("this is motorbike engine");
16    }
```

方法定义为最终
不可覆盖

```
18  公共类 TestEngines {
19    public static void main(String[] arg) {
20      MotorBike honda = new MotorBike ();
21      honda.engine();
        Car ford = new Car ();
        ford.engine () ;
```

```
$ javac TestEngines.java
错误：汽车中的 engine() 不能覆盖
引擎() 在车辆
  void engine() {
       ^
  重载方法为最终方法
```

```
17 }
```

# 方法继承

- 在 Java 中，*每个类都*是 **Object 类**的子类，即使不是明确定义，以扩展 Object
  - 某些方法，如 toString，继承自对象，因此为每个类定义
    - 对象转换为字符串表示法
  - 默认的 toString 方法会返回一个字符串，其中包含对象的类名
    - 更合适的字符串表示方法是覆盖 toString

# 重载 toString() 方法的示例

```
1  类 车辆 {
2  }
3  类 Car 扩展车辆 {
4    @Override
5    public String toString() {
6
7      return "Hello, this is car!";
8    }
9  }
10 类 MotorBike 扩展车辆 {
   }
```

```
11 公共类 TestEngines {
     public static void main(String[] arg) {
12
13     MotorBike honda = new MotorBike ();
14     Car ford = new Car();
15     System.out.println(honda.toString());
16
       System.out.println(ford.toString());
17
     }
18 }
```

```
$ javac TestEngines.java
MotorBike@5acf9800 你好,
这是汽车!
$
```

默认 toString() 输出

覆盖 toString() 输出

# 访问修改器

- 无论程序在哪里，都可以访问类的*公共*成员。
  指向该类*或其子*类对象*的*引用
- 类的*私有*成员只能在类内部访问
- 使子类能够直接访问超类的实例变量、
  我们可以在超类中将这些成员声明为*受保护*成员
  - 受保护的访问是介于公共访问和私人访问之间的访问级别
  - 所有公共和受保护的超类成员都保留其原有的
    成为子类成员时的访问修饰符

# 受保护的访问修饰符

- 超类的受保护成员可以被*该超类*、*其子类*和*同一软件包中的其他类的*成员访问（受保护成员也有软件包访问权限）
  - 只需使用成员名称，子类方法就可以引用从超类继承的公共成员和受保护成员
- 对子类隐藏超类的私有成员
  - 只能通过从超类继承的公共或受保护方法访问它们
  - 在许多情况下，最好使用私有实例变量来鼓励正确的软件工程

# 受保护变量的缺点

- 对于受保护的实例变量，我们可能需要修改所有的
  如果超类的实现发生了变化
  - 这样的类被称为易碎或脆性类，因为它们的一个微小变化
    上类可以 "破坏 "子类的实现
  - 您应该能够更改超类的实现，同时仍能为子类提供相同的服务
- 一个类的受保护成员对与包含受保护成员的类在同一软件包中的所有类都是可见的--这并不总是可取的（最小权限原则）

# 访问修改器摘要

| 访问 | 默认 | 私人 | 受保护的 | 公 |
|---|---|---|---|---|
| 同一班级 | 是 | 是 | 是 | 是 |
| 同一软件包子类 | 是 | 没有 | 是 | 是 |
| 同一软件包非子类 | 是 | 没有 | 是 | 是 |
| 不同的软件包子类 | 没有 | 没有 | 是 | 是 |
| 不同的软件包非子类 | 没有 | 没有 | 没有 | 是 |

- 访问修饰符允许*封装*（数据与其他类隐藏）、

# OOP的基本概念之一

# 调用超类构造函数

- 每个子类构造函数都必须隐式或显式调用其的构造函数来初始化继承的实例变量。
  从超类
  - 调用超类构造函数的语法：**super(arguments)**
  - 必须是构造函数主体中的第一条语句
  - 这让您可以指定如何实例化对象
- 如果子类构造函数没有明确调用超类的构造函数，编译器会尝试插入对超类默认或无参数构造函数的调用

- 您也可以显式地使用 super() 调用超类的无参数或默认构造函数，但通常不会这样做

# 超类构造函数示例

```
1   类 车辆 {
2     私人字符串 type;
3     公共车辆( ) {
4       this.type = "undefined";
5     }
6     public Vehicle(String type) {
7       this.type = type;
8     }
9   }
10  类 Car 扩展车辆 {
11    私人发动机引擎;
12    公共汽车( ) {
13      super("car");
14    }
15  }
```

```
16  公共类 TestEngines {
      public static void main(String[] arg) {
17
18      Car ford = new Car();
19      System.out.print("Type: ");
20
21      ford.printType();
22    }
    }
```

```
$ javac TestEngines.java 类型
: 汽车
$
```

带参数调用超类的构造函数。请注意，变量**类型**是私有的，因此不能在超

类 **Vehicle** 之

外直接访问

。

# 参考超级方法

- 当子类方法覆盖了继承的超类方法时，只要在超类方法名称前加上关键字 **super** 和点（.）分隔符，就可以从子类访问超类版本的方法。

```
1  类 车辆 {
2    public void engine() {
3      System.out.println("this is vehicle engine")
       ;
4    }
5  }
6  类 Car 扩展车辆 {
7    public void engine() {
8      11 }
9
```

```
12 公共类 TestEngines {
13   public static void main(String[] arg) {
14     Car ford = new Car ();
15     ford.engine();
16   }
17
```

```
super.engine();
System.out.println("this is car engine");
```

```
}
```

```
$ java
TestEngines
this is
vehicle
engine this
is car engine
$
```

# 有问题或意见?

1495 UNIVERSITY OF ABERDEEN

# JC2002 Java 程序设计

## 第 3 天，第 4 节：组成和多态性

# 阶级关系

- 继承关系基本上是 ***is-a*** 关系
  - 汽车（子类）*是*交通工具（超类）
  - 狗（亚纲）*是*哺乳动物（超纲）
- 但是，有些类的关系是 ***has-a*** 关系
  - 汽车有发动机
  - 狗有尾巴

- 人有名字
- Has-a 关系应通过现有类的*组合而*不是继承来创建

# 组成

- 一个类可以引用其他类的对象作为成员
  - 这就是所谓的组成，有时也称为 has-a 关系
- 合成用于简化复杂性，让我们可以创建对象依赖性更少
  - 示例AlarmClock 对象需要知道当前时间和发出报警声的时间，因此在 AlarmClock 对象中包含两个对时间对象的引用是合理的

# 组成示例

Car.java

```
1  public class Car {
2    private Engine engine;
3    public Car() {
4
5      this.engine = new Engine();
6    }
7    public void startCar() {
8
9      engine.makeNoise();
     }
   }
```

Engine.java

```
1  公共类 引擎 { 公共 void
2    makeNoise() {
3      System.out.println("Wrroom!");
4    }
5  }
```

汽车 *有* 发动机

# 组成还是继承?

- 软件工程界对组合和继承的相对优点进行了大量讨论
  - 每种方法都有自己的用武之地，但继承往往被过度使用，在许多情况下，组合更为合适
- 组合和继承往往是最好的方法
  - 最好想一想，*is-a* 还是 *has-a* 关系更自然地代表你的情况

# 构成与继承

## 组成

- 组合和聚合形成 "有-无 "关系，总和大于部分
- 对象是独立的，因此开发成本较高：可重复使用的内置依赖性较少

## 继承

- 信息时的继承
  层级内自由授权
- 更易于开发，但依赖性更强：改变超类中的某些内容，很容易就会影响到所有子类，从而造成破坏

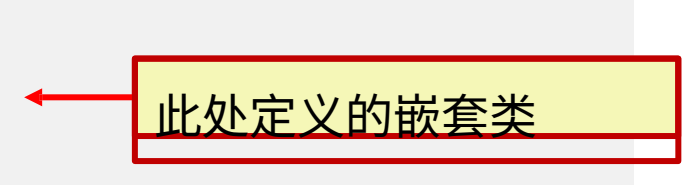
# 嵌套类

- Java 允许在类内声明类（嵌套类）
  - 要实例化嵌套（内部）类，首先需要实例化外层（外部）类
  - 非静态内部类可以访问外部类的其他成员，即使已声明为**私有**
- 嵌套类可视为一种 "组合"，因为外层类 "拥有 "内层类
  - 但是，组合的一些优点也会丧失，例如多态行为和可重用性：只有在绝对确定其他地方不需要嵌套类时，才使用嵌套类！

# 嵌套类示例

Car.java

```
公共类 汽车 { 私有类 发动机
  {
    public void makeNoise() {
      System.out.println("Wrroom!");
    }
  }
  private Engine engine;
  public Car() {
    this.engine = new Engine();
  }
  public void startCar() {
    engine.makeNoise();
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.startCar();
  }
}
```




```
JAVA 汽车
Wrroom!
$
```

# 匿名班级

- 在 Java 中，您可以声明匿名类
  - 匿名类与本地类类似，只是它们没有名称
  - 如果您只需要在一个地方使用本地类，请使用它们
- 匿名类在初始化语句中定义
  实例化时
  - 使用以下语法声明匿名类

```
超类 myClass = new SuperClass() {
    // 根据需要覆盖这里的方法
};
```

# 匿名类示例

Car.java

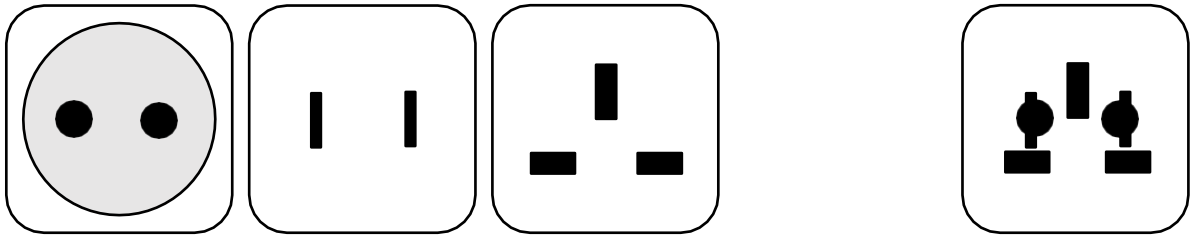
```
1   类 引擎 {
2     public void makeNoise() {
3       System.out.println("Put put put
        put!");
4     }
5   }
6   公共类 汽车 {
7     私人发动机引擎;
8     公共汽车( ) {
9       this.engine = new Engine() {
10        public void makeNoise() {
11          System.out.println("Wrrooom!");
12        }
13      };
14    }
```

```
15    public void startCar() {
16      engine.makeNoise();
17    }
18    public static void main(String[] args) {
19      Car car = new Car();
20      car.startCar();
21    }
22  }
```

的匿名子类
此处定义的发动机

# 多态性

- *多态性*使您可以定义一个接口，并使其多种执行方式
  - poly "意为 "许多"，"morphs "意为 "形式"：多态性意为 "许多形式"。

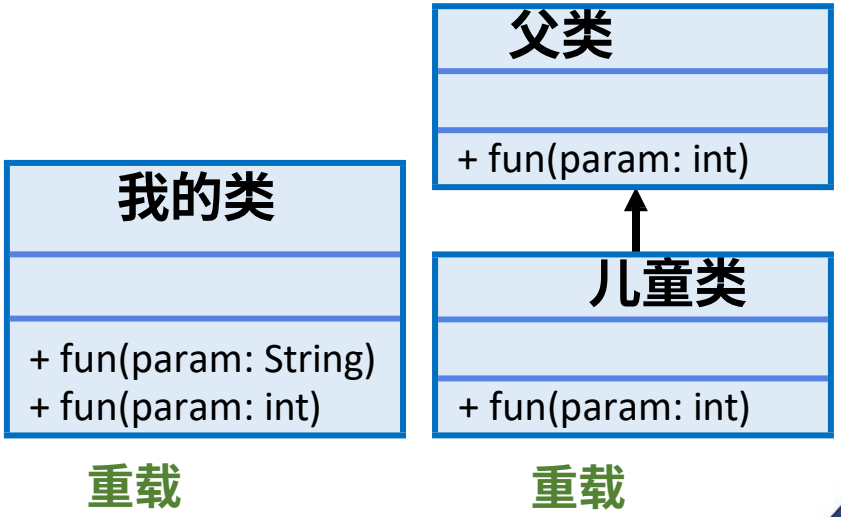


无多态性

有多态性

# 方法重载和覆盖

- 在 Java 中，多态性主要分为两种：
  - 编译时多态性（通过***方法重载***实现静态多态性）
  - 运行时多态性（通过***方法重载***实现动态方法调度）



# 重载示例

- 我们已经讨论过构造函数的重载，但其他方法也可以超载

```
1  类 Helper {
     static int Multiply(int a, int b) {return a * b;}
2
3    static double Multiply(double a, double b) {return a * b;}
4    public static void main(String[] args)
5    {
6
7      System.out.println(Helper.Multiply(2, 4));
8      System.out.println(Helper.Multiply(4.2, 3.8));
9    }
   }
```

```
$ java Helper
8
15.540000000000001
```

# 重载示例 (2)

- 不同版本的方法在参数类型或参数数

```
1  类 Helper {
2    static int Multiply(int a, int b) {return a * b;}
3    static int Multiply(int a, int b, int c) {return a * b * c;}
4    public static void main(String[] args)
5    {
6      System.out.println(Helper.Multiply(2, 4));
7
8      System.out.println(Helper.Multiply(2, 4, 8));
9    }
   }
```

```
$ java Helper
8
64
```

# 运行时覆盖示例

```
1  类 车辆 {
2    public void printType() {
3      System.out.println("undefined");
4    }
5  }
6  类 Car extends Vehicle { public
7    void printType() {
8      System.out.println("car");
9    }
10 }
11 类 MotorBike extends Vehicle { public
12   void printType() {
13     System.out.println("motorbike");
14   }
15 }
```

```
16 公共类 TestEngines {
17   public static void main(String[] arg) {
18
19     车辆 vehicle = new MotorBike();
20     System.out.print("Vehicle type 1: ");
21     vehicle.printType();
22     vehicle = new Car();
23     System.out.print("Vehicle type 2: ");
24     vehicle.printType();
25   }
}
```

```
车辆类型 1：摩托车 车辆类型 2
：汽车
$
```

# 覆盖数据成员

- 请注意，覆盖适用于方法，但不适用于数据成员！
  - 运行时多态性无法通过继承变量实现

```
1  类 车辆 {
2    int maxSpeed = 50;
3  }
4  类 Car 扩展车辆 {
5    int maxSpeed = 150;
6  }
7  公共类 TestEngines {
8    public static void main(String[] arg) {
9      Car ford = new Car();
10           11
```

```
$ javac TestEngines.java 最
高速度:  50
$
```

```
12  }
        System.out.printf("Max speed: %d%n", ford.maxSpeed);
    }
```

# 摘要

- Java 是一种*面向对象的语言*，因此，要理解了解 Java 的 OOP 概念至关重要
  - ***抽象：****类*、*对象*、*方法*和*变量*为复杂的底层数据和行为提供了简单的表现形式
  - ***封装：***可通过*访问修饰符*控制对类中私有成员的访问
  - ***继承：***可声明继承的*子类*共享高层*超类*的属性
  - ***多态性：***允许同名方法在不同上下文中工作

# 有问题或意见?