第5章 面向对象（下）

名家观点

时间不等人！历史不等人！时间属于奋进者！历史属于奋进者！为了实现中华民族伟大复兴的中国梦，我们必须同时间赛跑、同历史并进。

——习近平（习近平总书记在2020年春节团拜会上的讲话）

编程的艺术就是处理复杂性的艺术。

——艾兹格·迪科斯彻（Edsger Dijkstra，计算机科学家，1972年图灵奖得主）

代码胜于雄辩。（Talk is cheap. Show me the code）

——林纳斯·托瓦兹（Linus Torvalds，Linux之父）

本章学习目标

- 学会如何实现类的重用（继承和组合）。
- 掌握关键字this和super。
- 理解方法的覆盖。
- 掌握类的载入、静态语句块的调用、非静态语句块的调用、构造方法的调用。
- 理解对象的上溯造型和下溯造型。
- 掌握最终类、抽象类和接口的定义和应用。
- 掌握和灵活应用Java异常处理机制。
- 掌握常见英文单词。

课程思政——科学思维

张双南先生在《科学的目的、精神和方法是什么》中写到：科学的目的是发现各种规律；科学的精神包括质疑、独立、唯一；科学的方法包括逻辑化、定量化和实证化。

思维是人脑对客观事物间接和概括的反映，包括分析、综合、比较、概括、归纳、演绎和推理等能力。

在科学认识活动中，科学思维必须遵循以下三个基本原则。

（1）在逻辑上要求严密的逻辑性，达到归纳和演绎的统一。归纳方法是从个别或特殊的事物概括出共同本质或一般原理的逻辑思维方法，它是从个别到一般的推理。其目的

在于透过现象认识本质,通过特殊揭示一般。演绎思维是从一般到个别的推理,是根据一类事物都有的属性、关系、本质来推断该事物中个别事物也具有此属性、关系和本质的思维方法和推理形式。其基本形式是三段论,由大前提、小前提和结论三部分组成。只要前提是真的,在推理形式合乎逻辑的条件下,运用演绎推理得到的结论必然是真实的。

(2) 在方法上要求使用辩证地分析和综合这两种思维方法。分析与综合是抽象思维的基本方法。分析是把事物的整体或过程分解为各个要素,分别加以研究的思维方法和思维过程。只有先对各要素作出周密的分析,才可能从整体上进行正确的综合,从而真正地认识事物。综合就是把分解开来的各个要素结合起来,组成一个整体的思维方法和思维过程。只有对事物各种要素从内在联系上加以综合,才能正确地认识整个客观对象。

(3) 在体系上,要求实现逻辑与历史的一致,达到理论与实践的具体历史的统一。历史是指事物发展的历史和认识发展的历史,逻辑是指人的思维对客观事物发展规律的概括反映,即历史的事物在理性思维中的再现。历史是第一性的,是逻辑的客观基础;逻辑是第二性的,是对历史的抽象概括。

5-1 类的重用

5.1 类的重用

5.1.1 类的继承和组合

面向对象中的继承思想接近于人类的思维方式,可以大大提高代码的可重用性和健壮性,提高编程效率,降低软件维护的工作量。

类的重用有两种实现方式:继承和组合。

1. 继承

类的继承表达的是一般与特殊(IS A)的关系。那么,何时选择继承呢?一个很好的判断方法为:"B 是一个 A 吗?",如果是,则让 B 做 A 的子类。

Java 采用继承机制来组织、设计软件系统中的类和接口。在 Java 的单继承树状结构中,java.lang.Object 是 Java 中所有类的直接父类或间接父类。除根父类外,每个子类只能有一个直接父类,除最终类之外的类都可以有多个子类。

直观地讲,父类更加抽象,子类更加具体。子类是对父类的扩展,子类是一种特殊的父类。父类(ParentClass)又被称为基类(BaseClass)、超类(SuperClass)。子类(Subclass/ChildClass)又被称为衍生类(Derived Class)。

Java 采用"extends 父类名"的方式来实现单继承,采用"implements <接口列表>"的方式来实现多继承。

子类可以继承父类中未被 private 修饰的成员变量和方法,但不包括构造方法。在子类的类体中可以重写父类中的方法,增加新的属性和方法。所以,类除了自己的属性和方法外,还有从父类继承过来的属性和方法。

子类继承父类的语法格式如下:

```
<类修饰符> class <子类名> [extends <父类名>][implements <接口列表>]{
    …
}
```

说明：如果省略 extends 子句，编译器会自动加上 extends java.lang.Object。

2. 组合和聚合

组合和聚合表达的是整体与部分(Have a)的关系。组合强调整体和部分之间具有很强的“属于”关系，且它们的生存期是一致的。聚合则强调一种松散的家族和成员之间的关系。

何时选择组合或聚合呢？一个很好的判断方法为“A 有一个 B 吗?”，如果有，则让 B 做 A 的成员变量。

【示例程序 5-1】 类的组合示例(CombinationTest.java)。

功能描述：本程序描述了汽车类 Car，由 1 个引擎(Engine 类)、4 个车门(Door 类数组)组合而成。

```
01  class Engine {
02      public void start() {
03          System.out.println("------ 启动 ------");
04      }
05      public void stop() {
06          System.out.println("------ 停止 ------");
07      }
08  }
09  class Door {
10      public void open() {
11          System.out.println("------ 开门 ------");
12      }
13      public void close() {
14          System.out.println("------ 关门 ------");
15      }
16  }
17  class Car {
18      Engine engine = new Engine();
19      Door[] doors = new Door[4];
20  }
21  public class CombinationTest {
22      public static void main(String[] args) {
23          Car car = new Car();
24          car.engine.start();
25          for(int i = 0; i < 4; i++) {
26              car.doors[i] = new Door();
27          }
28          car.doors[0].close();
29      }
30  }
```

5.1.2 关键字 this 和 super

1. 关键字 this

this 指向当前对象,主要用来调用构造方法或访问成员变量(方法),语法格式如下。

(1) 在本类构造方法中调用其他构造方法,只能在第一行:

```
this([实参数列表]);
```

(2) 用于区分构造方法和对象方法中重名的对象变量和局部变量:

```
this.成员变量;
```

(3) 在构造方法和对象方法中调用其他对象方法(this 可能省略):

```
this.成员方法([实参数列表]);
```

说明:

- this()只能出现构造方法的第一行,在同一个构造方法中 super()和 this()不能同时出现。

2. 关键字 super

super 指向当前类的父类,主要用来调用父类的构造方法或访问父类的成员变量(方法),语法格式如下。

(1) 在本类构造方法中调用父类的构造方法,只能在第一行:

```
super([实参数列表]);
```

(2) 在构造方法或对象方法中访问父类中被覆盖的成员变量:

```
super.成员变量;
```

(3) 在构造方法或对象方法中访问父类中被覆盖的成员方法:

```
super.成员方法([实参数列表]);
```

说明:

(1) 调用父类的构造方法,只能出现在子类的构造方法中,且必须是第一行。

(2) super([实参数列表])中的参数,决定了调用父类哪个构造方法。如果子类构造方法中没有出现 super([实参数列表]),那么编译器将自动加入 super(),即调用父类的空构造方法。

5.1.3 方法的覆盖

覆盖是指子类的成员方法的名称和形式参数与父类的成员方法的名称和形式参数相同

的现象。

方法覆盖的规则如下。

（1）一般在子类继承父类时发生。

（2）3个相同：方法名、形式参数、返回类型相同。

（3）方法抛出的异常：子类方法抛出的异常应比父类方法抛出的异常更小或相等，即子类方法只能抛出父类方法异常或其异常的子类。

（4）方法的权限修饰符：子类方法的访问权限要大于或等于父类方法的访问权限。

（5）static()方法只能被static()方法覆盖，非static()方法只能被非static()方法覆盖，不能交叉覆盖。

（6）final()方法不能被覆盖。

【示例程序 5-2】 方法覆盖应用示例(Ostrich.java)。

功能描述： 本程序中Ostrich类的fly()方法覆盖Bird类中的fly()方法。

```
01  class Bird{
02      public void fly(){
03          System.out.println(" -- Bird: fly() -- ");
04      }
05  }
06  public class Ostrich extends Bird{
07      public void fly(){
08          System.out.println(" -- Ostrich: fly() -- ");
09      }
10      public static void main(String[] args) {
11          Ostrich o = new Ostrich();
12          o.fly();
13          Bird b = new Ostrich(); //上溯造型后
14          b.fly();
15      }
16  }
```

程序运行结果如下：

```
-- Ostrich: fly() --
-- Ostrich: fly() --
```

5.2　语句块和对象的造型

5-2　语句块和对象的造型

5.2.1　语句块

关键字static用来区分成员变量、成员方法、内部类、初始化块是属于类的还是属于对象的。有static修饰的成员属于类，否则属于对象。

【拓展知识 5-1】 语句块的隐含调用。Java语句一般写在方法中，供其他程序调用。语句块写在类体中方法外，由JVM自动调用。

(1) 静态语句块指在类体中、方法外定义的有 static 修饰的语句块。静态语句块所在类被 JVM 载入内存后自动执行一次,负责类的初始化。

JVM 要实例化一个类或访问某一个类中的方法时,需要先将该类载入内存。在将该类载入内存之前,JVM 自动先将这个类的父类载入内存。正如要上三楼,需要先上一楼、二楼,再上三楼。类载入内存后,JVM 自动执行该类的静态语句块。

(2) 非静态语句块指在类体中、方法外定义的未被 static 修饰的语句块。非静态语句在调用该类的构造方法前自动执行一次,用于初始化对象。调用一个类的构造方法,必须先调用其父类的构造方法。

JVM 使用"new 构造方法()"的方式实例化一个类时,必须先实例化其父类。因为编译器为每个类的构造方法都自动加上了一句"super();",实例化一个类之前,JVM 自动执行该类的非静态语句块。

【示例程序 5-3】 子类构造方法自动调用父类构造方法测试(ConStructor CallTest.java)。

功能描述:本程序中 Wolf 类继承了 Animal 类,Animal 类继承了 Creature 类。主类的 main()方法中只有一句 new Wolf(),请写出程序运行结果,与标准答案对比。

```
01  package chap05;
02  class Creature extends java.lang.Object {
03      public Creature() {
04          System.out.println("------ 生物 ------");
05      }
06  }
07  class Animal extends Creature {
08      public Animal() {
09          System.out.println("------ 动物 ------");
10      }
11  }
12  class Wolf extends Animal {
13      public Wolf() {
14          System.out.println("------- 狼 -------");
15      }
16  }
17  public class ConStructorCallTest {
18      public static void main(String[] args) {
19          new Wolf();
20      }
21  }
```

程序运行结果如下:

```
------ 生物 ------
------ 动物 ------
------- 狼 -------
```

【示例程序 5-4】 静态和非静态语句块、父类构造方法的自动隐含调用测试(D.java)。

功能描述：本程序中D类继承了B类，B类继承了A类。A类、B类、D类每个类中均包含了静态语句块、非静态语句块、构造方法，请写出调用D类的构造方法时自动执行的语句顺序（程序运行结果），与标准答案对比。

```
01  class A extends java.lang.Object{
02      {
03          System.out.println("------ 调用A的非static语句块 -----");
04      }
05      static{
06          System.out.println("------ 调用A的static语句块 -----");
07      }
08      A(){
09          System.out.println("------ 调用A() -----");
10      }
11  }
12  class B extends A {
13      static{
14          System.out.println("------ 调用B的static语句块 -----");
15      }
16      {
17          System.out.println("------ 调用B的非static语句块 -----");
18      }
19      B(){
20          System.out.println("------ 调用B() -----");
21      }
22  }
23  public class D extends B{
24      static{
25          System.out.println("------ 调用D的static语句块 -----");
26      }
27      {
28          System.out.println("------ 调用D的非static语句块 -----");
29      }
30      D(){
31          System.out.println("------ 调用D() -----");
32      }
33      public static void main(String args[]){
34          D d = new D();
35      }
36  }
```

程序运行结果如下：

```
------ 调用A的static语句块 -----
------ 调用B的static语句块 -----
------ 调用D的static语句块 -----
------ 调用A的非static语句块 -----
------ 调用A() -----
```

```
------ 调用 B 的非 static 语句块 -----
------ 调用 B() -----
------ 调用 D 的非 static 语句块 -----
------ 调用 D() -----
```

如果读者认为自己掌握了本节的内容,请通过 5.7 节自测题中的"二、程序运行题"验证自己掌握的程度。

5.2.2 对象的上溯造型和下溯造型

子类其实是一种特殊的父类。把一个子类对象直接赋值给一个父类对象变量的语法现象称为上溯造型。把父类对象强制转换赋值给子类对象变量的语法现象称为下溯造型。

某个类的对象可以完成以下操作之一。

(1) 上溯造型为任何一个父类类型或父接口(自动完成):即任何一个子类的变量(或对象)都可以被当成一个父类或父接口变量(或对象)来对待。例如:

```
Shape s = new Circle();
List i = new ArrayList();
```

(2) 通过下溯造型回到它自己所在的类(强制转换):一个对象被溯型为父类或父接口后,还可以再被下溯造型,回到它自己所在的类。只有曾经上溯造型过的对象才能进行下溯造型。对象不允许不经过上溯造型而直接下溯造型。否则运行时会抛出异常信息:java.lang.ClassCastException。

上溯造型的优点是:可以把不同类型的子类上溯为同一个父类类型,便于统一处理它们。缺点是:损失了子类新扩展增加的属性和方法(覆盖父类的方法不会损失),除非再进行下溯造型,否则这部分属性和方法不能被访问。

Java 的引用变量有两种类型,一种是编译时类型,另一种是运行时类型。编译时类型由声明该变量时使用的类型决定,运行时类型由实际赋给该变量的对象决定。如果编译时类型和运行时类型不一致,就会出现多态。

【示例程序 5-5】 引用类型上溯下溯造型示例(CastUpDownTest.java)。

功能描述:Fish 类继承了 Animal 类。Fish 对象可以自动上溯造型为 Animal 对象,然后下溯造型为 Fish 对象。本程序演示了在这个过程中方法的损失情况。

```
01  class Animals {
02      void breathe() {
03          System.out.println("…动物呼吸…");
04      }
05      //final 方法不能被子类覆盖
06      final static void live(Animals an){
07          an.breathe();
08      }
09  }
```

```
class Fish extends Animals {
    void swim() {                    //增加新的方法
        System.out.println("…鱼会游泳….");
    }
    @Override                        //覆盖父类的方法
    void breathe() {
        System.out.println("…鱼用鳃呼吸…");
    }
}
public class CastUpDownTest {
    public static void main(String args[]) {
        // Fish 对象上溯为 Animal 对象,将丢失增加的新方法 swim()
        Animals an = new Fish();
        //24 行找不到 swim(),编译出错
        // an.swim();
        // 上溯造型时,覆盖父类的方法不会损失
        an.breathe();
        // 下溯造型后,swim()又能被访问了
        Fish f = (Fish)an;
        f.swim();
    }
}
```

程序运行结果如下：

```
…鱼用鳃呼吸…
…鱼会游泳…
```

5-3　最终类、抽象类和接口

5.3　最终类、抽象类和接口

5.3.1　最终类

final 关键字可以作为变量修饰符、方法修饰符和类修饰符。

(1) final 用在变量前面，该变量成为常量，只能被赋值一次。

(2) final 用在方法前面，该方法成为最终方法，不能被子类的方法覆盖。

(3) final 用在类前面，该类成为最终类，只能实例化，不能被继承。

5.3.2　抽象类

abstract 关键字有以下两种用法。

(1) abstract 用作方法修饰符，表示该方法为抽象方法。抽象方法只有方法的定义(方法首部)，没有方法的实现(方法体)。例如：

```
public abstract double getArea();
```

(2) abstract 用作类修饰符,则该类为抽象类。包含抽象方法的类必须声明为抽象类。

注意:

(1) 抽象类不能被实例化为对象,只能被其他子类继承。

(2) 抽象方法不能为 static。

在下列情况下,一个类必须声明为抽象类。

(1) 当一个类包含抽象方法时。

(2) 当一个类继承了一个抽象类,并且没有实现父类的所有抽象方法,即只实现了部分抽象方法时。

(3) 当一个类实现了一个接口,并且没有实现接口中所有的抽象方法时。

【示例程序 5-6】 类继承抽象类应用示例(AbstractTest.java)。

功能描述: 本程序定义了抽象类 AShape,Square 类继承了抽象类 AShape,类 AbstractTest 进行了测试。

```
01  package chap05;
02  abstract class AShape{
03      //抽象方法只有方法的声明,没有方法的实现
04      public abstract double getArea();
05      public abstract double getPerimeter();
06  }
07  //类 Circle 继承了抽象类 AShape 就必须实现 AShape 中的所有抽象方法
08  //否则 Circle 只能声明为抽象类
09  class Circle extends AShape{
10      private double r;
11      // 无参构造方法
12      public Circle(){
13          super();
14      }
15      //有参数的构造方法
16      public Circle(double r){
17          super();
18          this.r = r;
19      }
20      // 实现继承抽象类 AShape 中的抽象方法
21      @Override
22      public double getArea(){
23          return r * r;
24      }
25      @Override
26      public double getPerimeter(){
27          return 2 * Math.PI * r;
28      }
29      // 通过覆盖 toString()方法,定制自己的输出信息
30      @Override
31      public String toString(){
32          //父类中的 toString()默认输出信息: chap05.Circle@7637f22
```

```
33          return "Circle[" + r + "]";
34      }
35      // private 变量的 Getters、Setters 方法
36      public double getR(){
37          return r;
38      }
39      public void setR(double r){
40          this.r = r;
41      }
42  }
43  public class AbstractTest{
44      public static void main(String[] args){
45          Circle s1 = new Circle();
46          s1.setR(9);
47          Circle s2 = new Circle(9);
48          System.out.println(s1.getArea()); //调用对象方法
49          System.out.println(s1.getPerimeter());
50          System.out.println(s2);
51          System.out.println(s2.toString());
52      }
53  }
```

5.3.3　接口

接口本质上是一个比抽象类更加抽象的类，接口中只能定义常量和抽象方法。抽象类只能对其子类的行为进行约束。但接口可以对实现本接口的所有类进行约束，范围更加广泛。

【拓展知识 5-2】 类、抽象类和接口。类是具体的，可以实例化。抽象类是包含抽象方法的类，只能被继承，不能被实例化。接口更加抽象，只能包含常量和抽象方法。接口是常量和抽象方法的集合。接口不能被实例化，只能被继承(extends)或实现(implements)。

1. 接口的定义

语法格式如下：

```
[接口修饰符] interface 接口名称 [extends 父接口列表]{
      //常量定义
      //抽象方法的定义
}
```

说明：

(1) 接口可以通过 extends 关键字继承其他接口，一个接口可以继承多个接口，这点与类的继承有所不同。

(2) 常量前如果没有 public、static、final 修饰符，编译器会自动加上。

(3) 接口中的抽象方法前会被编译器自动加上 abstract 关键字。

2. 类实现接口

定义好接口后，其他类就可以用 implements 关键字实现该接口，接受该接口的约束。

类实现接口的语法如下：

```
[类修饰符] class 类名 [extends 父类名] [implements <接口列表>]{
        …
        … //实现接口中的抽象方法
}
```

说明：

(1) 实现接口的类必须实现该接口中的所有抽象方法，只要有一个抽象方法没有实现，该类就只能被声明为抽象类。

(2) 在实现接口的时候，来自接口的方法必须声明成 public。

【示例程序 5-7】 类实现接口和继承抽象类示例(Person.java)。

功能描述： 本程序中 Person 类继承了司机抽象类(Driver)，实现了教师接口(ITeacher)和父亲接口(IFather)。一个人拥有多个角色，既是司机、又是教师和父亲。Person 类必须接受其父类和父接口的约束，即实现其父类和父接口中的抽象方法。

```
01  interface ITeacher {
02      public void teach();
03  }
04  interface IFather {
05      public void earnMoney();
06  }
07  abstract class Driver {
08      public abstract void drive();
09  }
10  public class Person extends Driver implements IFather,ITeacher{
11      @Override
12      public void teach() {
13          System.out.println("---------- 教书育人 ----------");
14      }
15      @Override
16      public void earnMoney() {
17          System.out.println("---------- 挣钱养家 ----------");
18      }
19      @Override
20      public void drive() {
21          System.out.println("---------- 开车上路 ----------");
22      }
23      public static void main(String[] args) {
24          Person p = new Person();
25          p.teach();
26          p.earnMoney();
27          p.drive();
28      }
29  }
```

5.4　异常处理机制

异常(Exception)是指程序运行过程中出现的可能会打断程序正常执行的事件或现象。例如,用户输入错误、除数为零、文件找不到、数组下标越界、内存不足等。为了加强程序的健壮性,编写程序时必须考虑到可能发生的异常事件并做出相应的处理。


5-4　异常处理机制

面向过程语言中用 if 语句实现程序的错误处理,业务逻辑代码和异常处理代码交叉在一起,程序阅读和维护都不太方便。Java 采用面向对象的异常处理机制,用 try…catch…finally 语句将程序中的业务逻辑代码和异常处理代码有效分离,便于程序的阅读、修改和维护。

【拓展知识 5-3】 异常处理机制。异常处理机制的完善性已经成为判断一门编程语言是否成熟的标准之一。它可以使程序中异常处理代码和正常业务代码分离,使得程序代码更加优雅,并提高了程序的健壮性。

5.4.1　方法调用堆栈

JVM 在执行字节码文件时,方法调用堆栈中详细记录了每一层方法调用的信息。程序运行过程中一旦发生异常,就会中止运行,并在控制台上输出方法调用堆栈中的信息。方法调用堆栈信息包括异常类型、每一层方法调用时异常发生所在类、方法、代码行等信息。通过查看方法调用堆栈信息,可以迅速定位异常发生位置,该信息是调试和修改程序的重要依据。

【示例程序 5-8】 程序发生异常时,控制台输出的方法调用堆栈信息(MethodCallTest.java)。

功能描述:本程序演示了当程序运行过程中发生异常时,自动在控制台上输出方法调用堆栈的信息。

```
01  public class MethodCallTest{
02      int[] arr = new int[3];
03      public void methodOne(){
04          methodTwo();
05          System.out.println("One");
06      }
07      public void methodTwo(){
08          methodThree();
09          System.out.println("Two");
10      }
11      public void methodThree(){
12          System.out.println(arr[3]);
13          System.out.println("Three");
14      }
15      public static void main(String[] args){
16          new MethodCallTest().methodOne();
17          System.out.println("main");
18      }
19  }
```

控制台输出信息如下：

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
for length 3
at chap05.MethodCallTest.methodThree(MethodCallTest.java: 12)
at chap05.MethodCallTest.methodTwo(MethodCallTest.java: 8)
at chap05.MethodCallTest.methodOne(MethodCallTest.java: 4)
at chap05.MethodCallTest.main(MethodCallTest.java: 17)
```

5.4.2 Exception 的概念、子类及其继承关系

在 Java 中，异常情况分为 Error 和 Exception 两大类。

(1) Error 类：指较少发生的内部系统错误，由 JVM 生成并抛出，包括动态链接失败、JVM 内部错误、资源耗尽等严重情况，程序员无能为力，只能让程序终止。

(2) Exception 类：由程序本身及环境所产生的异常，有补救或控制的可能，程序员可预先防范，增加程序的健壮性。

Java 中的 Exception 可以分为以下两大类。

(1) 编译时异常：也称为检查性异常，Java 编译器要求 Java 程序必须通过捕获或声明所有的检查性异常的方式进行异常处理，否则不能通过编译。在 Java 中，Exception 类中除了 RunTimeException 类及其子类外都是编译时异常。IOException 是典型的编译时异常，IOException 及其子类的继承结构如图 5-1 所示。

(2) 运行时异常(RuntimeException)：指可以通过编译，只有在 JVM 运行时才能发现的异常，如被 0 除、数组下标超范围等。运行时异常的产生比较频繁，处理较为麻烦，对程序可读性和运行效率影响不太大。因此用户可以检测也可以不检测。RuntimeException 及其子类的继承结构如图 5-2 所示。

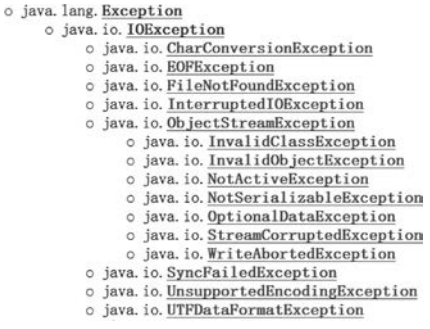

图 5-1 IOException 及其子类的继承结构

图 5-2 RuntimeException 及其子类的继承结构

5.4.3　Java 异常处理机制

Java 异常处理机制主要通过以下两种方式来处理异常。

（1）使用 try…catch…finally 语句对异常进行捕获和处理。

（2）在可能产生异常的方法定义首部用 throws 声明抛出异常。JVM 将类载入内存后调用 main()方法，main()方法再调用其他方法。在 Java 中，采用“谁调用，谁负责处理”的异常处理机制。

1. try…catch…finally 语句

语法格式如下：

```
try{
…//可能产生异常的代码
}catch(异常类型 1 异常对象 1){
…//异常处理代码
}catch(异常类型 2 异常对象 2){
…//异常处理代码
}[finally{
…//不管异常发生与否都执行的代码
}]
```

说明：

（1）try 语句块中包围的是可能产生异常的语句。

（2）finally 子句是 try…catch 的统一出口，一般用来处理“善后工作”。可以把不管异常发生与否都要执行的代码放到这里，例如，关闭数据库连接、关闭 Socket 连接、关闭文件流等代码。

（3）一个 try 语句块可以对应多个 catch 块，用于对多个异常类进行捕获。如果要捕获的异常类之间有父子继承关系时，应该将子类的 catch 块放置在父类的 catch 块之前。

（4）try…catch…finally 语句可以嵌套。

2. 在方法首部 throws 声明抛出异常

如果一个方法没有捕获可能引发的异常，就必须在方法首部声明可能发生异常，交由调用者来处理异常。throws 的语法格式如下：

```
throws <异常类型列表>
```

【示例程序 5-9】 try…catch…finally 语句测试(TryCatchFinally.java)。

功能描述：本程序演示了 try…catch…finally 语句的使用，阅读程序时要注意：程序发生异常时直接跳转到相应的 catch 子句，后面的代码将不会被执行；当有多个 catch 子句时，捕获的异常子类应在前，父类在后；无论发生异常与否，finally 子句中的代码都会被执行。

```
01  public class TryCatchFinally{
02      public static void proc(int mode){
```

```
        try{
            if(mode == 0){
                System.out.println("没有异常发生!");
            }else{
                int j = 4/0;          //出现 ArithmeticException,直接跳转到 12 行
                System.out.println("因 08 行出现异常,本行不被执行!");
            }
        }catch(ArithmeticException e ) {
            System.out.println("捕获到 ArithmeticException 异常!");
        }catch(Exception e ) {
            System.out.println("捕获到 Exception 异常!");
        }finally{
            System.out.println("无论发生异常与否都要执行!");
        }
    }
    public static void main( String args[] ){
        proc(0);                      //没有异常发生
        proc(1);                      //有异常发生
    }
}
```

程序运行结果如下：

```
没有异常发生!
无论发生异常与否都要执行!
捕获到 ArithmeticException 异常!
无论发生异常与否都要执行!
```

5.5 综合实例：编写平面图形程序，理解抽象类和接口

5.5.1 案例背景

面向对象思想更加贴近人类的行为模式。例如平面几何图形，如三角形、矩形、圆等，当不确定具体是什么平面图形时，只能要求计算周长和面积；当确定是某种平面图形后，就可以给出具体求周长和面积的算法。

在面向对象中，接口和抽象类不能被实例化为对象，只能被其他子类继承。

抽象类是包含抽象方法的类。抽象类可以约束所有继承它的类，即：继承抽象类的类必须实现抽象类中所有的抽象方法，否则它也只能是抽象类。一个类只能继承一个抽象类。

接口是抽象方法和常量的集合，可以约束所有实现的类。一个类可以实现多个接口。实现接口的类必须实现接口中的所有抽象方法，否则它只能是抽象类。

5.5.2 编程实践

可以把平面图形类设计成抽象类 AShape，甚至更抽象的接口 IShape，把计算周长和面

积的方法设计成抽象方法。平面图形（Triangle 类、Rectangle 类、Circle 类等）继承 AShape 抽象类或实现 IShape 接口。Triangle 类、Rectangle 类、Circle 类必须实现所有抽象方法，即编写计算周长和面积的算法。

【示例程序 5-10】 平面图形抽象类（CShape.java）。

功能描述： 抽象类 CShape 包含两个抽象方法：getArea()和 getPerimeter()。

```
01  public abstract class CShape {
02      //抽象方法
03      public abstract double getArea();
04      public abstract double getPerimeter();
05  }
```

【示例程序 5-11】 平面图形接口（IShape.java）。

功能描述： 接口 IShape 包含两个抽象方法：getArea()和 getPerimeter()。

```
01  public interface IShape {
02      //接口的方法只能是抽象方法,自动添加 public abstract
03      double getArea();
04      double getPerimeter();
05  }
```

【示例程序 5-12】 三角形类（Triangle.java）。

功能描述： 包含 3 个属性、构造方法、Getter()和 Setter()方法、覆盖 toString()方法、实现了抽象类 CShape 包含的两个抽象方法：getArea()和 getPerimeter()。

```
01  public class Triangle extends CShape{
02      private double a;
03      private double b;
04      private double c;
05      public Triangle() {
06      }
07      public Triangle(double a, double b, double c) {
08          this.a = a;
09          this.b = b;
10          this.c = c;
11      }
12      @Override              //实现父类 CShape 中的抽象方法
13      public double getArea() {
14          double l = (a + b + c)/2;
15          double s = Math.sqrt(l * (l - a) * (l - b) * (l - c));
16          return s;
17      }
18      @Override              //实现父类 CShape 中的抽象方法
19      public double getPerimeter() {
20          return a + b + c;
21      }
```

```
22      @Override            //覆盖父类 Object 中的抽象方法
23      public String toString() {
24          return "Triangle[" + a + "," + b + "," + c + "]";
25      }
26      public double getA() {
27          return a;
28      }
29      public void setA(double a) {
30          this.a = a;
31      }
32      public double getB() {
33          return b;
34      }
35      public void setB(double b) {
36          this.b = b;
37      }
38      public double getC() {
39          return c;
40      }
41      public void setC(double c) {
42          this.c = c;
43      }
44  }
```

【示例程序 5-13】 矩形类(Rectangle.java)。

功能描述：包含了两个属性、构造方法、Getter()和 Setter()方法、覆盖了 toString()方法、实现了抽象类 CShape 包含的两个抽象方法：getArea()和 getPerimeter()。

```
01  public class Rectangle extends CShape{
02      private double w;
03      private double h;
04      public Rectangle() {
05      }
06      public Rectangle(double w, double h) {
07          this.w = w;
08          this.h = h;
09      }
10      @Override              //实现父类 CShape 中的抽象方法
11      public double getArea() {
12          return w * h;
13      }
14      @Override              //实现父类 CShape 中的抽象方法
15      public double getPerimeter() {
16          return 2 * (w + h);
17      }
18      @Override              //覆盖父类 Object 中的抽象方法
19      public String toString() {
20          return "Rectangle[" + w + "," + h + "]";
```

```
    }
    public double getW() {
        return w;
    }
    public void setW(double w) {
        this.w = w;
    }
    public double getH() {
        return h;
    }
    public void setH(double h) {
        this.h = h;
    }
}
```

【示例程序 5-14】 圆形类(Circle.java)。

功能描述：Circle类实现了接口IShape。包含一个属性、构造方法、Getter()和Setter()方法、覆盖了toString()方法、实现了接口IShape包含的两个抽象方法：getArea()和getPerimeter()。

```
public class Circle implements IShape{
    private double r;
    public Circle() {
    }
    public Circle(double r) {
        this.r = r;
    }
    @Override
    public double getArea() {
        return Math.PI * r * r;
    }
    @Override
    public double getPerimeter() {
        return 2 * Math.PI * r;
    }

    @Override
    public String toString() {
        return "Circle[" + r + "]";
    }
    public double getR() {
        return r;
    }
    public void setR(double r) {
        this.r = r;
    }
}
```

5.6 本章小结

本章介绍了面向对象的高级部分,包括类的继承和组合,方法的覆盖,静态和非静态语句块,对象的上溯造型和下溯造型,抽象类,接口,异常处理机制等内容。

经过本章的学习,读者已经基本完成面向对象编程的学习,能够应用 Java 语言解决一定的实际问题。

5.7 自测题

一、选择题

1. Java 采用“_____<父类>”的方式来实现单继承,采用“_____<接口列表>”的方式来实现多继承。

2. 关键字_____指向当前类的对象,关键字_____指向当前类的父类。

3. 关键字_____用于标识成员变量、成员方法、内部类、初始化块等成员是属于类的还是属于对象的。

4. _____指在类体中、方法外定义的有_____修饰的语句块,当其所在类被 JVM 载入内存时,自动执行一次,负责_____ 的初始化。要将一个类载入内存,必须先载入其_____。

5. _____块指在类体中、方法外定义的语句块,当调用_____实例化对象之前,JVM 会自动执行一次,用于_____的初始化。要调用一个类的构造方法,JVM 会自动先调用_____的构造方法。

6. final 用在变量前面,该变量成为_____,只能被赋值一次。final 用在方法前面,该方法成为_____,不能被子类的方法覆盖。final 用在类前面,该类成为_____,只能实例化,不能被继承。

7. 关键字_____修饰的方法为抽象方法(只有方法的定义,没有方法的实现)。含有抽象方法的类必须声明为_____类。

8. _____本质上是一个比 更抽象的类。在接口中只能定义_____ 和_____。

9. 经过多次的上溯造型和下溯造型后,当不能确定某个对象是不是某个类的对象时,可以使用运算符_____来判断。

10. 关键字 private 修饰的成员的可见范围是:_____,没有权限修饰符成员的可见范围是:_____,关键字 protected 修饰的成员的可见范围是:_____,关键字 public 修饰的成员的可见范围是所有包中所有类。

11. 在 Java 中,可以用_____…_____…_____结构对异常进行捕获和处理。也可以在可能产生异常的方法定义首部用_____声明抛出异常。

12. 程序可能发生异常时,应该把不管异常发生与否都执行的代码放到_____子句中。

二、程序运行题

写出下列程序的运行结果。

```
01  class A {
02      int i = 9, j;
03      public A() {
04          prt("i = " + i + ",j = " + j);
05          j = 10;
06      }
07      static {
08          int x1 = prt("A is superclass.");
09      }
10      static int prt(String s) {
11          System.out.println(s);
12          return 11;
13      }
14  }
15  public class B extends A {
16      int k = prt("B is key.");
17      public B() {
18          prt("k = " + k + ",j = " + j);
19      }
20      static int x2 = prt("B is childclass.");
21      public static void main(String args[]) {
22          prt("A is key.");
23          B is = new B();
24      }
25  }
```

三、编程实践

1. 类的继承应用示例程序(ExtendsTest.java)。

编程要求：

(1) Vehicle 车辆类，受保护的属性有 wheels(车轮个数)和 weight(车重)。

(2) Car 汽车类是 Vehicle 类的子类，属性 loader(可载人数)。

(3) Truck 卡车类是 Vehicle 类的子类，属性 payload(载重)。

(4) 要求生成无参构造方法和包含所有属性的构造方法，要求自定义对象输出信息。

(5) 生成每一个类的对象，并测试相关方法。

2. 接口应用示例程序(ImplementsTest.java)。

编程要求：

(1) Biology 生物接口中定义了 breathe()抽象方法(用输出语句模拟即可)。

(2) Animal 动物接口继承了 Biology 接口，增加 eat()和 sleep()两个抽象方法。

(3) Human 人类接口继承了 Animal 接口，增加 think()和 learn()两个抽象方法。

(4) 定一个普通人类 Person 实现 Human 接口，并进行测试。

3. 游戏团队战斗力统计程序(GameRoleTest.java)。

5-5　游戏团队战斗力统计

编程要求：

(1) Role 角色类是所有职业的父类，包含受保护的属性：roleName(角色名字)，public int getAttack()(返回角色的攻击力，因角色不具体，只能定义成抽象方法)。

(2) Magicer 法师类继承了 Role 类,包含私有属性: name(姓名),grade(魔法等级 1～10)。方法: public int getAttack()(返回法师的攻击力,法师攻击力=魔法等级 * 5)。提供私有属性的 Getters/Setter()方法。

(3) Soldier 战士类继承了 Role 类,包含私有属性: name(姓名),attack(攻击力); 提供私有属性的 Getters/Setter()方法。

(4) Team 团队类,一个团队包括一个法师、若干战士(最多 6 个,用 Soldier 数组实现)。私有属性: num(战士实际个数)。定义了两个方法: public boolean addMember(Solider s)方法(当战士的实际个数不超过 6 个时,将 s 赋值给第 num 个数组元素),提供私有属性的 public int attackSum()方法(返回该团队的总攻击力)。

(5) 根据以上描述创建相应的类,并编写相应的测试代码。