

第七章 抽象、接口、内部类、枚举

1 abstract

1.1 问题引入

用前面学习的内容，实现下面案例，需求如下：

定义猫类（Cat）和狗类（Dog）

猫类成员方法：eat：猫吃鱼 sleep：躺着睡

狗类成员方法：eat：狗吃肉 sleep：趴着睡

根据前面所学基本内容（继承、重写），代码实现如下：

```
package com.briup.chap07.test;

//基类
class Animal {
    public void eat() {
        System.out.println("吃饭，吃的东西不确定...");
    }

    public void sleep() {
        System.out.println("睡觉，姿势不确定...");
    }
}

//子类Cat，重写方法
class Cat extends Animal {
    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }

    @Override
```

```

        public void sleep() {
            System.out.println("躺着睡");
        }
    }

    //子类Dog, 重写方法
    class Dog extends Animal {
        @Override
        public void eat() {
            System.out.println("狗吃肉");
        }

        @Override
        public void sleep() {
            System.out.println("趴着睡");
        }
    }

    public class Test011_Animal {
        public static void main(String[] args) {
            Animal a = new Animal();
            a.eat();
            a.sleep();

            System.out.println("-----");

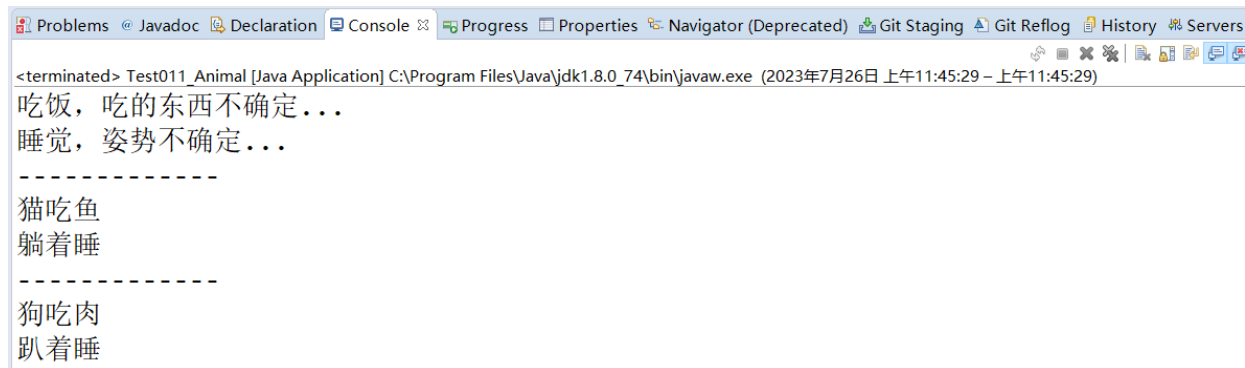
            Animal c = new Cat();
            c.eat();
            c.sleep();

            System.out.println("-----");

            Animal d = new Dog();
            d.eat();
            d.sleep();
        }
    }
}

```

运行效果：



```
<terminated> Test011_Animal [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月26日 上午11:45:29 - 上午11:45:29)
吃饭，吃的东西不确定...
睡觉，姿势不确定...
-----
猫吃鱼
躺着睡
-----
狗吃肉
趴着睡
```

思考，上述代码能够完成相应业务，但是否存在不合理的地方？

- 父类 `Animal` 中 `eat`、`sleep`，这两个方法的功能是不明确的，如何通过代码体现这种不明确
- 实际开发中，一般不会实例化 `Animal` 类对象，主要用 `Animal` 声明引用指向子类对象，如何通过代码保证 `Animal` 类不能实例化对象

上述问题，我们可以借助抽象方法和抽象类来实现！

1.2 抽象概述

抽象，简单可理解为不具体、高度概括的，专业描述为：抽象是一种将复杂的概念和现实世界问题简化为更易于理解和处理的表示方法。在计算机科学和编程中，抽象是一种关注问题的本质和关键特征，而忽略具体实现细节的方法。

在面向对象编程中，**抽象是通过定义类、接口和方法来实现的**，本章我们重点讨论抽象方法和抽象类，具体描述如下。

抽象方法：

将共性的行为（方法）抽取到父类之后，发现该方法的实现逻辑无法在父类中给出具体的实现，就可以将该方法定义为抽象方法。

抽象类：

如果一个类中存在抽象方法，那么该类就必须声明为抽象类。

1.3 抽象特点

抽象方法和抽象类必须使用 `abstract` 关键字修饰

```
//抽象方法定义格式
[权限修饰符] abstract 返回值类型 方法名(参数列表);
//特别注意：抽象方法只有方法声明，没有方法的实现
// 权限修饰符一般为public

//抽象类定义格式
[权限修饰符] abstract class 类名 {
    0或多个数据成员

    0或多个构造方法

    0或多个成员方法
}
```

抽象类和抽象方法的关系：

- 使用abstract修饰的类就是抽象类
- 抽象类可以包含，也可以不包含抽象方法
- 包含抽象方法的类，一定要声明为抽象类

抽象类和普通类区别：

- 抽象类必须使用abstract修饰符
- 抽象类相对普通类，多了包含抽象方法的能力

- 抽象类相对普通类，失去了实例化创建对象的能力

抽象类和普通类相同点：

- 符合继承关系特点，能够使用多态机制
- 子类可以重写从抽象类继承的方法
- 实例化子类对象需要借助父类构造器实现父类部分的初始化

基础案例：

要求：定义一个形状类Shape，包含抽象方法getArea()，再定义其子类Circle，重写抽象方法并进行功能测试。（该案例对抽象方法、抽象类基本知识点进行考核）

```
package com.briup.chap07.test;

// 因为该类中包含了抽象方法，则该类必须声明为抽象类
//定义形状类
abstract class Shape {
    private String name;

    public Shape() {
        System.out.println("Shape()...");
    }

    public Shape(String name) {
        System.out.println("Shape(String)");
        this.name = name;
    }

    //定义抽象方法，获取形状的面积
    //因为现在形状不确定，获取面积的功能实现也是不确定的，故而定义为抽象方法
    public abstract double getArea();
}
```

```

}

//正常子类：必须重写所有抽象方法
class Circle extends Shape {
    //半径
    private int r;

    public Circle() {
        //super();
        System.out.println("Circle()...");
    }

    //子类构造器需要借助抽象父类构造器，完成父类部分的初始化
    public Circle(String name, int r) {
        super(name);
        System.out.println("Circle(String, int)");
        this.r = r;
    }

    //重写抽象方法
    @Override
    public double getArea() {
        return 3.14 * r * r;
    }
}

//抽象类 抽象方法 基础测试
public class Test013_AbstractBasic {
    public static void main(String[] args) {

        //1.抽象类不能实例化对象，下面一行编译报错
        //Shape p = new Shape();    //error

        //2.抽象类引用可以指向子类（非抽象类）对象
        Shape p = new Circle("圆",2);

        //3.实际开发中，主要多态应用
        double area = p.getArea();
    }
}

```

```
        System.out.println("area: " + area);
    }
}
```

//程序运行效果:

Shape(String)...

Circle(String, int)...

area: 12.56

1.4 动物案例

结合抽象类、抽象方法基础内容补充，改良优化动物案例。

案例如下：

抽象父类Animal：

```
package com.briup.chap07.bean;

//该类中包含抽象方法，故而必须声明为抽象类
public abstract class Animal {
    //抽象类可以包含数据成员
    private String color;
    private int age;

    //抽象类中可以包含抽象方法
    public Animal() {}
    public Animal(String color, int age) {
        this.color = color;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Animal [color=" + color + ", age=" + age + "]";
    }
}
```

```

//因为eat和sleep在父类中无法确定其具体功能，故而设置为抽象方法
public abstract void eat();
public abstract void sleep();
}

```

正常子类Cat:

```

package com.briup.chap07.bean;

public class Cat extends Animal {
    //温顺度: 1-10
    private int meekValue;

    public Cat() {}
    public Cat(String color, int age, int meekValue) {
        super(color, age);
        this.meekValue = meekValue;
    }

    //子类重写所有的抽象方法
    @Override
    public void eat() {
        System.out.println("猫吃鱼");
    }
    @Override
    public void sleep() {
        System.out.println("躺着睡");
    }

    @Override
    public String toString() {
        String msg = super.toString();
        msg = "Cat: 温顺度=" + this.meekValue + ", " + msg;
        return msg;
    }
}

```


抽象子类Dog:

```
package com.briup.chap07.bean;

//子类中包含抽象方法，故而必须声明为抽象类
//不可以实例化对象
//public class Dog extends Animal {
public abstract class Dog extends Animal {
    //忠诚度
    private int loyalValue;

    public Dog() {}
    public Dog(String color, int age, int loyalValue) {
        super(color, age);
        this.loyalValue = loyalValue;
    }

    //子类重写部分抽象方法
    @Override
    public void eat() {
        System.out.println("狗吃肉");
    }

    //子类不重写sleep()抽象方法，则当前类仍旧包含抽象方法
    // @Override
    // public void sleep() {
    //     System.out.println("趴着睡");
    // }

    @Override
    public String toString() {
        String msg = super.toString();
        msg = "Dog: 忠诚度=" + this.loyalValue + ", " +
super.toString();
        return msg;
    }
}
```

基础测试：

```
package com.briup.chap07.test;

import com.briup.chap07.bean.Animal;
import com.briup.chap07.bean.Cat;
import com.briup.chap07.bean.Dog;

public class Test014_Animal {
    public static void main(String[] args) {

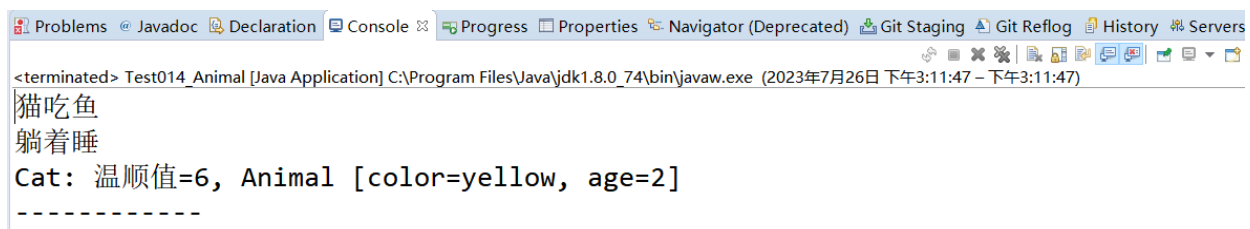
        //1.抽象父类 不可以实例化 对象
        //Animal a = new Animal(); //编译报错 error

        //2.抽象父类引用 可以指向 正常子类对象
        Animal c = new Cat("yellow", 2, 6);
        c.eat();
        c.sleep();
        System.out.println(c);

        System.out.println("-----");

        //3.子类如果是抽象类，不可以实例化对象
        // Animal d = new Dog("black", 2, 8); //编译报错
        // d.eat();
        // d.sleep();
        // System.out.println(d);
    }
}
```

运行效果：



```
<terminated> Test014_Animal [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月26日 下午3:11:47 - 下午3:11:47)
猫吃鱼
躺着睡
Cat: 温顺值=6, Animal [color=yellow, age=2]
-----
```

注意：子类继承抽象类，如果没有重写所有的抽象方法，则子类也是抽象类

额外测试1:

大家也可以把Dog子类中重写的sleep方法、测试类Dog对象测试放开，运行程序，观察效果！

额外测试2:

在上述测试基础上，去除Dog类的abstract修饰，再次运行程序，观察效果！

2 interface

引用数据类型：类、数组、接口

2.1 接口概述

接口是对Java单继承的补充。Java只支持单继承（亲爹唯一），如果在开发过程中想额外增强类的功能，可以借助接口实现（可以拜师，拜多个师傅也可以）。

接口是Java中一种重要的抽象机制，它提供了一种定义行为规范和实现多态性的方式。通过合理使用接口，可以提高代码的可扩展性、可维护性和灵活性。

接口是除了类和数组之外，另外一种**引用数据类型**，需要使用 `interface` 关键字来定义，接口最终也会被编译成 `.class文件`，但一定要明确接口并不是类，而是另外一种引用数据类型。

接口基础定义格式:

```
[修饰符] interface 接口名 {  
    //数据成员，可以定义多个  
    [public static final] 数据类型 数据成员 = 值;  
  
    //抽象方法：可以定义多个  
    [public abstract] 返回值类型 方法名(形参列表);  
}
```

案例如下：

```
package com.briup.chap07.test;  
  
//使用interface关键字来定义接口  
interface IAction {  
    //数据成员,下面2行效果一样  
    //public static final int NUM = 10;  
    int NUM = 10;  
  
    //成员方法,下面2行效果一样  
    //public abstract void start();  
    void start();  
  
    public abstract void end();  
}  
  
public class Test021_Basic {  
    public static void main(String[] args) {  
        //接口不可以实例化对象  
        //IAction ac = new IAction(); error  
  
        System.out.println(IAction.NUM);  
  
        //接口中数据成员默认 public static final, 故而下行编译报错  
        //IAction.NUM = 20;  
    }  
}
```

注意1，定义类使用关键字 `class`，定义接口使用关键字 `interface`

注意2，接口中的数据成员，默认 `public static final` 修饰，是常量，名称全大写（符合命名规范）

注意3，接口中的方法，默认 `public abstract` 修饰，是抽象方法

补充内容（后面章节具体讨论）：

- JDK8中，还允许在接口中编写**静态方法**和**默认方法**
- JDK9中，还允许在接口中编写**私有方法**

2.2 接口实现

Java中类和类之间的关系是继承，且只能是单继承

类和接口是实现关系，通过implements关键字表示，可以是单实现，也可以是多实现

子类还可以在继承一个父类的同时实现多个接口

接口的实现类书写格式：

```
//一个类可以同时实现多个接口
[修饰符] class 类名 implements 接口名1,接口名2,... {
    重写所有抽象方法
}
```

注意事项：

- 接口属于引用数据类型的一种，它不是类，它没有构造方法

- 接口的实现类（子类），可以是正常类（重写所有抽象方法），也可以是抽象类（包含抽象方法）
- 接口不能创建对象，一般用接口引用指向实现类对象

基础案例：

实现类实现单个接口案例。

定义IAction的实现类，重写所有抽象方法，最后进行功能测试。

```
package com.briup.chap07.test;

interface IAction {
    int NUM = 10;

    void start();

    public abstract void end();
}

//定义接口的实现类
class ActionImpl implements IAction {
    @Override
    public void start() {
        System.out.println("start开始执行 ");
    }

    @Override
    public void end() {
        System.out.println("执行完成，end结束");
    }
}

public class Test021_Basic {
    public static void main(String[] args) {
        //1.接口不能实例化对象，下面一行编译报错
    }
}
```

```

        //IAction ic = new IAction();

        //2.接口引用 指向实现类对象
        IAction ac = new ActionImpl();

        //3.接口数据成员访问测试
        System.out.println(IAction.NUM);
        System.out.println(ac.NUM);

        System.out.println("-----");

        //4.通过接口引用 调用 重写方法（多态体现）
        ac.start();
        ac.end();
    }
}

```

注意事项:

在类和接口的实现关系中，**可以使用多态**，因为类和接口的实现关系，可理解为继承的一种形式。

一个类可以同时实现多个接口，但需要把多个接口的抽象方法全部重写（后面案例中具体演示）。

2.3 接口继承

Java中，类和类之间是单继承关系，接口和接口之间是多继承

接口继承格式:

```
[修饰符] interface 子接口 extends 父接口1,父接口2... {  
    //新增成员或抽象方法  
}
```

例如：

```
interface Runnable {  
    void run();  
}  
  
interface Flyable {  
    void fly();  
}  
  
//接口多继承  
interface Action extends Runnable, Flyable {  
    void sing();  
}
```

2.4 综合案例

案例描述：

定义一个抽象父类Animal2，再定义两个接口IJumpAble跳火圈、ICycleAble骑自行车，最后定义一个猴子类Monkey，去继承Animal2，同时实现IJumpAble、ICycleAble，进行功能测试。

复杂实现类定义格式：

```
[修饰符] class 实现类 extends 父类 implements 接口名1, 接口名2, ...  
{  
    重写所有方法;  
}
```


案例描述:

```
package com.briup.chap07.test;

//定义抽象父类
abstract class Animal2 {
    private String color;
    private int age;

    public Animal2() {}
    public Animal2(String color, int age) {
        this.color = color;
        this.age = age;
    }

    // 抽象方法
    public abstract void eat();
    public abstract void sleep();
}

//定义跳火圈接口
interface IJumpAble {
    //默认修饰符为: public abstract
    void jump();
}

//定义骑自行车接口
interface ICycleAble {
    public abstract void cycle();
}

// 定义猴子类: 继承Animal类, 同时实现ICycleAble、IJumpAble接口
class Monkey extends Animal2 implements ICycleAble, IJumpAble {
    private String name;

    //自定义构造方法
```

```

public Monkey() {}

public Monkey(String color, int age, String name) {
    super(color, age);
    this.name = name;
}

//重写从父类继承的方法
@Override
public void eat() {
    System.out.println("猴子" + name + " 喜欢吃大桃子");
}
@Override
public void sleep() {
    System.out.println("猴子" + name + " 喜欢在树上睡觉");
}

// 重写从接口获取的抽象方法
@Override
public void cycle() {
    System.out.println("猴子 " + name + " 能够骑自行车");
}
@Override
public void jump() {
    System.out.println(name + " 能够跳火圈");
}
}

//测试类
public class Test024_Monkey {
    // 基本测试
    public static void main(String[] args) {
        //1.抽象类不能实例化对象，但可以定义引用指向子类对象
        Animal2 a = new Monkey("yellow", 2, "小悟空");

        //2.借助抽象类引用，只能访问抽象类中具有的方法
        a.eat();
        a.sleep();
    }
}

```

```

//下面两行编译报错【多态：编译看左边，运行看右边】
//a.jump();
//a.cycle();

System.out.println("*****");

//3.用接口引用指向实现类对象
ICycleAble c = new Monkey("yellow", 5, "马戏团小猴");

//4.借助接口引用调用接口中重写方法
c.cycle();

//注意：接口引用类型 只能调用 接口中具备的方法【多态：编译看左
边，运行看右边】
//下面三行编译报错
//c.eat(); error
//c.sleep(); error
//c.jump(); error

System.out.println("*****");

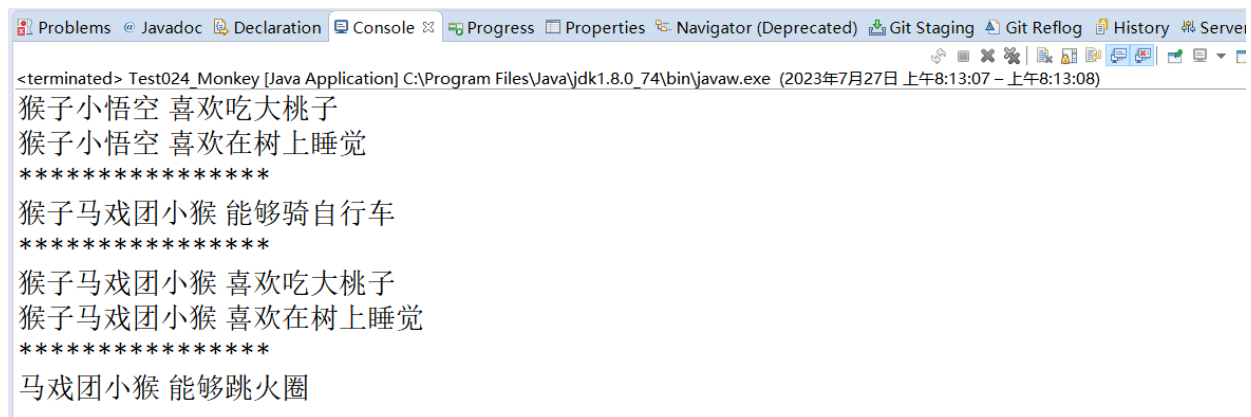
//5.如果想要调用父类方法或其他接口方法，可以借助类型转换实现
//注意，一定要借助instanceof额外判断引用指向对象的类型
if (c instanceof Monkey) {
    Monkey m = (Monkey) c;
    m.eat();
    m.sleep();
}

System.out.println("*****");

if (c instanceof IJumpAble) {
    IJumpAble j = (IJumpAble) c;
    j.jump();
}
}
}

```

运行效果：



```
<terminated> Test024_Monkey [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月27日 上午8:13:07 - 上午8:13:08)
猴子小悟空 喜欢吃大桃子
猴子小悟空 喜欢在树上睡觉
*****
猴子马戏团小猴 能够骑自行车
*****
猴子马戏团小猴 喜欢吃大桃子
猴子马戏团小猴 喜欢在树上睡觉
*****
马戏团小猴 能够跳火圈
```

注意事项：

接口多态应用时，编译看左边，运行看右边

即接口引用只能调用接口中包含的方法，成功调用的是重写以后的方法

2.5 类接口关系

- 类与类的关系

继承关系，只能单继承，但是可以多层继承

- 类与接口的关系

实现关系，可以单实现，也可以多实现，还可以在继承一个类的同时实现多个接口

- 接口与接口的关系

继承关系，可以单继承，也可以多继承

2.6 接口新特性

1) JDK8新特性：接口可以包含静态方法和默认方法

案例如下：

```
package com.briup.chap07.test;

//JDK8中接口 可以添加默认方法和static方法
interface JDK8Action {
    // 接口中静态常量
    String OPS_MODE = "auto";

    // 接口中抽象方法
    void start();
    void stop();

    //下面是JDK8新特性
    //默认方法
    public default void dFun() {
        System.out.println("in default fun() ...");
    }

    //静态方法
    public static void sFun() {
        System.out.println("in static fun() ...");
    }
}

class Demo01 implements JDK8Action {
    @Override
    public void start() {
        System.out.println("重写start() ...");
    }
    @Override
    public void stop() {
        System.out.println("重写stop() ...");
    }
}
```

```

}

public class Test026_JDK8 {
    public static void main(String[] args) {
        //1.接口引用指向实现类对象
        JDK8Action a = new Demo01();

        //2.调用实现类重写方法
        a.start();
        a.stop();

        //3.调用default方法
        a.dFun();

        //4.JDK8中接口可以定义static方法，只能通过接口名调用，但不能通过接口引用调用，也不能通过实现类名调用
        //a.sFun(); 编译报错
        //Demo01.sFun(); 编译报错
        JDK8Action.sFun();
    }
}

```

注意事项：JDK8接口可以定义static方法，只能通过接口名调用

2) JDK9新特性：接口可以包含私有方法

案例如下：

```

package com.briup.chap07.test;

//使用interface关键字来定义接口
interface JDK9Action {
    // 接口中的静态常量
    String OPS_MODE = "auto";

    // 接口中的抽象方法

```

```

    void start();

    //私有方法 jdk9以下报错
    private void run() {
        System.out.println("private run() ...");
    }
}

class Demo02 implements JDK8Action {
    @Override
    public void start() {
        System.out.println("重写start() ...");
    }
}

//测试类
public class Test026_JDK9 {
    public static void main(String[] args) {
        //1.接口引用指向实现类对象
        JDK9Action a = new Demo02();

        //2.调用实现类重写的抽象方法
        a.start();

        //3.调用接口private方法
        a.run();
    }
}

```

注意：测试上述案例需要用合适的JDK版本

2.7 常见面试题

接口和抽象类有什么区别？如何选择？

语法结果区别：

1. 定义方式：抽象类通过使用 `abstract` 关键字来定义，而接口使用 `interface` 关键字来定义
2. 数据成员：抽象类可以包含普通数据成员和 `static` 数据成员，而接口只能包含 `static final` 修饰的数据成员
3. 成员方法：抽象类可以包含具体的方法实现，而接口只能包含抽象方法，即没有方法体的方法声明
4. 构造函数：抽象类可以有构造函数，而接口不能有构造函数
5. 实现方式：一个类可以继承（`extends`）一个抽象类，而一个类可以实现（`implements`）多个接口
6. 多继承：Java不支持多继承，一个类只能继承一个抽象类，但可以实现多个接口

设计理念区别：

- **不同的实现类之间体现 like a 的关系**，接口更加强调行为规范的定义，适用于多个类具有相同行为规范的情况。

例如：飞机具备飞翔的行为，鸟也具备飞翔的行为，此时我们就可以定义接口包含抽象方法 `fly()`，然后让飞机和鸟分别去实现该接口。飞机 like a 鸟，因为它们都会 `fly()`。

- **子类 and 抽象父类体现的是 is a 的关系**，抽象类归根到底还是类，它比较特殊，不能被实例化，只能被继承。抽象类用于定义一种通用的模板或者规范，其中可包含了一些具体数据成员、方法实现和抽象方法声明。

例如：前面案例中的形状类Shape，它里面包含方法getArea()，但该方法功能不确定，所以定义成抽象方法，而包含了抽象方法的类Shape也必须被声明为抽象类。定义子类圆形类，其getArea()方法功能是明确的，则子类中重写方法。

结论：

- 如果仅仅是要额外扩展已存在类的功能，则选择定义接口，让类去实现接口
- 如果需要创建一组相关的类，且这些类之间有一些共同的行为和属性，那么可以定义一个类作为这些类的父类。如果不想实例化父类对象，则可以把这个父类设置为抽象类。

3 内部类

在一个类的内部再定义另外的一个类，这就是内部类。

正常类形式，也称为外部类或顶级类：

```
public class Test{  
  
}  
  
class A{  
  
}
```

内部类形式：

```
//外部类
public class Outer {
    //内部类
    public class Inner {

    }
}
```

内部类一共分为四种形式：

- 成员内部类
- 静态内部类
- 局部内部类
- 匿名内部类

3.1 成员内部类

在类中，除了可以定义成员方法、成员变量，还可以定义成员内部类。

成员内部类定义格式：

```

//外部类
[修饰符] class 外部类 {
    //省略...

    //内部类
    [权限修饰符] class 内部类 {
        0或多个数据成员
        0或多个构造方法
        0或多个成员方法

        //注意，不可以包含static成员或方法
    }
}

```

内部类对象实例化格式：

外部类名.内部类名 对象名 = 外部类对象.内部类对象；

例：Outer.Inner oi = new Outer().new Inner();

内部类中访问外部类中同名成员：

外部类名.this.成员名

例：int v = Outer.this.o_num;

注意事项：

- 成员内部类中可以**直接访问外部类中所有成员和方法（含private）**
- 在外部类中可以直接访问内部类所有的成员和方法（含private），但必须借助内部类对象
- **成员内部类内部不能定义static成员或方法**

案例展示:

定义外部类Outer, 包含数据成员num, 其内部定义内部类Inner, 也包含数据成员num, 测试内部类基本用法。

```
package com.briup.chap07.test;

//1.定义外部类
class Outer {
    private int o_num = 10;
    private int num = 20;

    //2.定义成员内部类
    class Inner {
        private int i_num = 30;
        //关键点: 包含数据成员与外部类中成员同名
        private int num = 40;

        //3.成员内部类 不能定义static 成员或方法
        //public static int s_num = 50;      error
        //public static void s_test() {}     error

        //4.定义内部类方法, 测试对外部类成员的访问
        void show() {
            //内部类中可以直接访问外部类所有成员
            System.out.println("in Inner,o_num: " + o_num);

            //局部变量
            int num = 50;
            System.out.println("in Inner,num: " + num);
        }

        //50
        //访问内部类中数据成员num
        System.out.println("in Inner,num: " + this.num);

        //40
        //访问外部类中同名成员num, 固定格式: 外部类名.this.成员名
        System.out.println("in Inner,num: " +
Outer.this.num); //20
    }
}
```

```

    }

    //5.定义外部类方法，测试访问内部类成员
    // 注意：外部类中可以访问内部类所有成员(含private)，但必须借助内部
    类对象
    public void disp() {
        //实例化内部类对象格式：
        // 外部类.内部类 引用 = 外部类对象.new 内部类(实参);
        Outer.Inner inn = this.new Inner();
        System.out.println("in Outer, i_num: " + inn.i_num);

        //简化格式：
        // 内部类 引用名 = new 内部类(实参列表);
        //解析：
        // 在外部类方法中，Inner类名前默认加'Outer.'，new Inner()前
        默认加'this.'
        Inner inn2 = new Inner();
        System.out.println("inn2.i_num: " + inn2.i_num);
    }
}

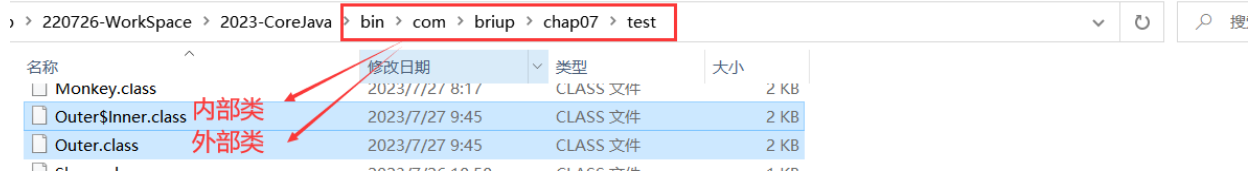
//测试类
public class Test031_Member {
    public static void main(String[] args) {
        //实例化成员内部类对象
        // 外部类.内部类 引用 = 外部类对象.new 内部类(实参);
        Outer out = new Outer();
        Outer.Inner inn = out.new Inner();
        inn.show();

        System.out.println("-----");

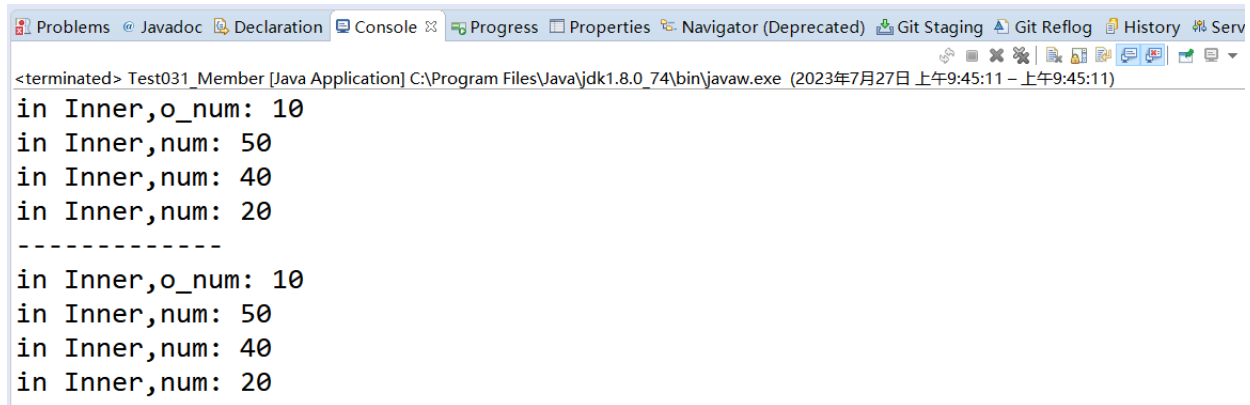
        Outer.Inner inn2 = new Outer().new Inner();
        inn2.show();
    }
}

```

注意，上述代码编译成功后，外部类和内部类都会生成对应的class文件

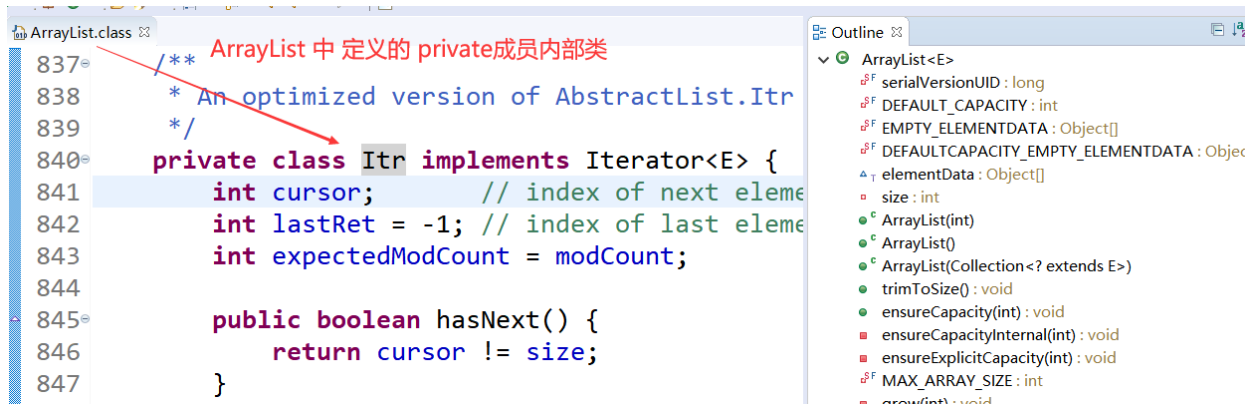


运行效果：



JavaAPI中的使用的成员内部类：

`java.util.ArrayList` 类中，就定义了好几个成员内部类，并且还是 `private` 修饰的



思考，什么情况下会使用内部类？

在对事物进行抽象的时候，若一个事物内部还包含其他事物，就可以考虑使用内部类这种结构。

例如，汽车（Car）中包含发动机（Engine），这时，Engine类就**可以考虑**（非必须）使用内部类来描述，定义在Car类中的成员位置。

这样设计，既可以表示Car和Engine的紧密联系的程度，也可以在Engine类中很方便的使用到Car里面的属性和方法

注意，这是从程序中，类和类之间的关系和意义进行考虑而设计的，其实这里即使不使用内部类的结构，使用普通的两个类也能完成功能，但是内部类的结构设计会更加符合实际意义，也能够好的完成功能，因为内部类访问外部的属性和方法会更加容易。

3.2 静态内部类

静态内部类属于成员内部类中的一种，其需要static关键字进行修饰。

静态内部类定义格式：

```
//外部类
[修饰符] class 外部类 {
    //省略...

    //静态内部类
    [权限修饰符] static class 内部类 {
        0或多个数据成员
        0或多个构造方法
        0或多个成员方法

        //注意：静态内部类成员方法内，只能访问外部类static成员或方法
    }
}
```

对象定义格式:

```
外部类名.内部类名 对象名 = new 外部类名.内部类名();
```

静态方法访问格式:

```
外部类名.内部类名.方法名();
```

如果在外部类中，外部类名可以省略。

注意事项:

- 相对于成员内部类，**静态内部类中可以定义static成员和方法**
- 静态内部类成员方法内，**只能访问外部类static成员或方法【静态只能访问静态】**
- 外部类方法可以访问静态内部类所有成员及方法

案例如下:

```
package com.briup.chap07.test;

class Outer2 {
    private int o_num = 10;
    private static int s_num = 20;

    //定义static内部类
    static class Inner {
        private int i_num = 30;

        //静态内部类中可以定义static成员
        private static int is_num = 40;

        //static成员方法只能访问外部类static成员【静态只能访问静态】
        public static void sFun() {
```



```

        //编译报错
        //System.out.println("Outer2 o_num: " + o_num);

        System.out.println("Outer2 s_num: " + Outer2.s_num);
        System.out.println("Outer2 s_num: " + s_num);
    }

    //普通成员方法
    void show() {
        //1.访问内部类中数据成员
        System.out.println("in Inner, num: " + this.i_num);
//30
        System.out.println("in Inner, is_num: " +
Inner.is_num);    //40

        //2.静态内部类方法中只能访问外部类的static成员
        //编译报错
        //System.out.println("in Inner,o_num: " + o_num);

        //3.访问外部类static成员，下面两行效果相同
        System.out.println("in Inner, s_num: " + s_num);
//20
        System.out.println("in Inner, s_num: " +
Outer2.s_num); //20
    }
}

//外部类方法可以访问 静态内部类所有成员及方法
public void disp() {
    //Outer2.Inner inn = new Inner();
    Outer2.Inner inn = new Outer2.Inner();
    System.out.println("in Outer, i_num: " + inn.i_num);
    System.out.println("in Outer, is_num: " + Inner.is_num);
    System.out.println("in Outer, is_num: " + inn.is_num);
}
}

public class Test032_Static {

```

```

    public static void main(String[] args) {
        //1.静态内部类调用static方法
        //编译报错，不识别Inner，会把Inner当成外部类去查找
        //Inner.sFun();

        //正确写法
        Outer2.Inner.sFun();

        System.out.println("-----");

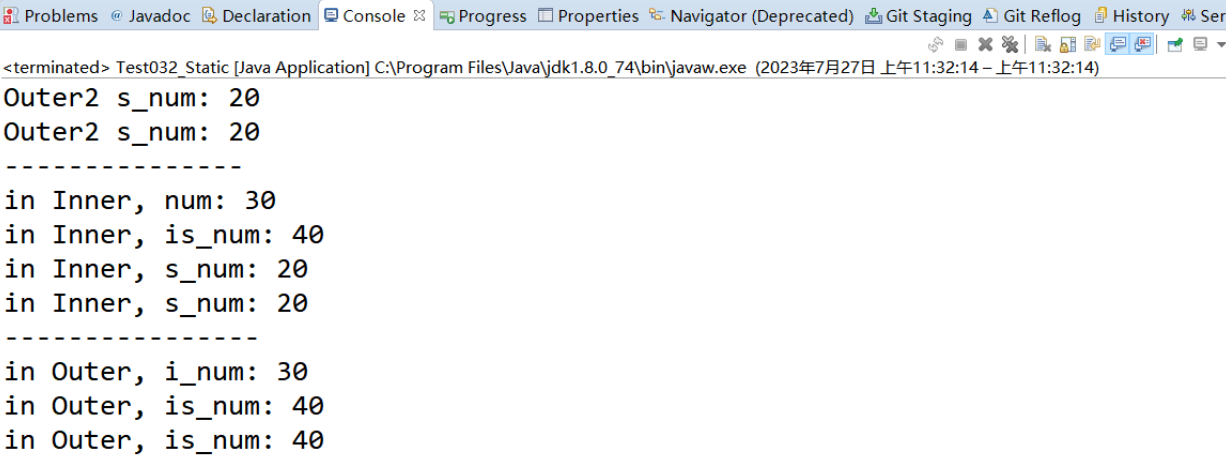
        //2.静态内部类普通方法调用
        //内部类对象 固定定义格式
        Outer2.Inner inn = new Outer2.Inner();
        inn.show();

        System.out.println("-----");

        //3.外部类对象访问内部类成员
        Outer2 outer = new Outer2();
        outer.disp();
    }
}

```

运行效果：



```

<terminated> Test032_Static [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月27日 上午11:32:14 - 上午11:32:14)
Outer2 s_num: 20
Outer2 s_num: 20
-----
in Inner, num: 30
in Inner, is_num: 40
in Inner, s_num: 20
in Inner, s_num: 20
-----
in Outer, i_num: 30
in Outer, is_num: 40
in Outer, is_num: 40

```

注意，在静态内部类中访问不了外部类中的非静态属性和方法

JavaAPI中静态内部类案例:

`java.lang.Integer` 类中, 就定义了一个静态内部类, 并且还是`private`修饰的

```
Integer.class
776  * may be set and saved in the private system properties in the
777  * sun.misc.VM class.
778  */
779 私有 静态 成员内部类
780  private static class IntegerCache {
781      static final int Low = -128;
782      static final int high;
783      static final Integer cache[];
784
785      static {
786          // high value may be configured by property
787          int h = 127;
```

3.3 局部内部类

在外部类的方法中定义的内部类, 我们称为局部内部类, 它的作用范围只是在当前方法中。

局部内部类是最不常用的一种内部类, 大家了解即可。

定义格式:

```
[修饰符] class 外部类 {
    //省略...

    //成员方法
    [修饰符] 返回值类型 方法名(形式参数列表) {
        //功能实现省略...

        //局部内部类定义
        class 成员内部类名 {
            0或多个数据成员
            0或多个构造方法
            0或多个成员方法
        }
    }
}
```

```
        //注意：局部内部类的作用范围在当前方法中，只能在该方法中使用
    }
}
```

注意事项:

局部内部类只能在定义的方法中使用。

案例展示:

```
package com.briup.chap07.test;

//外部类
class Outer3 {
    private int num = 10;

    //包含局部内部类的方法
    public void innerFun() {
        //下面两行效果一样
        //int num = 20;
        final int num = 20;

        //在方法内部，定义局部内部类
        class Inner {
            private int i_num = 30;

            public void test() {
                System.out.println("局部变量num: " + num);
                System.out.println("内部类成员变量this.i_num: " +
this.i_num);
                System.out.println("外部类成员变量Outer3.this.num:
" + Outer3.this.num);
            }
        }
    }
}
```

```

//方法中声明的局部变量，只要在内部类中使用，默认会加上
final修饰

//所以下面一行 编译会报错：给final变量赋值
//num = 60;

    }

}

//创建局部内部类对象
Inner inn = new Inner();
//访问内部类私有成员
System.out.println(inn.i_num);

System.out.println("-----");

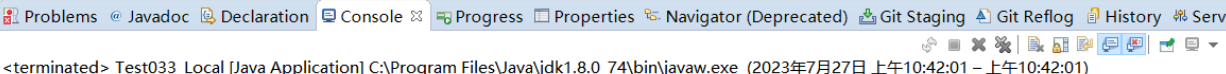
//调用内部方法
inn.test();
}
}

//测试类
public class Test033_Local {
    public static void main(String[] args) {
        Outer3 outer = new Outer3();

        outer.innerFun();
    }
}

```

运行效果：



```

<terminated> Test033_Local [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月27日 上午10:42:01 – 上午10:42:01)
30
-----
局部变量num: 20
成员变量this.num: 30
外部类成员变量Outer3.this.num: 10

```

面试常考：JDK8中，方法中定义的局部变量，如果在局部内部类中对其访问操作，那么这个局部变量会默认加上final修饰，其值不能改变。

3.4 匿名内部类

匿名内部类，是一种没有名字的内部类，**本质上是一个特殊的局部内部类**（定义在方法内部）。

在之后的代码中，匿名内部类是使用最多的内部类（**必须掌握**）。

常规接口、抽象类的操作步骤：

- 声明一个类，去实现这个接口，或去继承抽象父类
- 重写所有抽象方法
- 用接口或引用去指向子类或实现类对象
- 通过接口或抽象类引用去调用重写的方法

在这个过程中，我们的**核心任务是重写抽象方法，最后再调用这些重写的方法**。所以上述过程过于复杂，使用匿名内部类，可以大大简化上述步骤！

匿名内部类书写格式：

```
父类或接口类型 变量名 = new 父类或接口(构造方法实参列表) {  
    // 重写所有的抽象方法  
    @Override  
    public 返回值类型 method1(形参列表) {  
        方法体实现  
    }  
  
    @Override  
    public 返回值类型 method2(形参列表) {  
        方法体实现  
    }  
}
```

```
//省略...  
};  
  
//匿名内部类对象调用方法  
变量名.重写方法(实参列表);
```

匿名内部类的俩种形式:

- 利用父类，进行声明并创建匿名内部类对象，这个匿名内部类默认就是这个父类的子类型
- 利用接口，进行声明并创建匿名内部类对象，这个匿名内部类默认就是这个接口的实现类

匿名内部类注意事项:

- 匿名内部类**必须依托于一个接口或一个父类（可以是抽象类，也可以是普通类）**
- 匿名内部类在声明的同时，就必须创建出对象，否则后面就没法创建了
- 匿名内部类中无法定义构造器

1) 匿名内部类实现接口案例

定义ISleep接口，然后通过匿名内部类对象调用方法。

```
package com.briup.chap07.test;  
  
//定义接口  
interface ISleep {  
    void sleep();  
}  
  
// 匿名内部类基础测试
```

```

public class Test034_Interface {
    public static void main(String[] args) {
        //1.正常写法
        ISleep s1 = new ISleep() {
            @Override
            public void sleep() {
                System.out.println("躺着睡");
            }
        };
        s1.sleep();

        System.out.println("-----");

        //2.简化写法
        ISleep s2 = new ISleep() {
            @Override
            public void sleep() {
                System.out.println("趴着睡");
            }
        };
        s2.sleep();

        System.out.println("-----");

        //3.进一步简化写法
        // 匿名内部类对象直接调用抽象方法
        new ISleep() {
            @Override
            public void sleep() {
                System.out.println("水里面睡");
            }
        }.sleep();
    }
}

```

//运行效果:

躺着睡

趴着睡

水里面睡

2) 匿名内部类实现抽象类案例

```
package com.briup.chap07.test;

//定义抽象类
abstract class MyThread {
    //抽象方法
    public abstract void run();

    //普通方法
    public void test() {
        System.out.println("in Mythread,test ...");
    }
}

public class Test034_Abstract {
    public static void main(String[] args) {
        //1.普通写法
        MyThread th = new MyThread() {
            @Override
            public void run() {
                System.out.println("重写 run1");
            }
        };

        th.run();
        th.test();

        System.out.println("-----");

        //2.简化写法
        new MyThread() {
```

```

        public void run() {
            System.out.println("in run2 ...");
        }
    }.run();

    //注意，匿名对象只能使用一次，因为没有名字，无法再次访问
}
}

//运行效果：
重写 run1
in Mythread,test ...
-----
in run2 ...

```

3) 匿名内部类对象使用父类构造方法案例

```

package com.briup.chap07.test;

abstract class Animal3 {
    private String name;

    public Animal3() {}
    public Animal3(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    //抽象方法
    public abstract void eat();
    public abstract void sleep();
}

```

```

public class Test034_Constructor {
    public static void main(String[] args) {
        //实例化匿名内部类对象
        Animal3 a = new Animal3() {
            //必须重写所有的抽象方法
            @Override
            public void eat() {
                //注意：匿名内部类属于子类，在重写方法中可通过super关键字访问父类方法或成员
                System.out.println(super.getName() + " 喜欢吃鱼");
            }

            @Override
            public void sleep() {
                System.out.println(super.getName() + " 睡觉");
            }
        };

        //父类引用调用重写方法
        a.sleep();
        a.eat();

        System.out.println("-----");

        //实例化子类对象，调用父类构造器对父类进行初始化
        new Animal3("小汤姆") {
            //必须重写所有的抽象方法
            @Override
            public void eat() {
                //注意：匿名内部类属于子类，在重写方法中可通过super关键字访问父类方法或成员
                System.out.println(super.getName() + " 喜欢抓老鼠吃");
            }

            @Override
            public void sleep() {
                System.out.println(super.getName() + " 睡觉");
            }
        };
    }
}

```

```
    }  
    }.eat();  
}  
}
```

//运行效果:

null 睡觉

null 喜欢吃鱼

小汤姆 喜欢抓老鼠吃

内部类应用场景选择:

- 考虑这个内部类，如果需要反复的进行多次使用（必须有名字）
在这个内部类中，如果需要定义静态的属性和方法，选择使用**静态内部类**
在这个内部类中，如果需要访问外部类的非静态属性和方法，选择使用**成员内部类**
- 考虑这个内部类，如果只需要使用一次（可以没有名字）
选择使用**匿名内部类**
- **局部内部类**，几乎不会使用

4 包装类

Java中有八种基本数据类型，它们只能表示一些最简单的数字，这些数字最小的在内存中占1个字节8位，最大占8个字节64位。这些都是简单的数字，不是对象，所以不能用来调用方法或者属性，功能不够强大。

4.1 概述

针对这八种基本类型，JavaAPI又专门提供了对应的类类型，目的就是为分别把这八种基本类型的数据，包装成对应的类类型，这时候就变成对象了，就可以调用方法了或者访问属性了。

基本类型	包装类型
boolean	java.lang.Boolean
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character

4.2 基础案例

在这些包装类型中，都定义相关的属性和方法，例如Integer中：

```
package com.briup.chap07.test;

public class Test042_Integer {
    public static void main(String[] args) {
        int i = 1;
        //编译报错，因为i不是对象
        //i.toString();
    }
}
```

```

//1.int --> Integer
//Integer o = new Integer(i);
Integer o = Integer.valueOf(1);

//2.Integer --> int
int j = o.intValue();

System.out.println(o.toString());

//3.包装类静态成员访问
System.out.println(Integer.MIN_VALUE);
System.out.println(Integer.MAX_VALUE);

//4.包装类静态方法调用
//把100分别转为十进制、二进制、八进制、十六进制的字符串形式
System.out.println(Integer.toString(100));
System.out.println(Integer.toString(100,2));
System.out.println(Integer.toString(100,8));
System.out.println(Integer.toString(100,16));

//把100转为二进制形式
System.out.println(Integer.toBinaryString(100));

//把字符串"100"转为int类型的100
int num = Integer.parseInt("100");
System.out.println(num);
}
}

```

注意：一定要掌握 `public static int parseInt(String s)` 方法。

4.3 装箱拆箱

JDK1.5或以上，可以支持基本类型和包装类型之间的自动装箱、自动拆箱，这简化了基本类型和包装类型之间的转换。

- **自动装箱：**基本数据类型值 自动转化为 包装类对象
- **自动拆箱：**包装类对象 自动转化为 基本数据类型值

案例展示：

int、Integer自动装箱拆箱展示。

```
package com.briup.chap07.test;

public class Test043_Integer {
    public static void main(String[] args) {
        //JDK1.5 之前
        //Integer o = new Integer(1);
        //Integer o = Integer.valueOf(1);

        //JDK1.5 之后
        //自动装箱，这里会自动把数字1包装成Integer类型的对象
        Integer o = 1;

        //JDK1.5 之前
        //Integer o = new Integer(1);
        //int i = o.intValue();

        //JDK1.5 之后
        //Integer o = new Integer(1);
        //自动拆箱，这里会自动把对象o拆箱为一个int类型的数字，并把数字赋值给int类型的变量i
        int i = o;
    }
}
```

其他的基本类型和包装类型之间的转换，与此类似

注意事项:

使用基本类型和包装类时，要考虑隐式类型转换。

不同类型的基本数据类型和包装类，是不可以自动装箱拆箱的，例如int和Long。

具体案例如下：

```
package com.briup.chap07.test;

public class Test043_Test {
    public void test1(int i) {
    }

    public void test2(Integer i) {
    }

    public void test3(long i) {
    }

    public void test4(Long i) {
    }

    public static void main(String[] args) {
        Test043_Test t=new Test043_Test();
        t.test1(1);// 编译通过 int i = 1; 正常赋值
        t.test2(1);// 编译通过 Integer i = 1; 自动装箱
        t.test3(1);// 编译通过 long i = 1; 隐式类型转换

        // 编译报错
        // 错误的代码: Long i = 1;
        // int和Long 之间没有任何关系
        //t.test4(1);

        t.test4(1L);// 编译通过 Long i = 1L; 自动装箱
    }
}
```


4.4 Integer缓冲区

常见面试题：分析下面程序的输出结果。

```
public static void main(String[] args) {  
    Integer i1 = new Integer(127);  
    Integer i2 = new Integer(127);  
    System.out.println("i1 == i2: " + (i1 == i2));    //?  
  
    Integer i3 = 127;  
    Integer i4 = 127;  
    System.out.println("i3 == i4: " + (i3 == i4));    //?  
  
    Integer i5 = 128;  
    Integer i6 = 128;  
    System.out.println("i5 == i6: " + (i5 == i6));    //?  
}
```

//程序的输出结果为：

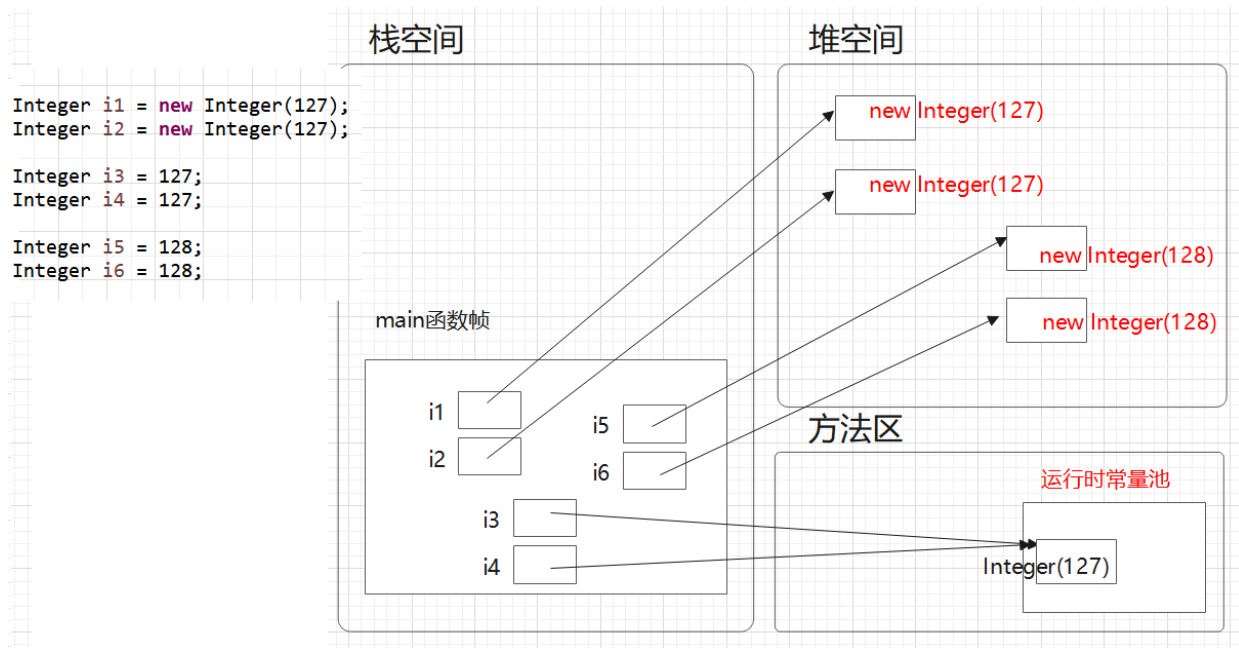
```
i1 == i2: true  
i3 == i4: false  
i5 == i6: false
```

在Java中**方法区**有一个叫做**运行时常量池（Runtime Constant Pool）**的区域，用于存储编译期间生成的各种字面量和符号引用，**Integer常量池**就存储在该区域。

Integer常量池是一个特殊的缓存机制，用于存储在范围 **[-128, 127]** 之间的Integer常量对象。这个范围是Java中的一个固定范围，超出这个范围的整型常量不会被缓存。

当使用**自动装箱（Autoboxing）**将一个整数赋值给一个Integer对象时，如果该整数在 **[-128, 127]** 范围内，那么会从Integer常量池中获取一个已经存在的对象，而不是创建一个新的对象。这是因为在这个范围内的整数常见且频繁使用，**通过复用对象可以节省内存空间和提高性能。**

Integer对象内存构成：



结果分析：

`i1 == i2` 结果为false，图上很清楚，无需多解释。

`i3 == i4` 结果为true，因为127在[-128, 127]这个范围中，i3和i4直接复用了常量池中Integer(127)对象，故而两者哈希码值相同，== 比较也相同。

`i5 == i6` 结果为false，因为128不在[-128, 127]这个范围中，i5和i6自动装箱过程中，系统底层执行了 `Integer i5 = new Integer(128);` `Integer i6 = new Integer(128);`，故而i5和i6指向2个不同的对象。

Integer源码分析：

```
package java.lang;
```

```

public final class Integer extends Number implements
Comparable<Integer> {
    //省略...

    /**
     * Cache to support the object identity semantics of
     * autoboxing for values between 128 and 127 (inclusive) as required
     * by JLS.
     */
    private static class IntegerCache {
        static final int low = -128;
        static final int high;
        static final Integer cache[];

        static {
            // high value may be configured by property
            int h = 127;

            // 省略...

            high = h;

            cache = new Integer[(high - low) + 1];
            int j = low;
            for(int k = 0; k < cache.length; k++)
                cache[k] = new Integer(j++);

            // range [-128, 127] must be interned (JLS7 5.1.7)
            assert IntegerCache.high >= 127;
        }

        private IntegerCache() {}
    }

    //省略...
}

```

补充内容:

`System` 类方法:

```
public static native int identityHashCode(Object x);
```

该方法会返回对象的哈希码，即Java根据对象在内存中的地址计算出来一个整数值，**结论：相同地址的对象调用该方法结果相同，不同地址对象结果不同！**。

注意，哈希码并不是对象的内存地址。

验证案例:

```
package com.briup.chap07.test;

public class Test044_Cache {
    public static void main(String[] args) {
        Integer i1 = new Integer(127);
        Integer i2 = new Integer(127);
        System.out.println("i1: " + System.identityHashCode(i1));
        System.out.println("i2: " + System.identityHashCode(i2));
        System.out.println("i1 == i2: " + (i1 == i2));    //false

        System.out.println("-----");

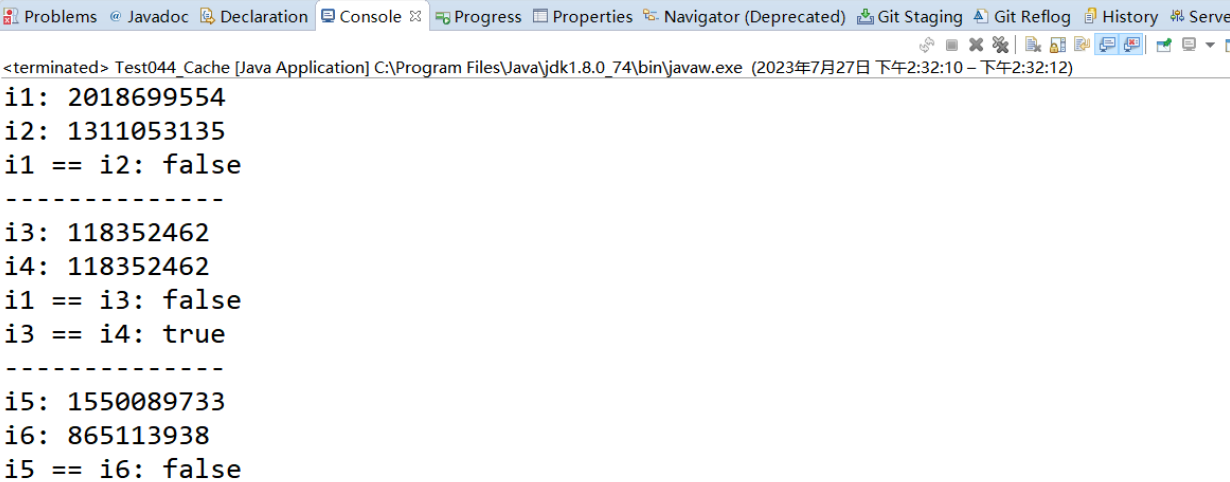
        Integer i3 = 127;
        Integer i4 = 127;
        System.out.println("i3: " + System.identityHashCode(i3));
        System.out.println("i4: " + System.identityHashCode(i4));
        System.out.println("i1 == i3: " + (i1 == i3));    //false
        System.out.println("i3 == i4: " + (i3 == i4));    //true

        System.out.println("-----");

        Integer i5 = 128;
        Integer i6 = 128;
```

```
        System.out.println("i5: " + System.identityHashCode(i5));  
        System.out.println("i6: " + System.identityHashCode(i6));  
        System.out.println("i5 == i6: " + (i5 == i6));    //false  
    }  
}
```

运行效果：



The screenshot shows an IDE console window with the following output:

```
<terminated> Test044_Cache [Java Application] C:\Program Files\Java\jdk1.8.0_74\bin\javaw.exe (2023年7月27日 下午2:32:10 - 下午2:32:12)  
i1: 2018699554  
i2: 1311053135  
i1 == i2: false  
-----  
i3: 118352462  
i4: 118352462  
i1 == i3: false  
i3 == i4: true  
-----  
i5: 1550089733  
i6: 865113938  
i5 == i6: false
```

5 Object类

Object类是所有类的父类型，类中定义的方法，java中所有对象都可以调用

5.1 toString

该方法前面课程已经讲过，其可以返回一个对象默认的字符串形式。

```

package java.lang;

/**
 * Class {@code Object} is the root of the class hierarchy.
 * Every class has {@code Object} as a superclass. All objects,
 */
public class Object{
    //省略...

    public String toString() {
        return getClass().getName() + "@" +
Integer.toHexString(hashCode());
    }
}

```

子类中可以对该方法进行重写:

```

public class Student{
    private String name;
    private int age;

    //重写方法
    @Override
    public String toString() {
        return "Student[name="+name+", age="+age+"]";
    }
}

//注意，输出对象时，自动调用对象.toString()方法
Student s1 = new Student();
//下面两行效果相同
System.out.println(s1);
System.out.println(s1.toString());

```

5.2 equals

该方法可以比较两个对象是否相等（**重点掌握**）

源码如下：

```
package java.lang;

public class Object {
    /**
     * Indicates whether some other object is "equal to" this
     one.
     * 默认比较的是两个对象堆内存地址值，实际开发中，我们往往需要重写该方法，比较对象属性值。
     */
    public boolean equals(Object obj) {
        return (this == obj);
    }
}
```

可以看出，Object中的equals方法，是直接使用的==号进行的比较，比较两个对象的地址值是否相等

案例1：

定义Stu类，包含name和age属性，创建两个对象进行equals比较。

```
package com.briup.chap07.test;

class Stu {
    private String name;
    private int age;

    public Stu() {}
    public Stu(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
}
```

```

    }

    //get|set省略...
}

public class Test053_Equals {
    public static void main(String[] args) {
        Stu s1 = new Stu("tom", 20);
        Stu s2 = new Stu("tom", 20);

        System.out.println("s1==s2: " + (s1 == s2));    //false
        System.out.println(s1.equals(s2));              //false
    }
}

//运行效果:
s1==s2: false
false

```

结果分析:

`s1==s2`结果为`false`，`==`比较的是对象的引用，即比较两个对象是否指向内存中的同一个地址。`s1`和`s2`是两个不同对象，不指向同一个地址，故而结果`false`。

`s1.equals(s2)`结果为`false`，`Stu`没有重写`equals`方法，默认使用`==`比较`s1`和`s2`，结果`false`。

案例2:

在上述案例的基础上，重写`Stu`类中的`equals`方法，比较对象的属性值是否相同，然后运行测试。

```

package com.briup.chap07.test;

public class Test053_Equals {
    public static void main(String[] args) {

```



```

        Stu s1 = new Stu("tom", 20);
        Stu s2 = new Stu("tom", 20);

        System.out.println("s1==s2: " + (s1 == s2));    //false
        System.out.println(s1.equals(s2));              //false
    }
}

class Stu {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    //重写equals方法
    @Override
    public boolean equals(Object obj) {
        boolean flag = false;

        //1.obj类型判断，如果不是Stu则返回false

```

```

        if(!(obj instanceof Stu))
            return false;

        //向下转型
        Stu s = (Stu)obj;

        //2.如果比较的两个对象 指向同一块内存（同一个对象）
        if(this == obj)
            return true;

        //3.属性值比较，如果姓名年龄全都相同，则修改返回标志为true
        if(name.equals(s.getName())
            && (age == s.getAge()))
            flag = true;

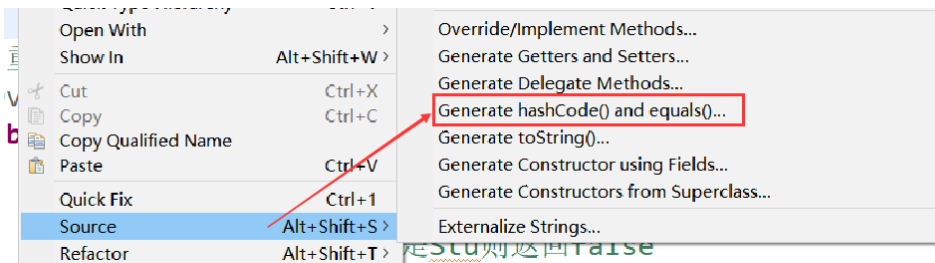
        return flag;
    }
}

//运行效果：
s1==s2: false
true

```

注意：一般情况下，程序员不需要手动重写equals方法，STS中提供了自动生成（建议）的快捷键。

Alt+Shift+S，然后选择操作即可：



自动生成代码如下：

```

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Stu other = (Stu) obj;
    if (age != other.age)
        return false;
    if (name == null) {
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name))
        return false;
    return true;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + age;
    result = prime * result + ((name == null) ? 0 :
name.hashCode());
    return result;
}

```

5.3 hashCode

该方法的具体细节，我们在集合章节再补充，现阶段大家大致了解即可！

该方法返回一个int值，该int值是JVM根据对象在内存的中的特征（地址值），通过**哈希算法**计算出的一个结果。

Hash，一般翻译做“散列”，也可以音译为“哈希”，就是把任意长度的数据输入，通过散列算法，变换成固定长度的输出，该输出就是散列值。

一个任意长度的输入转为一个固定长度的输出，是一种压缩映射，也就是说，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。

注意：通常情况下我们认为Object中hashCode方法返回的是对象的内存地址值，但实际上并不是。

hashCode源码：

```
package java.lang;

public class Object {
    /**
     * Returns a hash code value for the object. This method is
     * supported for the benefit of hash tables such as those
     * provided by
     * {@link java.util.HashMap}.
     */
    public native int hashCode();
}
```

引用变量的hashCode值：

- 两个引用变量指向同一个对象，则它们的hashCode值一定相等
- 两个引用变量的hashCode值相同，则它们有可能指向同一个对象，也可能指向不同对象
- 两个引用变量的hashCode值不同，则它们肯定不可能指向同一个对象

在自定义类中，如果需要重写equals方法的话，我们一般会同时重写hashCode()，建议使用STS自动生成重写代码（equals章节已经讲过）。

5.4 getClass

该方法的具体细节，我们在反射章节再补充，现阶段大家大致了解即可！

该方法是非常重要的一种方法，它返回引用变量在运行时所指向的字节码对象。

该方法是native修饰的本地方法，不是Java语言实现的。

源码如下：

```
package java.lang;

public class Object {
    //省略...

    /**
     * 获取类的字节码对象（反射部分会专门学习）
     * Returns the runtime class of this {@code Object}. The
     returned
     * {@code Class} object is the object that is locked by
     {@code
     * static synchronized} methods of the represented class.
     */
    public final native Class<?> getClass();
}
```

注意：子类中不能重写getClass，调用的一定是Object中的getClass方法。

```
package com.briup.chap07.test;

public class Test052_GetClass {
    public static void main(String[] args) {
        Test052_GetClass tg = new Test052_GetClass();
        System.out.println(tg.getClass());
    }
}

//运行结果
class com.briup.chap07.test.Test052_GetClass
```

6 String类

字符串String，是程序中使用最多的一种数据，JVM在内存中专门设置了一块区域（字符串常量池），来提高字符串对象的使用效率。

6.1 概述

在Java中，`String`是一个类，用于表示字符串，它是Java中最常用的类之一，用于处理文本数据。

String基础内容：

- String 类在 java.lang 包下，所以使用的时候不需要导包
- Java 程序中所有字符串字面值（如"abc"）都是 String类对象
- 字符串值不可变，`String`对象是不可变的，一旦创建，它们的值就不能被修改

常用构造方法：

方法名	说明
<code>public String()</code>	创建一个空白字符串对象，不含有任何内容
<code>public String(char[] chs)</code>	根据字符数组的内容，来创建字符串对象
<code>public String(String original)</code>	根据传入的字符串内容，来创建字符串对象
<code>String s = "abc" ;</code>	直接赋值的方式创建字符串对象，内容就是abc

案例展示：

```
package com.briup.chap07.test;

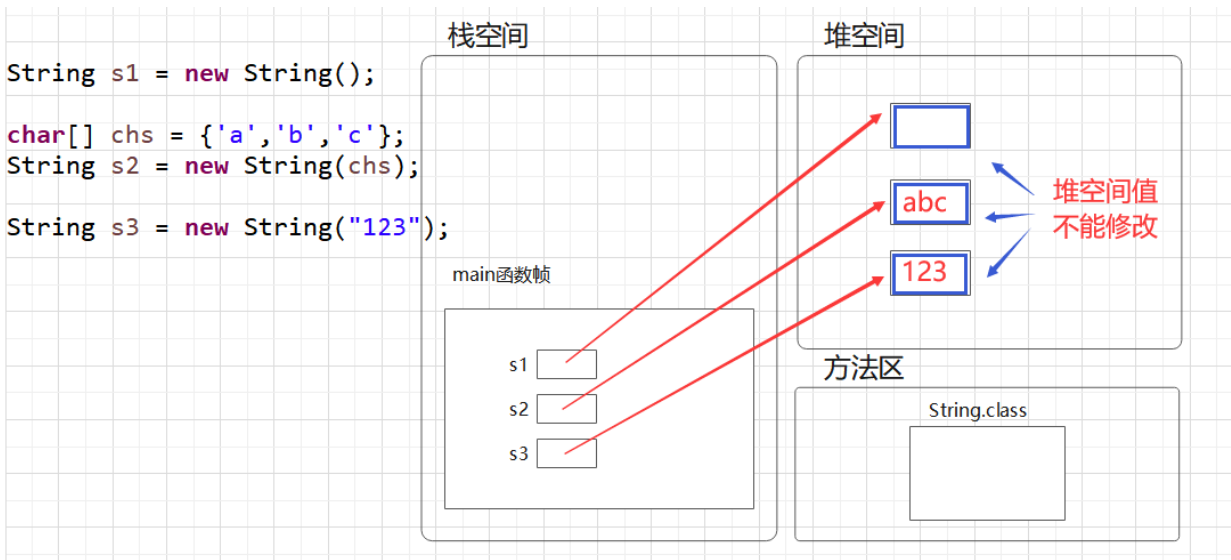
public class Test061_Basic {
    public static void main(String[] args) {
        // public String() : 创建一个空白字符串对象，不含有任何内容
        String s1 = new String();
        System.out.println(s1);

        // public String(char[] chs) : 根据字符数组的内容，来创建字符串对象
        char[] chs = {'a', 'b', 'c'};
        String s2 = new String(chs);
        System.out.println(s2);

        // public String(String original) : 根据传入的字符串内容，来创建字符串对象
        String s3 = new String("123");
        System.out.println(s3);

        String s4 = "hello";
        System.out.println(s4);
    }
}
```

注意，字符串对象创建以后，堆空间中字符串值不可以被修改，具体如下图：



6.2 常量池

问题引入：

创建字符串对象，和其他普通对象一样，会占用计算机的资源（时间和空间），作为最常用的数据类型，大量频繁的创建字符串对象，会极大程度地影响程序的性能。

JVM为了提高性能和减少内存开销，在实例化字符串常量时进行了一些优化

- 为字符串开辟一个字符串常量池，类似于缓存区
- 创建字符串常量时，首先会检查字符串常量池中是否存在该字符串，如果存在该字符串，则返回该实例的引用，如果不存在，则实例化创建该字符串，并放入池中

String常量池：

在Java中，String常量池是一块特殊的内存区域，用于存储字符串常量。String常量池的设计目的是为了节省内存和提高性能。

当我们创建字符串常量时，如果字符串常量池中已经存在相同内容的字符串，那么新创建的字符串常量会直接引用已存在的字符串对象，而不会创建新的对象。这样可以避免重复创建相同内容的字符串，节省内存空间。

在JDK8及之后的版本中，**字符串常量池的位置与其他对象的存储位置，都位于堆内存中**。这样做的好处是，字符串常量池的大小可以根据需要进行调整，并且可以享受到垃圾回收器对堆内存的优化。

Java将字符串放入String常量池的方式：

1. **直接赋值**：通过直接赋值方式创建的字符串常量会被放入常量池中。

例如：`String str = "Hello";`

2. **调用String类提供intern()方法**：可以将字符串对象放入常量池中，并返回常量池中的引用。

例如：`String str = new String("World").intern();`

注意：**通过new关键字创建的字符串对象不会放入常量池中**，而是在堆内存中创建一个新的对象。只有通过直接赋值或调用intern()方法才能将字符串放入常量池中。

案例1：

```
package com.briup.chap07.test;

public class Test062_String {
    public static void main(String[] args) {
        String s1 = "Hello"; // 字符串常量，放入常量池
        String s2 = "Hello"; // 直接引用常量池中的字符串对象
        System.out.println(s1 == s2); // true，引用相同

        // 直接new String对象，不会将'World'放入常量池
        String s3 = new String("World");

        // 调用intern()方法，将'World'放入常量池，并返回常量池中的引用
    }
}
```

```

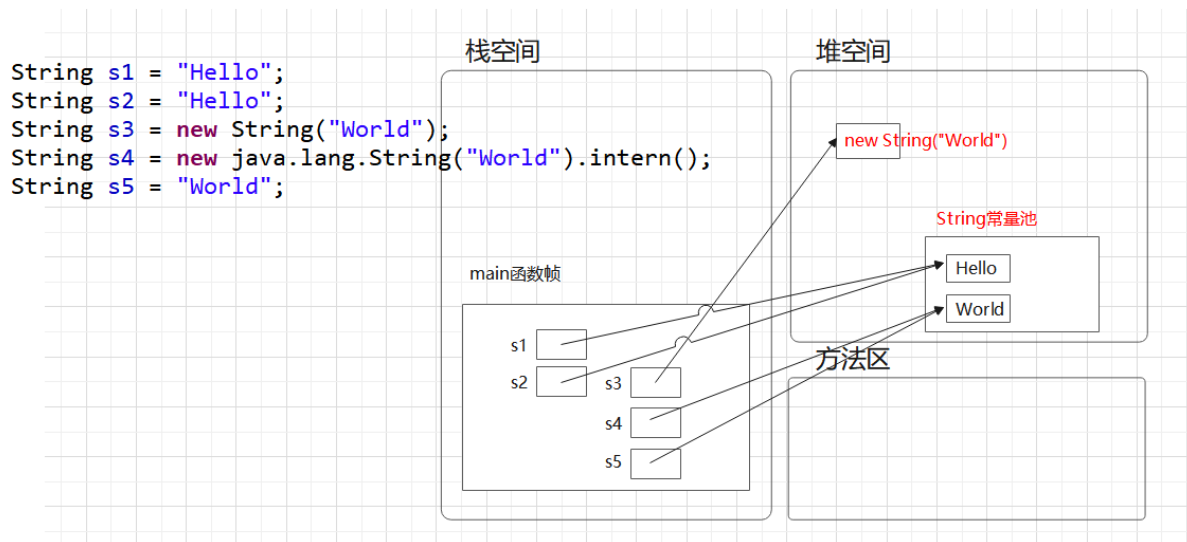
String s4 = new java.lang.String("World").intern();

String s5 = "World";

System.out.println(s3 == s4); // false, 引用不同
System.out.println(s4 == s5); // true, 引用相同
}
}

```

对应内存图为：



案例2:

```

package com.briup.chap07.test;

public class Test062_String2 {
    public static void main(String[] args) {
        String s1 = "a";
        String s2 = "b";

        // 常量优化机制: "a" 和 "b"都是字面值常量, 借助 + 连接, 其结果
        // "ab" 也被当作常量
        String s3 = "a" + "b";
        String s4 = "ab";

        System.out.println(s3.equals(s4)); // true
        System.out.println(s3 == s4);      // true
    }
}

```

```

        System.out.println("-----");

        String s5 = s1 + s2;
        System.out.println(s4.equals(s5)); // true
        System.out.println(s4 == s5);      // false

        System.out.println("-----");

        String s6 = (s1 + s2).intern();
        System.out.println(s4.equals(s6)); // true
        System.out.println(s4 == s6);      // true
    }
}

```

注意事项:

使用 + 拼接多个字符串常量，拼接的结果仍旧是字符串常量

如果结果字符串常量在常量池中不存在，则Java会将其放入到字符串常量池中

```

//final常量测试
public static void main(String[] args) {
    String str = "ab";

    //final修饰的String常量
    final String str1 = "a";
    String str2 = str1 + "b";

    System.out.println(str == str2);
}

//结果为true

```

6.3 常见方法

String源码:

```
package java.lang;

public final class String
    implements java.io.Serializable, Comparable<String>,
    CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    //省略...

    //获取字符串字符个数
    public int length();

    //比较字符串的内容，严格区分大小写
    public boolean equals(Object anObject);

    //返回指定索引处的 char 值
    public char charAt(int index);

    //将字符串拆分为字符数组后返回
    public char[] toCharArray();

    //根据传入的规则切割字符串，得到字符串数组
    public String[] split(String regex);

    //根据开始和结束索引进行截取，得到新的字符串 [begin,end)
    public String substring(int begin, int end);

    //从传入的索引处截取，截取到末尾，得到新的字符串
    [begin,str.length())
    public String substring(int begin);

    //使用新值，将字符串中的旧值替换，得到新的字符串
```

```
    public String replace(CharSequence target, CharSequence  
replacement);  
}
```

基础案例1:

从键盘录入一行字符串，获取其长度和索引2位置上的字符，最后统计字符串中数字字符的个数。

```
package com.briup.chap07.test;  
  
import java.util.Scanner;  
  
public class Test063_Function {  
    public static void main(String[] args) {  
        //1.实例化sc对象，获取从键盘录入整行字符串  
        Scanner sc = new Scanner(System.in);  
        System.out.println("input string: ");  
        String str = sc.nextLine();  
  
        //2.获取字符串长度 及 索引2上的字符  
        System.out.println("length: " + str.length());  
        System.out.println("charAt(2): " + str.charAt(2));  
  
        //3.统计字符串中数字字符的个数  
        char[] array = str.toCharArray();  
        int count = 0;  
        for (char c : array) {  
            if(c > '0' && c < '9') {  
                count++;  
            }  
        }  
        System.out.println("数字字符个数: " + count);  
    }  
}
```

登录案例2:

假设用户名和密码是"admin"和"briup", 从键盘录入用户名和密码, 如果匹配成功则输出"登录成功", 否则输出"登录失败, 用户名或密码有误! "

```
public static void main01(String[] args) {  
    //1.实例化sc对象  
    Scanner sc = new Scanner(System.in);  
  
    //2.从键盘分别录入用户名和密码  
    System.out.println("input username: ");  
    String username = sc.nextLine();  
    System.out.println("input password: ");  
    String password = sc.nextLine();  
  
    //3.登录校验  
    //注意: 一般 字符串常量值.equals方法, 防止空指针异常  
    if("admin".equals(username)  
        && "briup".equals(password)) {  
        System.out.println("登录成功! ");  
    }else {  
        System.out.println("登录失败, 用户名或密码错误! ");  
    }  
}
```

字符串拆分案例3:

从键盘录入学生的信息, 格式为: 学号:姓名:分数, 例: 001:zs:78, 请拆分学生信息并输出。

```
public static void main(String[] args) {  
    //1.实例化sc对象, 获取从键盘录入学生信息  
    Scanner sc = new Scanner(System.in);  
    System.out.println("input student msg: ");  
    // 001:zs:78  
    String str = sc.nextLine();
```

```

//2.按照:进行分割
String[] arr = str.split(":");

//3.输出学生基本信息
System.out.println("id: " + arr[0]);
System.out.println("name: " + arr[1]);

//注意, 分数为int类型, 需要String --> int
int score = Integer.parseInt(arr[2]);
System.out.println("score: " + score);
}

```

字符串拆分案例4:

从键盘录入学生的信息, 格式为: 学号.姓名.分数, 例: 001.zs.78, 请拆分学生信息并输出。

注意: 相对上面案例, 当前案例中分隔符为 . 属于特殊符号, 使用时需去除其特殊含义。

```

public static void main(String[] args) {
    //1.实例化sc对象, 获取从键盘录入学生字符串
    Scanner sc = new Scanner(System.in);
    System.out.println("input student msg: ");
    // 001.zs.78
    String str = sc.nextLine();

    //2.字符串拆分
    //特殊含义的字符, 作为分隔符, 需要去除其特殊含义, 下面两种方式
    //String[] arr = str.split("\\.");
    String[] arr = str.split("[.]");
    System.out.println("id: " + arr[0]);
    System.out.println("name: " + arr[1]);

    int score = Integer.parseInt(arr[2]);
    System.out.println("score: " + score);
}

```

字符串截取案例5:

从键盘录入一个手机号，将中间四位号码修改为 **** 输出，如键盘录入 13800001234，处理后最终效果为： 138****1234。

```
public static void main(String[] args) {  
    //1.实例化sc对象，获取手机字符串  
    Scanner sc = new Scanner(System.in);  
    System.out.println("input tel: ");  
    String tel = sc.nextLine();  
  
    //2.截取字符串前三位  
    String start = tel.substring(0,3);  
  
    //3.截取字符串后四位  
    String end = tel.substring(7);  
  
    //4.将截取后的两个字符串，中间加上****进行拼接，输出结果  
    System.out.println(start + "****" + end);  
}
```

字符串替换案例6:

用字符串替换replace方法，重新实现上述功能。

```
public static void main(String[] args) {  
    //1.实例化sc对象，获取手机字符串  
    Scanner sc = new Scanner(System.in);  
    System.out.println("input tel: ");  
    String tel = sc.nextLine();  
  
    //2.截取字符串中间4位  
    String mid = tel.substring(3,3+4);
```



```
//3.用****替换中间4位，得到新tel
String newTel = tel.replace(mid, "****");

//4.输出新tel
System.out.println("newTel: " + newTel);
}
```

其他String相关方法，自行参考API学习使用。

7 枚举

7.1 枚举类概述

枚举，是JDK1.5引入的新特性，可以通过关键字 `enum` 来定义枚举类。

枚举类意义

1) Java中的类，从语法上来说，可以创建无数个对象

例如，`学生类 Student`，我们可以创建出10个、20个或更多数量的学生对象，并且从实际业务上讲也没有问题，因为实际情况中确实会存在很多不同的学生。

2) Java特殊类，其所能创建的对象个数是固定的

例如，`性别类 Gender`，表示人的性别，从语法上来说，可以创建出无数个性别对象，但是从实际意义上看，我们只需要创建2个对象就可以了：性别男、性别女。

3) 如何实现上述Java特殊类

我们可以将 `Gender` 定义为一个 `枚举类型 (enum)`，在枚举类型中，我们需要提前定义 `枚举元素`（枚举类型对象固定的几个取值），以后开发过程中只能使用枚举元素给 `Gender` 类对象赋值。

例如：

```
package com.briup.chap07.test;

//1.定义枚举类
enum Gender {
    MALE, FEMALE
}

public class Test071_EnumBasic {
    public static void main(String[] args) {
        //2.枚举类实例化对象
        // 固定格式：枚举类 引用名 = 枚举类.枚举元素值；
        Gender g1 = Gender.MALE;
        Gender g2 = Gender.MALE;

        System.out.println(g1);
        System.out.println(g2);

        System.out.println("-----");

        Gender g3 = Gender.FEMALE;
        Gender g4 = Gender.FEMALE;

        System.out.println(g3);
        System.out.println(g4);

        //3.错误用法：枚举对象的取值，只能是枚举元素
        //Gender g5 = Gender.BOY; error
    }
}

//输出枚举对象，默认输出枚举元素名称字符串
MALE
```

MALE

FEMALE

FEMALE

上述案例中，我们定义了枚举类 `Gender`，有且只有两个对象：`MALE`、`FEMALE`。

4) 枚举类分析

找到`Gender`类编译生成的字节码文件，对其反编译，观察枚举类定义细节，具体如下：

`Gender.class`文件位置：

220726-WorkSpace > 2023-CoreJava > bin > com > briup > chap07 > test				
名称	修改日期	类型	大小	
Gender.class	2023/7/26 11:17	CLASS 文件	1 KB	
Test071_EnumBasic.class	2023/7/26 11:17	CLASS 文件	1 KB	

反编译命令：`javap -p Gender.class`

```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.18363.1556]
(c) 2019 Microsoft Corporation。保留所有权利。

F:\develop\220726-WorkSpace\2023-CoreJava\bin\com\briup\chap07\test>javap -p Gender.class
Compiled from "Test071_EnumBasic.java"
final class com.briup.chap07.test.Gender extends java.lang.Enum<com.briup.chap07.test.Gender> {
    public static final com.briup.chap07.test.Gender MALE;
    public static final com.briup.chap07.test.Gender FEMALE;
    private static final com.briup.chap07.test.Gender[] ENUM$VALUES;
    static {};
    private com.briup.chap07.test.Gender(java.lang.String, int);
    public static com.briup.chap07.test.Gender[] values();
    public static com.briup.chap07.test.Gender valueOf(java.lang.String);
}
```

结合上图我们可知：

- 枚举类`Gender`本质上是一个`final`修饰的类，不可以被继承
- 枚举类会默认继承 `java.lang.Enum` 这个抽象泛型类

```
package java.lang;

public abstract class Enum<E extends Enum<E>>
```

```

        implements Comparable<E>, Serializable {
//枚举元素名
        private final String name;

        public final String name() {
            return name;
        }

//枚举元素编号，从0开始
        private final int ordinal;

        public final int ordinal() {
            return ordinal;
        }

//...省略
    }

```

- 枚举元素，本质上是**枚举类对象**，且由 **static**和**final** 修饰
- 枚举类提供私有构造器，我们在类外不能主动创建枚举类对象
- 枚举类中可以包含 **public static** 静态方法
- 其他细节我们暂不考虑

7.2 枚举基本定义

定义格式：

```

[修饰符] enum 枚举类名 {
    枚举元素1, 枚举元素2, ...枚举元素n;
}

```

案例展示：

定义枚举类Week，要求包含多个枚举元素。

```

package com.briup.chap07.test;

//枚举类基本定义
enum Week {
    //枚举元素必须写在第一行，如果有多个的话，用逗号','隔开，
    //最后用分号';'结束
    //如果';'后面没有其他内容的话，';'可以省略，但不建议省略
    MON, TUE, WED; //WED()

    //WED和WED()效果一样，含义为：执行枚举类默认构造器去实例化枚举元素对
    象
    //源代码类似：public static final Week WED = new Week();

    //注意：用户没有提供构造器，系统会提供默认的构造器 private Week()
    {}
}

public class Test072_Define01 {
    public static void main(String[] args) {
        //1.枚举元素引用格式：枚举类名.枚举元素名；
        //注意枚举类名不能省略
        Week w1 = Week.MON;
        Week w2 = Week.TUE;
        Week w3 = Week.WED;

        //2.输出枚举对象，默认输出枚举元素名
        System.out.println(w1);
        System.out.println(w2.toString());

        System.out.println("-----");

        //3.获取枚举元素名：跟元素名一样的同名字符串
        System.out.println(w3.name());

        //4.获取枚举元素编号：从0开始，逐个加1
        System.out.println(w1.ordinal()); //0
        System.out.println(w2.ordinal()); //1
    }
}

```

```
        System.out.println(w3.ordinal());    //2
    }
}
```

7.3 构造方法定义

包含数据成员、构造方法的枚举类。

定义格式：

```
[修饰符] enum 枚举类名 {
    枚举元素1(实际参数列表), ...枚举元素n(实际参数列表);

    //枚举类数据成员和成员方法，可以包含多个
    [修饰符] 数据类型 数据成员名;

    [修饰符] 返回值类型 成员方法名(形参列表) {
        方法体实现;
    }

    //枚举类构造方法，可以包含多个
    //注意，必须使用private进行修饰
    private 构造方法;
}
```

案例展示：

定义枚举类Week2，要求包含私有数据成员desc和构造方法。

```
package com.briup.chap07.test;

//包含数据成员和构造方法的枚举类
enum Week2 {
    //枚举元素必须为第一行有效代码
    //枚举元素调用的 构造方法，必须存在，否则编译报错
}
```

```

MON, TUE(), WED("星期三");

/* 源代码描述:
 *    public static final Week MON = new Week();
 *    public static final Week TUE = new Week();
 *    public static final Week WED = new Week("星期三");
 */

//枚举类数据成员和成员方法, 可以包含一个或多个
private String desc;

public String getDesc() {
    return desc;
}
public void setDesc(String desc) {
    this.desc = desc;
}

//枚举类构造方法, 如果不提供, 系统会提供默认构造方法, private修饰
//如果用户自定义枚举类构造方法, 则系统不再提供
private Week2() {}

//自定义枚举方法, 注意必须用private修饰
private Week2(String desc) {
    this.desc = desc;
}
}

public class Test073_Define02 {
    public static void main(String[] args) {
        Week2 w1 = Week2.MON;
        System.out.println(w1);

        System.out.println("-----");

        Week2 w2 = Week2.TUE;
        System.out.println(w2);
        System.out.println("w2.desc: " + w2.getDesc());
    }
}

```

```

        System.out.println("-----");

        Week2 w3 = Week2.WED;
        System.out.println(w3);
        System.out.println("w3.desc: " + w3.getDesc());
    }
}

```

7.4 抽象方法定义

包含抽象方法的枚举类.

定义格式:

```

[修饰符] enum 枚举类名 {
    枚举元素1(实参列表) {
        重写所有抽象方法;
    }, ...枚举元素n(实参列表) {
        重写所有抽象方法;
    };

    //可以包含多个抽象方法
    抽象方法声明;

    //数据成员、成员方法及构造方法略...
}

```

案例展示:

定义枚举类Week3, 要求包含抽象方法show()。

```

package com.briup.chap07.test;

//定义枚举类的第三种情况: 包含抽象方法

```



```

enum Week3 {
    //注意：包含抽象方法的枚举类是抽象类，不能直接实例化对象
    //所以定义枚举类元素(所有)时候，一定要重写抽象方法
    //注意：必须在所有的枚举元素定义中，重写所有抽象方法
    //MON {
    MON() {
        //在枚举元素中重写抽象方法
        @Override
        public void show() {
            System.out.println("in show, MON: 周一");
        }
    }, TUE("星期二") {
        //注意，每个枚举元素中都要重写重写方法，且要重写所有的抽象方法
        @Override
        public void show() {
            System.out.println("in show, TUE: " +
this.getDesc());
        }
    };

    //枚举类数据成员及get方法
    private String desc;

    public String getDesc() {
        return desc;
    }

    //枚举类自定义构造方法
    private Week3() {}

    private Week3(String desc) {
        this.desc = desc;
    }

    //枚举类 包含的 抽象方法（可以0或多个）
    public abstract void show();
}

```

```

public class Test074_Define03 {
    public static void main(String[] args) {
        Week3 w1 = Week3.MON;
        System.out.println(w1);
        w1.show();

        System.out.println("-----");

        Week3 w2 = Week3.TUE;
        System.out.println(w2);
        w2.show();
    }
}

```

7.5 枚举总结

在实际项目开发中，我们定义枚举类型，大多数情况下使用最基本的定义方式，偶尔会添加属性、方法和构造方法，并不会写的那么复杂。

枚举类注意事项：

- 定义枚举类要使用关键字 `enum`
- 所有枚举类都是 `java.lang.Enum` 的子类
- 枚举类的第一行上必须是枚举元素（枚举项）
- 最后一个枚举项后的分号是可以省略的，但是 `;` 后面还有其他有效代码，这个分号就不能省略，建议不要省略
- 用户如果不提供构造方法，系统会提供默认的构造方法：`private 枚举类() {}`
- 用户可以提供构造方法，但必须用 `private` 修饰，同时系统不再提供默认构造方法
- 枚举类也可以有抽象方法，但是枚举元素必须重写所有抽象方法

