Java程序设计

2019春季

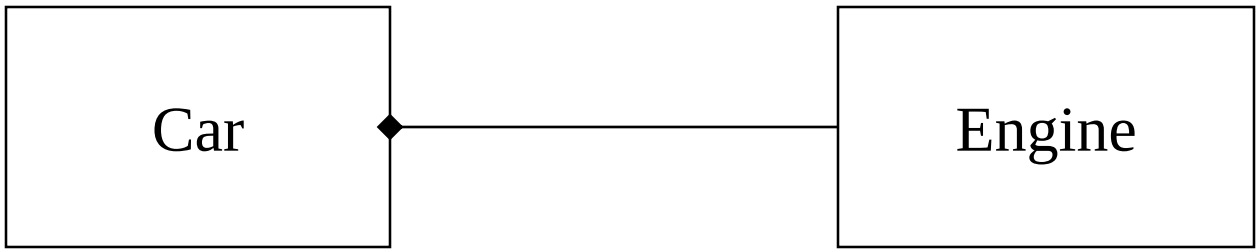
彭启民

pengqm@aliyun.com

继承

组合(compostion)

- 部分—整体关系（“has-a”）。例如，飞机可由发动机、机身、机械控制系统、电子控制系统等构成。
- 一个新类是由其它对象的成员或类型组成合成体，只要把对象的reference直接放到新的类里

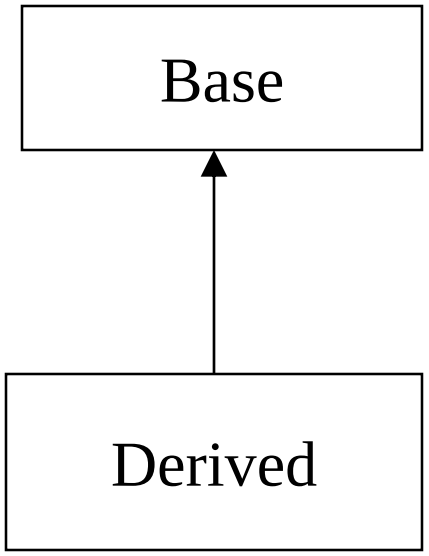


继承

- 现有类派生出新类。
- 显示继承别的类、隐含地继承根类Object.
- 在定义类时，你加上extends和基类的名字,做完后，新类自动获得基类的全部成员和方法。
  - 创建一个定义了共有特性的通用类为被继承的父类，子类可以继承该类，并添加/修改自己的独有特性（字段、行为）。

继承

- Base类:超类或父类
- Derived类: 继承类或子类

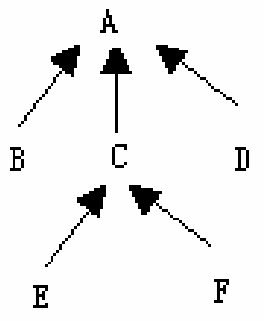


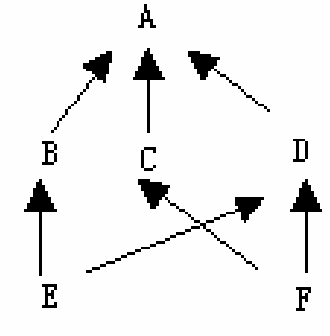
继承

- 类之间的继承关系表示类之间的内在联系以及对属性和操作的共享，即子类可以沿用父类（被继承类）的某些特征。
- 子类也可以具有自己独立的属性和操作。
  - Is-a: 仅仅重写; 例如, 圆是一种图形
  - Is-like-a: 添加新的函数; 例如, 三角形具有图形的特性

- 子类中常要加入它自己的实例变量和方法，故子类一般要比它的父类大（？）。
  - 并不因为是父类就意味着有更多的功能。实际相反，子类比其父类具有更多的功能。因为子类是父类的扩展，增加了父类没有的属性和方法。
- 子类比父类更具体，因而代表了较少的对象。

- 如果子类只从一个父类继承，则称为单继承；如果子类从一个以上父类继承，则称为多继承。
- Java不支持多重继承，但它支持“接口”。接口使Java获得了多重继承的许多优点，摒弃了相应的缺点。





- 创建格式如下：

［类修饰符］class 子类名 ［extends 父类名］［implements 接口名称列表］
{
    类体
}

子类/扩展类

- 父类名跟在extends 关键字后面，用来说明继承关系，即当前类是哪个已经存在类的子类。
  - 添加新的属性和方法（增加）
  - 重写: 改变现存基类函数的行为（替换）
- 子类从父类继承有两个主要的方面：
  - 属性的继承
    - 例如，公司是一个父类，一个公司有名称、地址、经理、雇员等，这些都属于结构方面。
  - 方法的继承
    - 一个父类定义了若干操作，如一个公司要有任命经理、录用职工等操作，子公司也将继承这些行为。

Inheritance

```
public class StringWrapper {
    String s = new String("");
    public void append(String a) { s
  += a; }
  } // end class StringWrapper

  public class StringPrinter extends
  StringWrapper {
    public void print() {
        System.out.println("s="+s);
    }
  } // end class StringPrinter
```

Inheritance

```
public class MyClass {
    public static void main(String[] args) {
        StringPrinter mySP = new
StringPrinter();
        mySP.append("Hello");
        mySP.append(" there");
        mySP.print();
    } // end main method
} // end class MyClass
```

- 尽管被继承的父类成员没有在子类声明中列出，但是这些成员确实存在于子类中。
  - 属性、方法.

- 成员访问和继承
  - 子类可以访问其父类的public成员，但不能访问父类的private成员。
  - protected访问是public和private访问之间一个保护性的中间层次,从子类的方法能直接引用父类的protected类型的实例变量。

类分级结构

- Object类（java.lang包）：Java中所有类的根
  - Object是Java中的一个特殊类，它是所有类的一个隐性的超类。Java中的每一个类都是从java.lang.Object类中继承， Object类提供了一套任何类的任何对象均可使用的方法。
  - 当类被定义后，如果没有指定继承，那么类的超类是Object。
  - 其它所有类都是Object的子类。这就意味着Object类型的引用变量可以引用任何类的对象。而且，因为数组是作为一个类来实现的，所以一个Object类型的变量也可以引用任何数组。

- Runtime 类（java.lang包）
  - Java程序的运行环境。
  - 每个Java程序都有一个自动创建的Runtime实例

Runtime.getRuntime() //获取当前程序的Runtime实例

方法继承、覆盖与重载

- 方法继承（ Inheritance ）
  - 对于子类对象，可以使用父类的方法，即使这些方法没有明显地在子类中定义，它们也自动地从父类中继承过来了。
- 方法覆盖（ Overriding ）
  - 子类定义同名方法来覆盖父类的方法，是多态技术的一个实现。当父类方法在子类中被覆盖时，通常是子类版本调用父类版本，并做一些附加的工作。

- **方法重载**（overloading）
  - 可以用相同的方法名但不同的参数表来定义方法（参数表中参数的数量、类型或次序有差异），这称为方法重载。
  - 编译器通过将在不同方法头部中的参数类型和在特定的方法调用中使用值的类型进行比较，从而挑选出正确的方法。

关于方法覆盖

- 当通过父类引用调用一个方法时，Java会正确地选择与那个对象对应的类的覆盖方法。
- 方法覆盖中，子类在重新定义父类已有的方法时，应保持与父类完全相同的方法头声明，即与父类完全相同的方法名、返回值和参数列表。
- 子类可以添加字段，也可以添加方法或者覆盖父类中的方法。然而，继承不能去除父类中的任何字段和方法。

控制符和修饰符

访问控制符

Java推出了“访问控制符”的概念，访问控制的级别在“最大访问”和“最小访问”的范围之间，分别包括：public，“defaut”（无关键字），protected以及private。

- 设定客户端程序员可以使用和不可以使用的界限
- 将接口和具体实现进行分离

	类的访问控制符（一种符号）				
	public		公共访问		
属性与方法的访问控制符（四种符号）					
符号	**public**	**protected**	**private protected**	**private**	
含义	公共访问	保护访问	私有保护访问	私有访问	

缺省访问控制符

一个类没有访问控制符，则为缺省的访问控制特性。

该类只能被同一个包中的类访问和引用，而不可以被其他包中的类使用，这种访问特性称为包访问性。

通过声明类的访问控制符可以使整个程序结构清晰、严谨，减少可能产生类间干扰和错误。

公共访问控制符public

Java中类的访问控制符：public，即公共的。

（类的修饰符还有： abstract final）

一个类被声明为公共类，表明它可以被所有的其他类所访问和引用，这里的访问和引用是指这个类作为整体是可见和可使用的，程序的其他部分可以创建这个类的对象、访问这个类内部可见的成员变量和调用它的可见的方法。

私有访问控制符private

用private修饰的属性或方法只能被该类自身所访问和修改，而不能被任何其他类，包括该类的子类，来获取和引用。

- **1. 对私有数据访问的方法**

例如:有三个实例字段，它们含有在Employee类的实例内部被操作的数据。

```
private string name;
private double salary;
private Date hireDay;
```

private(私有的）关键字用来确保可以访问这些实例字段的只能是Employee类本身的方法。

私有访问控制符private

- **2. 私有方法**

在下面的情况下可以选择私有方法：
（1）与类的使用者无关的那些方法。
（2）如果类的实现改变了，不容易维护的那些方法。

保护访问控制符protected

- 用protected修饰的成员变量可以被三种类所引用：该类自身、与它在同一个包中的其他类、在**其他包中**的该类的子类。
- 使用protected修饰符的主要作用是允许其他包中的它的子类来访问父类的特定属性。

- **私有保护访问控制符private protected**

private和protected按顺序连用构成一个完整的访问控制符：私有保护访问控制符。用private protected修饰的成员变量可以被**两种类**访问和引用，一种是该类本身，一种是该类的所有子类，不论这些子类是与该类在同一个包里，还是处于其他的包中。

相对于protected，private protected修饰符把同一包内的非子类排除在可访问的范围之外，使得成员变量更专有于具有明确继承关系的类，而不是松散地组合在一起的包。

构造函数(构造方法)

- 生成实例的方法是通过“new”调用类的构造函数（构造方法）。
- 与类名同名，没有返回类型。

构造函数

- 使用对象，首先必须构造它们，并指定它们的初始状态，然后将方法应用于对象。
- 在Java程序设计语言中，使用构造函数/构造器（constructor）来构造新的实例。
- 一个构造函数是一个新的方法，它的作用是构造并初始化对象。
- 一旦为某个类定义了构造函数，那么在创建对象时，构造函数将立即被调用。

构造函数的作用

- 对象初始化
- 引入更多的灵活度（变量赋值或更复杂的操作）
- **Java中可以不定义构造函数，此时系统会自动为该系统生成一个默认的构造函数。这个构造函数的名字与类名相同，它没有任何形式参数，也不完成任何操作。**
  - 无论您是否定义构造函数，所有的类都有构造函数，因为Java自动提供了一个默认的构造函数来初始化所有的成员变量为0。一旦定义了自己的构造函数，Java就不会再使用默认构造函数了。
- **为了避免失去控制，一般将构造函数的声明与创建分开处理。**

构造函数是类的一种特殊方法

- 构造函数的方法名与类名相同。
- 构造函数没有返回类型。
- 一个类可以有多个构造函数。
- 构造函数可以有0个、1个或多个参数。
- 构造函数的主要作用是完成对类对象的初始化工作。
- 在创建一个类的新对象的同时，系统会自动调用该类的构造函数为新对象初始化。

构造函数vs.一般函数

- 写法不同
- 运行不同
  - 构造函数在对象建立时运行，只运行一次
  - 一般方法可多次调用
- 功能不同
  - 构造函数用于初始化对象。
  - 一般方法给对象添加具具体功能。

构造器编写规则

- 避免在类的构造器中初始化其他类
  - 可能会引起巨大的类初始化的“链式反应”!
- 避免在构造器中对静态变量赋值
  - 创建多个对象时将引起不必要的赋值
- 不要在构造器中创建类的实例
- 访问修饰符，访问权限修饰符对构造方法的重载不影响。
- 当一个类中没有定义构造函数时，那么系统会默认加入一个空参数的构造函数。
- 当在类中自定义了构造函数后，默认的构造函数就没有了。

```
public class Test {
    public static String name="张三";
    public int id;
    public static String getName()
    {
       return name;
    }
}
Test t = new Test();
```

```
public class Test {
    public static String name="张三";
    public int id;
    public Test(){ };
    public static String getName()
    {
       return name;
    }
}
Test t = new Test();
```

```
public class Test {
    public static String name="张三";
    public int id;
    public Test(int i){ id = i;};
    public static String getName()
    {
       return name;
    }
}
Test t = new Test();//？？？
```

```
public class Test {
    public static String name="张三";
    public int id;
    public Test(int i){ id = i;};
    public static String getName()
    {
       return name;
    }
}
Test t = new Test(2);
```

```
public class Test {
    public static String name="张三";
    public int id;
    public Test(){};//如果没有这个形式的构造函数会如何？
    public Test(int i){ id = i;};
    public static String getName()
    {
       return name;
    }
}
Test t1 = new Test(2);
Test t2 = new Test();//若第四行，将不会有默认的无参构造函数！
```

基类的初始化

- 派生类的构造函数自动地调用基类的构造函数。
- 构造行为从基类开始，基类会在派生类的构造函数访问它之前先进行初始化
- 如果基类为带参数的构造函数，则子类必须有参数传基类以便正常构造
  - 注意：会妨碍派生类的构造函数捕获基类抛出的异常

在子类中使用构造函数

- 子类构造函数总是先调用（显式的或隐式地）其父类的构造函数，以创建和初始化子类的父类成员。
- 构造函数不能继承，它们只属于定义它们的类
- 当创建一个子类对象时，子类构造函数首先调用父类的构造函数并执行，接着才执行子类构造函数。基类的构造函数是第一个执行，被包含类只有使用时才构造。

```
class Art {
      public Art() {
         System.out.println("Art constructor");
      }
} // end class Art
class Drawing extends Art {
      public Drawing() {
         System.out.println("Drawing constructor");
      }
} // end class Drawing
class Cartoon extends Drawing{
      public Cartoon() {
         System.out.println("Cartoon constructor");
      }
} // end class Drawing
Art a = new Cartoon();
```

- 调用另一个构造函数
  - 关键字this指向隐式参数。如果构造函数第一个语句具有形式this(……)，那么此构造函数调用此类中的另一个构造函数。

```
public Employee(double s)
{
    //calls Employee(String，double)
    this(“Employee #”+nextId，s);
    nextId++;
}
```

- **默认初始化**
  - 如果在构造函数中没有明确地给某个字段赋值，那么此字段会被自动地赋值以一个默认值：
    - 数字被赋值以0
    - 布尔类型被赋值以false
    - 对象引用被赋值以null。

- **显式初始化**
  - 在类的定义中，可简单地把一个值赋值给任何字段。在执行构造函数前，此赋值会被执行。
  - 当类中所有的构造函数都需要把某一特定的实例字段赋值以相同的值时，此语法非常有用。

```
private String name=”lili”;
```

初始化块

- 在类声明中可以包含任意数量的代码块。只要构造了此类的一个对象，这些代码块就会被执行。
- 初始化块首先被运行，然后构造函数的主体部分被执行。
- 是构造器的补充，不能接收任何参数，用于定义类实例化生成的所有对象共有的属性、方法等内容。

- 初始化块
  - 静态初始化块
    - 使用static定义,当类装载到系统时执行一次.若在静态初始化块中想初始化变量,那仅能初始化static修饰的数据成员.

    static {

    }
  - 非静态初始化块
    - 在每个对象生成时都会被执行一次,可以初始化类的实例变量。

```
class Employee
{
        public Employee(String n，double s)
        {
                name=n;
                salary=s;
        }

       public Employee()

       {
              name=” ”;
              salary=0;
       }
  //对象初始化模块
      {
              id=nextId;
              nextId++;
      }
       private String name;
       private double salary;
       private int id;
       private static int nextId;
}
```

```
class Root
{
        {
                System.out.println("Root的普通初始化块");
        }
        public Root()
        {
                System.out.println("Root的无参数构造函数");
        }
        public Root(String name)
        {
                this();
                System.out.println("Root的带参构造函数"+name);
        }
}
class Leaf extends Root
{
        {
                System.out.println("Leaf的普通初始化块");
        }
        public Leaf()
        {
                super("abc");
                System.out.println("Leaf的无参构造函数");
        }
}
public class Testt{
        public static void main(String[] args)
        {
                new Leaf();
        }
}
```

运行结果：
Root的普通初始化块
Root的无参数构造函数
Root的带参构造函数abc
Leaf的普通初始化块
Leaf的无参构造函数

先执行的是父类的初始化块，父类的构造函数,再初始化子类的初始化块，子类的构造函数。

super()方法执行父类的构造方法随父类的构造方法一起执行，其它方法是不会在初始化块前面执行。

```
class Blocks {
    static {
        System.out.println("父类静态初始化块");
    }

    {
        System.out.println("父类初始化块");
    }

    Blocks() {
        System.out.println("父类构造函数");
    }
}

public class InitailizeBlocks extends Blocks {
    static {
        System.out.println("子类静态初始化块");
    }

    {
        System.out.println("子类初始化块");
    }

    public InitailizeBlocks() {
        System.out.println("子类构造函数");
    }

    public static void main(String[] args) {
        new InitailizeBlocks();
    }
}
```

输出结果：
父类静态初始化块
子类静态初始化块
父类初始化块
父类构造函数
子类初始化块
子类构造函数

静态代码块在类加载时被执行，而非静态代码(包括初始化代码块和构造函数)在生成对象时才被执行，故父类和子类的静态初始化代码块最早执行；初始化块与构造函数的执行顺序，前者要早于后者。

一个java文件从被加载到被卸载要经历4个阶段：
加载->链接（验证+准备+解析）->初始化（使用前的准备）->使用->卸载
其中加载、链接的过程是由jvm负责（自定义加载除外）

- this和super是常用来指代父类对象与子类对象的关键字.
  - this表示的是当前对象本身，更准确地说，this代表了当前对象的一个引用。对象的引用可以理解为对象的另一个名字，通过引用可以顺利地访问对象，包括修改对象的属性、调用对象的方法。
  - super表示的是当前对象的直接父类对象，是当前对象的直接父类对象的引用。所谓直接父类是相对于当前对象的其他“祖先”而言的。

- 在Java中利用关键字super子类可以调用父类构造函数，也可以访问父类成员或直接调用父类中的方法。

- 例：

  定义 雇员类 Employee 的两个子类：

  一般雇员类:CommonEmployee

  主　管　类:ManagerEmployee

  public class CommonEmployee extends Emplyee

  public class ManagerEmployee extends Emplyee

```
public ManagerEmployee（String n,double s,int
   year,int month,int day）
{
    super(n,s,year,month,day)；
    bonus=0；
}
```

- super(n,s,year,month,day)：以n、s、year、month、day为参数调用Employee父类的构造函数。

- 构造函数不被继承, 它们只属于定义它们的类。
- 子类构造函数总是先调用（显式的或隐式地）其父类的构造函数以创建和初始化子类的父类成员.
  - 显式调用时super（调用参数列表）语句是第一条语句。

多态：抽象类与接口

- 例:

```
public class Male{
    public void drinkWater(){
    //豪爽地
    …
    };
}
public class Famle{
    public void drinkWater (){
    //优雅地
    …
    };
}
```

- 例:

```
public class H{
    public void excute(Male m){
       m. drinkWater();…
    };
    public void excute(Famle f){
       f. drinkWater();…
    };
}
```

```
public class People{
    public void  drinkWater (){
    …
    };
}
```

```
public class Male extends People{
public void drinkWater(){
//豪爽地
…
};
}
public class Famle extends People {
public void drinkWater (){
//优雅地
…
};
}
```

- 例:

```
public class H{
    public void excute(People p){
       p. drinkWater();…
    };
}
```

- 将一条消息发给对象时，如果并不知道对方的具体类型是什么，但采取的行动同样是正确的，这种情况就叫作“多态/多形性”（Polymorphism）。
  - 引用类型
  - 实例对象类型
  - 可用于方法参数和方法返回类型
  - 对应于简单工厂模式

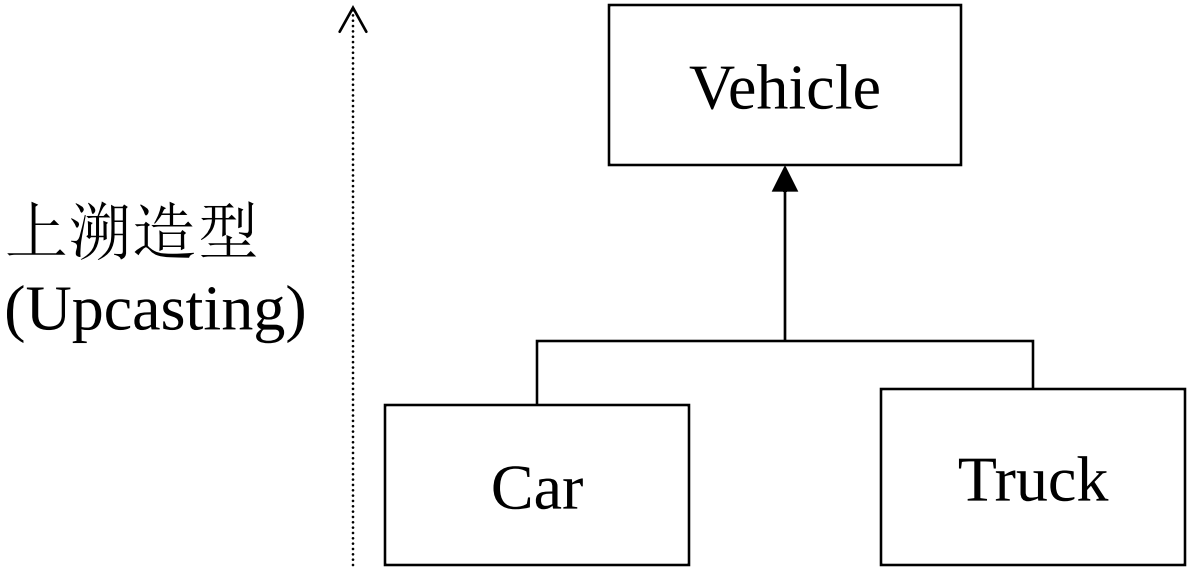
- 把衍生类型当作它的基本类型处理的过程叫作“Upcasting”（上溯造型）。
- 不是上溯造型成一种更“通用”的类型，而是造型成一种更“特殊”的类型，这种造型方法叫作“下溯造型”（Downcasting）。

上溯造型/上传

- 新类就是一种原有的类
- 派生类到基类的上传总是安全的，因为你是把一个较具体类型转换优较为一般的类型。
- 在上传的过程中，类的接口只会减小，不会增大。

上溯造型

- 将继承类作为基类来处理



- 方法的重写Overriding和重载Overloading是Java多态性的不同表现。
  - 重写Overriding是父类与子类之间多态性的一种表现
    - 如果在子类中定义某方法与其父类有相同的名称和参数，则该方法被重写 (Overriding)。子类的对象使用这个方法时，将调用子类中的定义，对它而言，父类中的定义如同被"屏蔽"了。
  - 重载Overloading是一个类中多态性的一种表现。
    - 如果在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型，则称为方法的重载(Overloading)。Overloaded的方法是可以改变返回值的类型。

- 多态提高了程序可扩充性，调用多态性行为的软件传送给对象的消息（即方法调用）与对象的类型无关，因此能响应已有消息的新类型可以直接加入系统，而不用修改基本系统。

多态性

- 静态多态性（早期绑定: 编译期 (C++缺省模式)）
- 动态多态性（后期绑定: 运行期 (Java缺省模式)）：
  - 系统会自动根据运行时的实例类型调用相应的方法。
  - 用父类的引用作为方法的参数，而在方法调用时传入子类的实例。

- 调用一个对象方法的机制
  - 编译器检查对象的声明类型和方法名。
  - 编译器检查方法调用中的参数类型。如果在所有的叫做f的方法中有一个其参数类型同调用提供的参数类型最匹配，那么该方法就会被选择调用。这个过程称作超载选择。
  - 当程序运行并且使用动态绑定来调用一个方法时，那么虚拟机必须调用同x所指向的对象的实际类型相匹配的方法版本。

父类对象与和子类对象转化的原则

- 子类对象可以被视为是其父类的一个对象；
- 父类对象不能当成是其某一个子类的对象；
- 如果一个方法的形式参数定义的是父类对象，那么调用这个方法时，可以使用子类对象作为形式参数；
- 如果父类对象引用指向的实际是一个子类对象，那么这个父类对象的引用可以用强制类型转换转化成子类对象的引用。

- 子类不能访问父类的private成员，但子类可以访问其父类的public，protected和包访问成员；要访问父类的包访问成员，子类一定要在父类的包内。
- 子类构造函数总是先调用（显式的或隐式地）其父类的构造函数，以创建和初始化子类的父类成员。
- 子类的对象可以当作其父类的对象对待，反之则不行。

- 父类可以是一个子类的直接父类或间接父类。直接父类是子类显式使用extends说明的类，间接父类是子类从层次结构树的前几层上继承的类。当父类某成员不适合一个子类时，可以在子类中覆盖它。
  - 继承性实现了软件的复用，这不但节省了开发时间，也鼓励人们使用已经验证无误和调试过的高质量软件。
- 一个子类对象引用可以隐式地转换成一个父类对象引用。使用显式的类型转换，可以把父类引用转换成子类引用。如果目标不是子类对象，将产生Class CastException例外处理。
- 父类代表共性，从一个父类派生的所有类都继承了这个父类的功能。

- 当通过父类引用调用一个方法时，Java会正确地选择与那个对象对应的类的覆盖方法。在多态性中，一个方法调用可能会产生不同动作，这取决于接受调用的对象类型。
- 当从具体类（子类）中实例化某些对象时，这些引用可以用来对这些对象进行多态性操作。经常要把一些新类加入系统，这时可以用动态方法绑定（也称迟绑定）接纳它们。在编译方法调用时，不需要知道对象的类型。只有执行时，才会选择相应对象的方法。
- 在动态方法绑定中，执行方法调用时系统将会找到接受调用对象所属的类的相应方法。对于父类提供的方法，子类可以覆盖它的父类版本。

- 多态实现不依赖具体的类，而是依赖于抽象。
- Java中的多态通过抽象类和接口实现。

- abstract是抽象修饰符，可以用来修饰类或方法。

抽象类

- 抽象类:继承类的基类; 没有具体实例对象的类，包含抽象方法的类。
  - 当一个类被声明为abstract时被称为抽象类。
  - 抽象类不能实例化为任何对象。
- 建立抽象类的意图就是为所有由它派生出来的类创建一个公共接口。

```
abstract class Demo {
        string name;
        abstract void method1();
        abstract void method2();
        int myplus(short) a, short b){
          return (int)(a+b);
        }
        …
}
```

抽象方法

- 抽象方法无函数定义;继承类必须实现该函数，它属于一种不完整的方法，只含有一个声明，没有方法主体。
- 抽象方法声明时采用的语法：

```
abstract void X();
```

接口

- 多重继承是指一个子类继承多个父类。Java不支持多重继承，但Java提供了接口。
- 接口具有多继承的许多优点，而却没有它的缺点。

接口

- 在面向对象的程序设计中，定义一个类必须做什么而不是将怎样做有时是很有益的。
- 抽象方法为方法定义了签名但不提供实现方式。一个子类必须提供自己的由其超类定义的抽象方法的实现方式。这样，抽象方法就指定了方法的接口而不是实现方式。

接口

- Java接口（Interface）是对符合接口需求的类的一套规范。
- 在Java中用关键字interface把一个类的接口和实现方式完全分开。

```
public interface calrect {
    public static final float t_pi=3.14159;
    public abstract int calarea();
    public abstract int calgirth();
    public abstract int getx();
    public abstract int gety();
}
```

- 接口是用来组织应用中的各类并调节它们的相互关系的一种结构，接口主要作用是可以帮助实现类似于类的多重继承的功能
  - Java中出于简化程序结构的考虑，只支持单重继承，即一个类至多只能有一个直接父类。然而在解决实际问题的过程中，仅仅依靠单重继承在很多情况下都不能将问题的复杂性表述完整，需要其他的机制作为辅助。

- Java中声明接口的语法

```
[public]  interface  接口名  [extends  父接口名列表]
{  //接口体;
   //常量域声明
   [public] [static] [final] 域类型 域名=常量值;
   //抽象方法声明
   [public][abstract][native]返回值 方法名(参数列表)
    [throw异常列表];
}
```

接口声明

- 所有成员域都具有public,static,final的属性
  - 接口中可以包含pubic final static修饰的常量数据，但这不是必须的。

接口声明

- 所有方法都默认具有public,abstract属性
  - 接口中的方法都是用abstract修饰的抽象方法
  - 只有说明没有定义
    - 在接口中只能给出这些抽象方法的方法名、返回值和参数列表，而不能定义方法体，即仅仅规定了一组信息交换、传输和处理的“接口”。
  - 注意新版本中的变化！

接口声明

- JDK1.8：接口里面可以有方法体，lamdba语法

```
//方法可以有方法体，要加上default
//可以有静态方法
public interface Drawable {
     int a = 10;
    void draw(int i);
    default  void  method(){
        System.out.println("default修饰");
    }
   static void info(){
       System.out.println("static修饰");
      }
}
```

接口声明

- JDK1.9：增加了私有方法的声明

```
public interface Drawable {
     int a = 10;
    void draw(int i);
    default  void  method(){
        System.out.println("default修饰");
    }
   static void info(){
       System.out.println("static修饰");
   }
   private static void Test(){ }
}
```

美国当地时间2019年 3 月 19 日， Java 12 正式发布。

- 为使用接口，一个类必须声明实现（关键字implements）接口，按指定的参数个数和返回类型定义每个方法。
  - 在类的声明部分，用implements关键字声明该类将要实现的接口，可以是多个。

接口的实现

- 一个类在实现某接口的抽象方法时，必须使用完全相同的方法头。
- 接口的抽象方法，其访问限制符都已指定是public，所以类在实现方法时，必须显式地使用public修饰符。

接口的实现

- 如果实现某接口的类不是abstract的抽象类，则在类的定义部分必须实现指定接口的所有抽象方法，即为所有抽象方法定义方法体，而且方法头部分应该与接口中的定义完全一致，即有完全相同的返回值和参数列表；
- 如果实现某接口的类是abstract的抽象类，则它可以不实现该接口所有的方法。

接口的实现

- 接口中的静态方法，只能使用接口名称调用。
- 接口中default方法属于实现接口类的对象方法，可以重写。
- 接口中的private static方法只能在接口内调用。
- 继承的接口实现子接口的抽象方法需要加dafault