一、面向对象的三大特征之多态

1.0 什么是多态?

- 同类型的对象，执行同一个行为，会表现出不同的行为特征。

2.0 多态的常见形式

1. 父类类型 对象名称 = new 子类构造器;
2. 接口 对象名称 = new 实现类构造器;

3.0 多态中成员访问特点

- 方法调用：编译看左边，运行看右边。
- 变量调用：编译看左边，运行也看左边。（多态侧重行为多态）

4.0 多态三前提

1. 继承关系
2. 父句柄(引用)指向子类对象(向上转型)
3. 方法重写

5.0 注意

- 多态是行为多态而不是属性多态

6.0 多态案例

```
package inteface_demo;

import javafx.animation.Animation;

/**
 * ClassName: Animal
 * Description:
 * date:2022/3/8
 *
 * @author fgcy
 * @since JDK 1.8
 */

public class Animal {
    public String name = "父类动物";


    public void run() {
        System.out.println("动物跑");
```

```
    }
}


class Dog extends Animal {
    public String name = "子类狗";

    @Override
    public void run() {
        System.out.println("狗刨");
    }
}

class Cat extends Animal {
    public String name = "子类猫";

    @Override
    public void run() {
        System.out.println("猫跑");
    }
}

class Test {
    public static void main(String[] args) {
        final Animal cat = new Cat();//1.继承关系 2.父句柄(引用)指向子类对象(向上转型) 3.方法重写 格式:父类类型/接口=new 子类构造器/实现类构造器
        cat.run();//多态指的是方法多态，编译看左边，运行看右边（编译的时候看看左边类型的变量有没有这个方法，真正是运行右边的对象的方法）
        System.out.println(cat.name);//对于变量就不涉及多态，即编译看左边，运行也看左边

        final Animal dog = new Dog();
        dog.run();
        System.out.println(dog.name);
    }
}
```

7.0 多态优势

- 在多态的形势下，右边的对象可以实现解耦合，便于扩展与维护

  ```
  ~~~
  Animal a = new Dog();//这只狗可以随时替换掉，换成猫；下面的业务照跑
  a.run();
  ~~~
  ```

- 定义方法的时候，使用父类型作为参数，该方法就可以接收这父类的一切子类对象，体现出多态的扩展性与便利

```
package inteface_demo;


/**
 * ClassName: Animal
 * Description:
 * date:2022/3/8
 *
```

```
 * @author fgcy
 * @since JDK 1.8
 */

public abstract class Animal {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

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

    abstract void run();
}

/**
 * ClassName: Animal
 * Description:
 * date:2022/3/9
 *
 * @author fgcy
 * @since JDK 1.8
 */

class Dog extends Animal {

    public Dog(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println(getName() + "狗刨");

    }
}

/**
 * ClassName: Animal
 * Description:
 * date:2022/3/9
 *
 * @author fgcy
 * @since JDK 1.8
 */
class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    @Override
```

```
    public void run() {
        System.out.println(getName() + "猫跑");
    }
}

class MyTest {
    //这个方法通过多态类型的局部变量来接收参数，提高了扩展性，可以接收这个类本类和所有的子类
    public static void go(Animal animal) {
        System.out.println("开始");
        animal.run();
        System.out.println("结束");
    }


    public static void main(String[] args) {
        final Dog dog = new Dog("旺财");//局部变量收到该值（子类对象）时体现多态
        final Animal dog2 = new Dog("旺财");//此时就体现多态
        final Cat cat = new Cat("Tom");
        final Animal cat2 = new Cat("Tom");
        go(dog);//这个被称为-------》对象回调，我把对象送过去，执行的是这个对象的方法
        go(cat);
    }


}
```

8.0 问题

多态下不能使用子类的独有功能
原因编译看左边，运行看右边，所以编译时就会报错
    解决：
     向下转型，强制类型转化

8.1 引用数据类型的类型转换

- **自动类型转换**

  （从子到父）：子类对象赋值给父类类型的变量指向。

- **强制类型转换**

  （从父到子）：此时必须进行强制类型转换：子类 对象变量 = (子类)父类类型的变量

  注意

```
如果转型后的类型和对象真实类型不是同一种类型，那么在转换的时候就会出现
ClassCastException;
        Animal t = new Tortoise();
//有继承关系/实现的2个类型就可以进行强制转换，编译无问题;如果发现强制转换后的类型不是对象真实类型则出现异常 ClassCastException
        Dog d = (Dog)t;

Java建议强转转换前使用instanceof判断当前对象的真实类型，再进行强制转换
    变量名 instanceof 真实类型
```

判断关键字左边的变量指向的对象的真实类型，是否是右边的类型或者是其子类类型，是则返回true，反之

```
    //String与基本数据类型及其包装类之间不能强转-------》ClassCaseException
       /*
        Object a = "aaa";
        int b = (int) a;
        Object a=123;
        String b=(String) a;
        */
```

```
package inteface_demo;


/**
 * ClassName: Animal
 * Description:
 * date:2022/3/8
 *
 * @author fgcy
 * @since JDK 1.8
 */

public abstract class Animal {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

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

    abstract void run();
}

/**
 * ClassName: Animal
 * Description:
 * date:2022/3/9
 *
 * @author fgcy
 * @since JDK 1.8
 */

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

    @Override
```

```
    public void run() {
        System.out.println(getName() + "狗刨");

    }

    public void eat() {
        System.out.println("狗吃🍙");
    }
}

/**
 * ClassName: Animal
 * Description:
 * date:2022/3/9
 *
 * @author fgcy
 * @since JDK 1.8
 */
class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println(getName() + "猫跑");
    }

    public void eat() {
        System.out.println("猫吃九");
    }
}

class MyTest {
    //这个方法通过多态类型的局部变量来接收参数，提高了扩展性，可以接收这个类本类和所有的子类
    public static void go(Animal animal) {

        if (animal instanceof Dog) {//判断左边是否是右边的本类型或者是子类型
            Dog dog = (Dog) animal;//只要两者由继承关系，编译时就不会报错；但如果类型不一致，运行时就会报ClassCaseException
            dog.eat();
        }
        if (animal instanceof Cat) {
            Cat cat = (Cat) animal;
            cat.eat();
        }
    }


    public static void main(String[] args) {
        final Cat cat = new Cat("汤姆");
        go(cat);
        final Dog dog = new Dog("谢狗");
        go(dog);
    }
}
```

好处

> 可以解决多态下的劣势，可以实现调用子类独有的功能

9.0 多态综合案例

```
package polymorphic;

/**
 * ClassName: Usb
 * Description:Usb接口，规定有公共的方法，连接和断开连接
 * date:2022/3/10
 *
 * @author fgcy
 * @since JDK 1.8
 */
public interface Usb {

    void connect();

    void unconnect();
}
-----------------------------------------------------------------------------------
---------------------------------------


    package polymorphic;

/**
 * ClassName: Mouse
 * Description:鼠标类，Usb接口的实现类，是现有连接和断开连接的方法 还有自己独有的功能
 * date:2022/3/10
 *
 * @author fgcy
 * @since JDK 1.8
 */

public class Mouse implements Usb {
    private String name;

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void connect() {
        System.out.println(getName() + "已连接");
    }
```

```
    @Override
    public void unconnect() {
        System.out.println(getName() + "已移除");
    }

    //自己独有的功能
    public void click() {
        System.out.println("鼠标点击");
    }
}
-------------------------------------------------------------------------------
---------------------------------------
    package polymorphic;


/**
 * ClassName: Keyboard
 * Description:键盘类，Usb接口的实现类，是现有连接和断开连接的方法，还有自己独有的功能
 * date:2022/3/10
 *
 * @author fgcy
 * @since JDK 1.8
 */

public class Keyboard implements Usb {
    private String name;

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void connect() {
        System.out.println(getName() + "已连接");
    }

    @Override
    public void unconnect() {
        System.out.println(getName() + "已移除");
    }
    //自己独有的功能
    public void write() {
        System.out.println("键盘输入测试");
    }
}
-------------------------------------------------------------------------------
---------------------------------------
    package polymorphic;
```

```java
import java.security.KeyStore;

/**
 * ClassName: Computer
 * Description:连接Usb的实现类，并调用实现类的方法
 * date:2022/3/10
 *
 * @author fgcy
 * @since JDK 1.8
 */

public class Computer {
    private String name;

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void start() {
        System.out.println(getName() + "电脑启动");
    }

    public void connect(Usb usb) {
        usb.connect();//USB接口的独有功能

        if (usb instanceof Keyboard) {
            Keyboard keyboard = (Keyboard) usb;
            keyboard.write();
        }
        if (usb instanceof Mouse) {
            Mouse mouse = (Mouse) usb;
            mouse.click();
        }
        usb.unconnect();//USB接口的独有功能
    }
    ----------------------------------------------------------------------------
--------------------------
            /**
     * ClassName: Computer
     * Description:面向对象编程（将鼠标和键盘组装到电脑中）
     * date:2022/3/10
     *
     * @author fgcy
     * @since JDK 1.8
     */

    public static void main(String[] args) {
        final Computer computer = new Computer("拯救者");
        final Keyboard keyboard = new Keyboard("罗技");
        final Mouse mouse = new Mouse("罗技");
```

```
        computer.start();
        computer.connect(keyboard);//对象回调（将一个对象作为参数传递过去，实际调用的是该对象的方法）
        computer.connect(mouse);//对象回调（将一个对象作为参数传递过去，实际调用的是该对象的方法）

    }
================================================================
救者电脑启动
罗技已连接
键盘输入测试
罗技已移除
罗技已连接
鼠标点击
罗技已移除
================================================================
```

二、内部类

1.0 内部类概念

定义在一个类里面的类，里面的类可以理解成（寄生），外部类可以理解成（宿主）

- 形式

```
public class People{
    // 内部类
    public class Heart{
    }
}
```

- 特点

1. 当一个事物的内部，还有一个部分需要一个完整的结构进行描述，而这个内部的完整的结构又只为外部事物提供服务，那么整个内部的完整结构可以选择使用内部类来设计。
2. 内部类通常可以方便访问外部类的成员，包括私有的成员。
3. 内部类提供了更好的封装性，内部类本身就可以用private protectecd等修饰，封装性可以做更多控制
4. 内部类的修饰符可以是:private、缺省、protected、public，但类的修饰符只能是public（主类）和缺省（次类）

2.0 静态内部类

- 特点

1.有static修饰，属于外部类本身
2.它的特点和使用与普通类是完全一样的，类有的成分它都有，只是位置在别人里面而已
3.静态内部类中可以直接访问外部类的静态成员，外部类的实例成员必须用外部类对象访问
4.开发中实际上用的还是比较少

- 格式

```
public class Outer{
    // 静态成员内部类
    public static class Inner{
    }
}
静态内部类创建对象的格式:
    外部类名.内部类名 对象名 = new 外部类名.内部类构造器;
    范例：Outer.Inner in =  new Outer.Inner();
```

- 案例

```
package inner_class_demo;


public class StaticInnerClass {
    private static int price;
    private String sex;

    public static class Inner {
        private String name;//静态内部类可以有实例成员变量
        private static String codeName;//静态内部类可以有静态成员变量


        //静态内部类可以有实例成员方法
        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public static String getCodeName() {
            return codeName;
        }

        public static void setCodeName(String codeName) {
            Inner.codeName = codeName;
        }

        public Inner() {
        }

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

        //静态内部类可以有静态成员方法
        public static void voice() {
            System.out.println("静态内部类的静态方法访问外部类的静态变量" + price);
        }

        public void init() {
//            System.out.println(sex);不能直接访问外部类的实例成员变量
            //只能用外部类的实例来调用
            final StaticInnerClass outClass = new StaticInnerClass();
```

```
            System.out.println(outClass.sex);
        }
    }


    public static void main(String[] args) {
        //实例化静态内部类的格式 注意类型，走的还是构造器
        StaticInnerClass.Inner inner = new StaticInnerClass.Inner("a");
        System.out.println(inner.getName());
        inner.voice();//调用内部类的静态方法 不推荐写法
        StaticInnerClass.Inner.voice();//调用内部类的静态方法 正确写法
//        StaticInnerClass.voice();//调用内部类的静态方法 错误写法


    }
}
================================================================================
a
静态内部类的静态方法访问外部类的静态变量0
静态内部类的静态方法访问外部类的静态变量0
```

3.0 成员内部类

1无static修饰，属于外部类的对象。

2.JDK16之前，成员内部类中不能定义静态成员，JDK 16开始也可以定义静态成员了。

创建格式：
    格式：外部类名.内部类名 对象名 = new 外部类构造器.new 内部类构造器();
    范例：Outer.Inner in = new Outer().new Inner();

```
public class Outer {
    // 成员内部类
    public class Inner {
    }
}
```

3.成员内部类中是可以直接访问外部类的静态成员，外部类的静态成员只有一份可以被共享访问

4、成员内部类的实例方法中可以直接访问外部类的实例成员，因为必须先有外部类对象，才能有成员内部类对象，所以可以直接访问外部类对象的实例成员

5.静态内部类，内部类与外部类的关系相对独立；类似于定义了两个类；成员内部类则需要依赖外部类才能产生，可能更加契合面向对象语法

```
package inner_class_demo;


public class MemberClass {
    private static String name;
    private int sex;

    public MemberClass(int sex) {
        this.sex = sex;
    }

    class Inner {
```

```
        private String number;
//        private static String hobbis;//jdk16之后才可以定义静态的变量和方法

        public Inner(String number) {
            this.number = number;
        }

        public String getNumber() {
            return number;
        }

        public void setNumber(String number) {
            this.number = number;
        }

        /*   public static void t(){
            System.out.println("jdk16之后才可以定义静态的方法");
        }*/
        public void get() {
            System.out.println("外部类的实例变量" + sex);
            System.out.println("外部类的静态变量" + name);
        }
    }

    public static void main(String[] args) {
        //成员内部类实例需要使用外部类的实例调用内部类的构造器生成
        final Inner inner = new MemberClass(12).new Inner("138");
        System.out.println(inner.getNumber());//访问成员内部类的成员方法
        inner.get();
//        MemberClass.Inner.t();//调用成员内部类的静态方法
    }
}
```

4.0 面试题

```
package inner_class_demo;

/**
 * ClassName: StaticInnerClass
 * Description:静态内部类访问不同位置的变量
 * date:2022/3/10
 *
 * @author fgcy
 * @since JDK 1.8
 */

public class StaticInnerClass {
    private static int price;
    private static String name = "外部类变量";

    public static class Inner {
        private String name = "成员变量";//静态内部类可以有实例成员变量

        public void test() {
            String name = "局部变量";
            System.out.println(name);
```

```
            System.out.println(this.name);
            //（外部类类名.this.变量名） 访问的是成员内部类的外部类的变量 有this代表对象所以这种方式是成员内部类的方式
            System.out.println(StaticInnerClass.name);
        }
    }
    public static void main(String[] args) {
        final Inner inner = new Inner();
        inner.test();
    }
}

==========================================================
局部变量
成员变量
外部类变量
-------------------------------------------------------------------------------------
----
    package inner_class_demo;


/**
 * ClassName: MemberClass
 * Description:成员内部类访问不同位置的变量
 * date:2022/3/10
 *
 * @author fgcy
 * @since JDK 1.8
 */

public class MemberClass {
    private String name = "外部类的变量";

    class Inner {
        private String name = "成员变量";

        public void test() {
            String name = "局部变量";
            System.out.println(name);
            System.out.println(this.name);
            System.out.println(MemberClass.this.name);
        }
    }

    public static void main(String[] args) {
        final Inner inner = new MemberClass().new Inner();
        inner.test();
    }
}
================================================================

局部变量
成员变量
外部类变量
```

5.0 局部内部类

（鸡肋语法，了解即可）
局部内部类放在方法、代码块、构造器等执行体中。
局部内部类的类文件名为： 外部类$N内部类.class。

6.0 匿名内部类

本质上是一个没有名字的局部内部类，定义在方法中、代码块中、等。
作用：方便创建子类对象，最终目的为了简化代码编写

格式：

```
new 类|抽象类名|或者接口名() {
重写方法;
};
    Animal a = new Animal() {
        public void run() {
        }
    };
    a. run();
```

特点总结：

匿名内部类是一个没有名字的内部类。
匿名内部类写出来就会产生一个匿名内部类的对象。
匿名内部类的对象类型相当于是当前new的那个的类型的本类型或子类类型
局部内部类的类文件名为： 外部类$N内部类.class。

```
package incognit_demo;


public class Father {
    public void eat() {
        System.out.println("父吃");
    }

    public static void main(String[] args) {

        //匿名内部类，存在与方法，代码块，构造器中（本质上就是局部内部类，并且没有引用指向）
        //这里是类的匿名内部类
        Father father = new Father() {
            @Override
            public void eat() {
                super.eat();
                System.out.println("子也吃");
            }
        };
        father.eat();
    }
}
```

```
================================================================
父吃
子也吃

--------------------------------------------------------------------------------
----------------------------------------
package incognit_demo;
```

```
public abstract class Father {
    public abstract void eat();

    public static void main(String[] args) {

        //匿名内部类，存在与方法，代码块，构造器中
        //这里是抽象类的匿名内部类
        //这里的匿名内部类会产生一个该抽象类的子类实例，通过夫类型引用接
        Father father = new Father() {
            @Override
            public void eat() {
                System.out.println("通过匿名内部类获得匿名内部类对象（抽象类的子类实
例）");
            }
        };
        father.eat();
    }
}

================================================================
通过匿名内部类获得匿名内部类对象（抽象类的子类实例）
--------------------------------------------------------------------------------
-----------------------------------------  package incognit_demo;


public class Test {

    public static void main(String[] args) {
        final Father father = new Father() {
            @Override
            public void eat() {
                System.out.println("通过匿名内部类获得匿名内部类对象（接口实现类的实
例）");
            }
        };
        father.eat();
    }
}

interface Father {
    void eat();//公开，抽象
}
============================================================
通过匿名内部类获得匿名内部类对象（接口实现类的实例）
```

- 实际开发案例

```
package incognit_demo;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * ClassName: PracticalDemo <br/>
```

```
 * Description: <br/>
 * date: 2022/3/10 13:22<br/>
 *
 * @author fgcy<br />
 * @since JDK 1.8
 */
public class PracticalDemo {
    public static void main(String[] args) {
        final JFrame win = new JFrame("登录");

        final JPanel panel = new JPanel();
        final JButton btnLogin = new JButton("登录");
//对象回调
//         匿名内部类产生的匿名内部类对象作为入参
   /*     btnLogin.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(win, "点击小子");
            }
        });*/

        //lambda表达式
        btnLogin.addActionListener(e -> JOptionPane.showMessageDialog(win, "点击
小子2"));

        panel.add(btnLogin);
        win.add(panel);
        win.setVisible(true);//可见
        win.setLocationRelativeTo(null);//居中
        win.setSize(400, 300);//大小
    }
}
```

开发中不是我们主动去定义匿名内部类的，而是别人需要我们写或者我们可以写的时候才会使用。
匿名内部类的代码可以实现代码进一步的简化（回扣主题）

类的四大金刚：
    构造器，方法，属性，代码块

三、常用API

- **API**

```
API(Application Programming interface)  应用程序编程接口。
简单来说：就是Java帮我们已经写好的一些方法，我们直接拿过来用就可以了
```

1.0 Object

```
一个类要么默认继承了Object类，要么间接继承了Object类，Object类是Java中的祖宗类
Object类的方法是一切子类对象都可以直接使用的，所以我们要学习Object类的方法
```

Object类的常用方法：

方法名	说明
public String toString()	默认是返回当前对象在堆内存中的地址信息:类的全限名@内存地址
public boolean equals(Object o)	默认是比较当前对象与另一个对象的地址是否相同，相同返回true，不同返回false

- **toString()**

Object类中的toString方法，默认是返回一个字符串地址的
但是，开发中直接输出对象，默认输出对象的地址其实是毫无意义的，开发中输出对象变量，更多的时候是希望看到对象的内容数据而不是对象的地址信息
所以，toString存在的意义是为了被子类重写，以便返回对象的内容信息，而不是地址信息！！

```
    @Override
    public String toString() {
        return "Student{name='" + name + "\'" +
                ",dept='" + depth + "\'" +
                ",age=" + age +
                ",id='" + id + "\'}";
    }
--------------------------------------------------------------------------------

public static void main(String[] args) {
        final Student student = new Student("张三", "调理农务系", 18,
"1212121210");
//        System.out.println(student.toString());//本来需要这么写
        System.out.println(student);//但是，是直接输出的话可以省略toString方法，他也会自动调用
    }

================================================================================
    Student{name='张三',dept='调理农务系',age=18,id='1212121210'}
```

- **equals()**

1. Object的equals方法默认是比较两个对象地址是否相等，Object的equals方法和==是等价的，本质就是equals调用了==

```
 public boolean equals(Object obj) {
        return (this == obj);
  }
```

2. 父类equals方法存在的意义就是为了被子类重写，以便子类自己来定制比较规则

```
  @Override
        public boolean equals(Object o) {
            if (o instanceof Student) {
                Student student = (Student) o;
                return Objects.equals(this.name, student.getName()) &&
                        Objects.equals(this.depth, student.getDepth()) &&
Objects.equals(this.id, student.getId()) &&
                        this.age == student.getAge();
            }
            return false;
```

```
        }


    @Override
     public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Student student = (Student) o;
        return age == student.age && Objects.equals(name, student.name) &&
Objects.equals(depth, student.depth) && Objects.equals(id, student.id);
    }
```

2.0 Objects

Objects是一个工具类，提供了一些方法去完成一些功能，是Object的子类
官方在进行字符串比较时，没有用字符串对象的的equals方法，而是选择了Objects的equals方法来比较
使用Objects的equals方法在进行对象的比较会更安全

```
public static boolean equals(Object a, Object b) {
        return (a == b) || (a != null && a.equals(b));
    }
```

首先看看是不是同一个对象，是直接返回真;否则看看调用者是否为null，若为null直接返回假，否则开始比较内容

3.0 StringBuilder

StringBuilder是一个可变的字符串类，我们可以把它看成是一个对象容器。
作用：提高字符串的操作效率，如拼接、修改等

StringBuilder 构造器

名称	说明
public StringBuilder()	创建一个空白的可变的字符串对象，不包含任何内容
public StringBuilder(String str)	创建一个指定字符串内容的可变字符串对象

StringBuilder常用方法

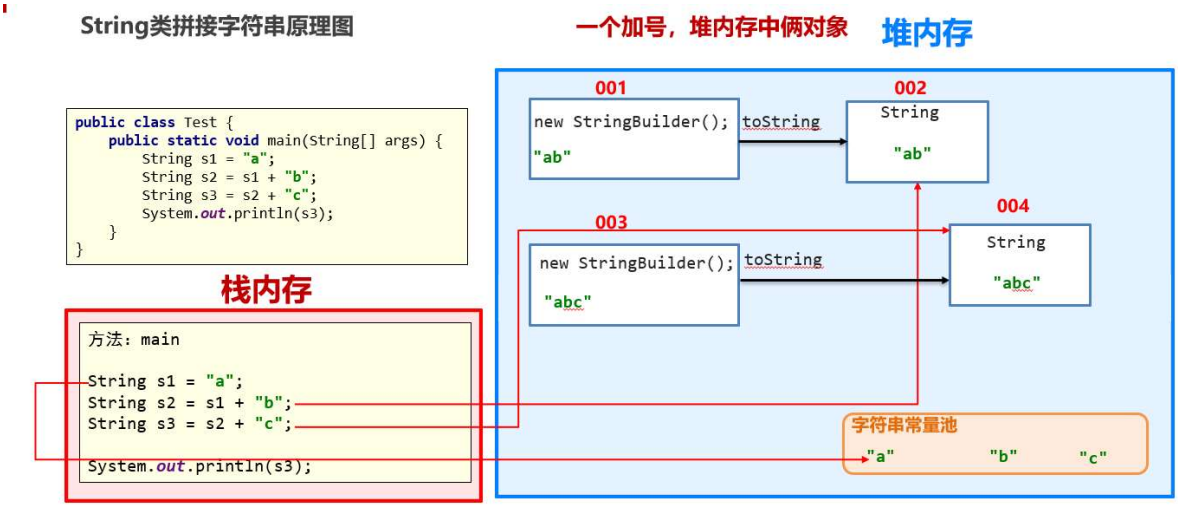
方法名称	说明
public StringBuilder append(任意类型)	添加数据并返回StringBuilder对象本身
public StringBuilder reverse()	将对象的内容反转
public int length()	返回对象内容长度
public String toString()	通过toString()就可以实现把StringBuilder转换为String

```
package api_object;
```

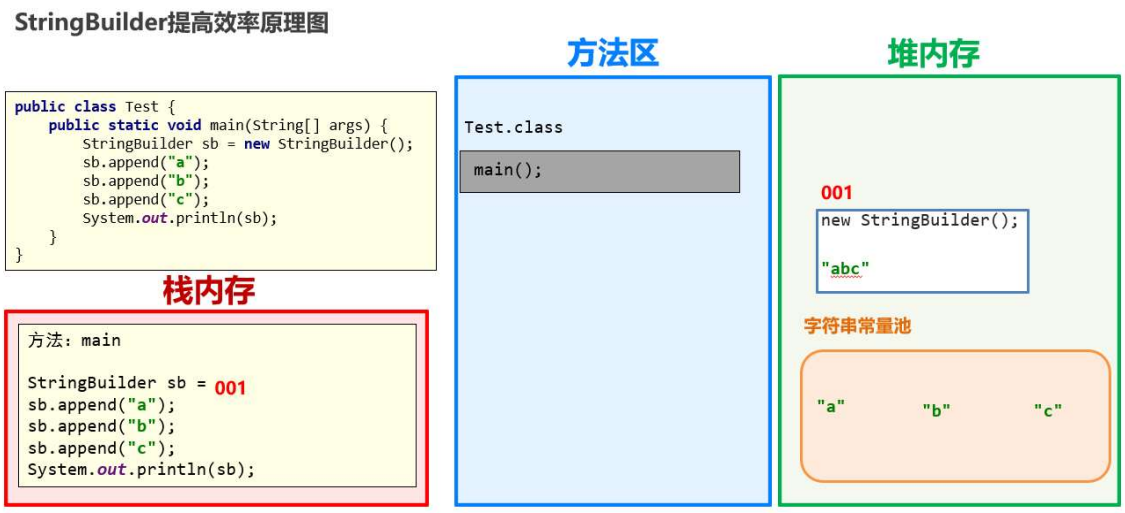
```java
/**
 * ClassName: StringBuilderDemo
 * Description:
 * date:2022/3/10
 *
 * @author fgcy
 * @since JDK 1.8
 */
public class StringBuilderDemo {
    public static void main(String[] args) {
        final StringBuilder sb = new StringBuilder();
        System.out.println(sb);//直接输出默认会调用该对象的toString方法 ""

        sb.append(123).append('a').append(true).append(12.3).append("时");//支持链式编程
        System.out.println(sb.toString());//直接输出默认会调用该对象的toString方法

        final StringBuilder reverse = sb.reverse();//将源对象的值反转，并返回一个带有反转后结果的StringBuilder对象
        System.out.println(reverse);

        final String s = sb.toString();//将StringBuilder转为String
        System.out.println(s);

        System.out.println(sb.length());
    }
}
```

```
================================================================================
==================================
123atrue12.3时
时3.21eurta321
时3.21eurta321
13
```

3.1 String类拼接字符串原理图



1. 首先将Test.class文件加载进方法区内存，并暴露访问静态方法的入口
2. 将main方法提取到栈内存中运行
3. 在栈内存中开辟一块存储String类型变量的空间，在字符串常量池中开辟一块空间存储字符“a”的实例把地址给到栈
3.在栈内存中开辟一块存储String类型变量的空间，在字符串常量池中开辟一块空间存储字符“b”的实例，然后在堆内存中创建一个StringBuilder的对象用于拼接字符串，然后将拼接好的StringBuilder对象转为String对象，再把地址给到栈中的变量
4. 同理
.....
一个加号，堆内存中两对象



1. 首先将Test.class文件加载进方法区内存，并暴露访问静态方法的入口
2. 将main方法提取到栈内存中运行
3. 在栈内存中创建一个StringBuilder类型的变量，在堆内存中创建一个StringBuilder对象，将对象的地址给到变量
4. 在字符串常量池中创建一个对象，然后把字符串对象添加到StringBuilder对象中进行操作
.....

- 总结

String ：内容是不可变的、拼接字符串性能差。
StringBuilder：内容是可变的、拼接字符串性能好、代码优雅。

定义字符串使用String
拼接、修改等操作字符串使用StringBuilder

3.2 数组拼接输出

```
public String concat(int[] arr) {
        if (arr == null) return null;
        final StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < arr.length; i++) {
            sb.append(arr[i]).append(i == arr.length - 1 ? "" : ",");
        }
        sb.append("]");
        return sb.toString();
```

```
    }

    @Test
    public void test1() {
        System.out.println(concat(null));
        System.out.println(concat(new int[0]));
        System.out.println(concat(new int[]{12, 55, 18, 34, 78, 145, 21}));
    }
============================================================
null
[]
[12,55,18,34,78,145,21]
```

4.0 Math

包含执行基本数字运算的方法，Math类没有提供公开的构造器。工具类一般都会私有构造器
如何使用类中的成员呢？看类的成员是否都是静态的，如果是，通过类名就可以直接调用

Math 类的常用方法

方法名	说明
public static int abs(int a)	获取参数绝对值
public static double ceil(double a)	向上取整
public static double floor(double a)	向下取整
public static int round(float a)	四舍五入
public static int max(int a,int b)	获取两个int值中的较大值
public static double pow(double a,double b)	返回a的b次幂的值
public static double random()	返回值为double的随机值，范围[0.0,1.0)

```
    @Test
    public void test1() {
        System.out.println(Math.abs(-887));//887
        System.out.println(Math.sqrt(9));//3
        System.out.println(Math.max(12, 1 - 2));//12
        final int i = (int) (Math.random() * 7) + 3;//3----9
        System.out.println(i);
        System.out.println(Math.pow(2, 10));//1024
        System.out.println(Math.round(3.14));//3
        System.out.println(Math.round(3.5));//4
        System.out.println(Math.ceil(1.000001));//2
        System.out.println(Math.ceil(1.0));//1
        System.out.println(Math.floor(1.9999999999));//1
        System.out.println(Math.floor(1.0));//1
    }
============================================================
887
3.0
12
5
1024.0
```

```
3
4
2.0
1.0
1.0
1.0
```

5.0 System

System也是一个工具类，代表了当前系统，提供了一些与系统相关的方法

System 类的常用方法

方法名	说明
public static void exit(int status)	终止当前运行的 Java 虚拟机，非零表示异常终止
public static long currentTimeMillis()	返回当前系统的时间毫秒值形式
public static void arraycopy(数据源数组, 起始索引, 目的地数组, 起始索引, 拷贝个数)	数组拷贝

```java
package api_object;

import org.junit.Test;

import java.util.Arrays;

/**
 * ClassName: SystemDemo <br/>
 * Description: <br/>
 * date: 2022/3/10 15:51<br/>
 *
 * @author fgcy<br />
 * @since JDK 1.8
 */
public class SystemDemo {
    @Test
    public void test1() {

        System.out.println("程序开始");

        final long start = System.currentTimeMillis();
        int counter = 0;
        for (int i = 0; i < 100000000; i++) {
            counter++;
        }
        final long end = System.currentTimeMillis();
        System.out.println("循环一亿次，耗时："+(end - start) / 1000.0);


        final int[] ints1 = {12, 2, 1, 78, 45, 64, 55, 34};
        final int[] ints2 = new int[5];

        System.arraycopy(ints1, 1, ints2, 2, 3);
```

```
        System.out.println(Arrays.toString(ints2));
        System.exit(0);//人为关掉JVM


        System.out.println("程序结束");

    }
}
```

```
==========================================================================
程序开始
循环一亿次，耗时：0.002
[0, 0, 2, 1, 78]
```

1970年1月1日 算C语言的生日
原因：
1969年8月，贝尔实验室的程序员肯汤普逊利用妻儿离开一个月的机会，开始着手创造一个全新的革命性的操作系统，他使用B编译语言在老旧的PDP-7机器上开发出了 Unix的一个版本。
随后，汤普逊和同事丹尼斯里奇改进了B语言，开发出了C语言，重写了UNIX。

计算机认为时间是有起点的，起始时间： 1970年1月1日 00:00:00

时间毫秒值：指的是从1970年1月1日 00:00:00走到此刻的总的毫秒数，应该是很大的。 1s = 1000ms

6.0 BigDecimal

用于解决浮点型运算精度失真的问题

- 失真问题

```
    @Test
    public void test1() {
        System.out.println(0.09 + 0.01);//0.09999999999999999
        System.out.println(1.0 - 0.32);//0.6799999999999999
        System.out.println(1.015 * 100);//101.49999999999999
        System.out.println(1.301 / 100);//0.013009999999999999
        System.out.println("-------------------------");
        double c = 0.1 + 0.2;//0.30000000000000004
        System.out.println(c);
    }
```

- 使用

1. 创建对象BigDecimal封装浮点型数据（最好的方式是调用方法）
    public static BigDecimal valueOf(double val): 包装浮点数成为BigDecimal对象。

2. 通过BigDecimal类的构造器得到BigDecimal对象（比较麻烦，不推荐）
    BigDecimal(String val)//Translates the string representation of a BigDecimal into a BigDecimal.
3. 其他方法不推荐，存在精度问题

BigDecima常用API

方法名	说明
public BigDecimal add(BigDecimal b)	加法
public BigDecimal subtract(BigDecimal b)	减法
public BigDecimal multiply(BigDecimal b)	乘法
public BigDecimal divide(BigDecimal b)	除法
public BigDecimal divide (另一个BigDecimal对象，精确几位，舍入模式)	除法

```
    @Test
    public void test2() {
        final BigDecimal a = new BigDecimal("0.1");
        final BigDecimal b = new BigDecimal("0.2");
        System.out.println(a.add(b));//0.3

        final BigDecimal c = BigDecimal.valueOf(1.0);
        final BigDecimal d = BigDecimal.valueOf(0.32);
        System.out.println(c.subtract(d));//0.68

        final BigDecimal e = BigDecimal.valueOf(1.015);
        final BigDecimal f = BigDecimal.valueOf(100);
        System.out.println(e.multiply(f));//101.500

        final BigDecimal g = BigDecimal.valueOf(1.301);
        System.out.println(g.divide(f));//0.01301

    }
```

```
    @Test
    public void test3(){
        final BigDecimal a = new BigDecimal("10");
        final BigDecimal b = new BigDecimal("3");
        System.out.println(a.divide(b,2,BigDecimal.ROUND_UP));
    }
```

参数1 ：表示参与运算的BigDecimal 对象。
    参数2 ， 表示小数点后面精确到多少位
    参数3 ， 舍入模式
    BigDecimal.ROUND_UP 进一法
    BigDecimal.ROUND_FLOOR 去尾法
    BigDecimal.ROUND_HALF_UP 四舍五入