代码块{}

Java变量的作用域:块作用域

测试框架

一,代码块的分类:

1.普通代码块:在方法内的代码块

```
//测试普通代码块的执行流程
    //调用方法时,普通代码块从上倒下依次执行
    @Test
    void test2() {
        {
            System.out.println("普通代码快1");
        }
        {
            System.out.println("普通代码快2");
        }
        System.out.println("2");

    }
```

2.构造代码块;在类中的代码块

```
    //测试构造代码的执行流程:
    // 创建对象时,构造代码块从上倒下依次执行
@Test
    void test3() {
        Girl girl=new Girl();
                                ();
    }
```

Variable 'girl' is never used

3.静态代码块：由static修饰的代码块

```
    //测试静态代码块的执行流程:
    //当类被第一次加载时执行
    //静态代码块从上向下先执行,在执行构造代码块
    @Test
    void test4() {
        Girl girl=new Girl();
        Girl girl1=new Girl();

    }
}
```

```
static {
    System.out.println("静态代码块1!!!");
}
{
    System.out.println("构造代码块1!!!!");
}
{
    System.out.println("构造代码块2!!!!");
}

public void setAge(int age) { this.age = age; }

static {
    System.out.println("静态代码块2!!!");
}
```

二.类的分类

普通类,抽象类,final

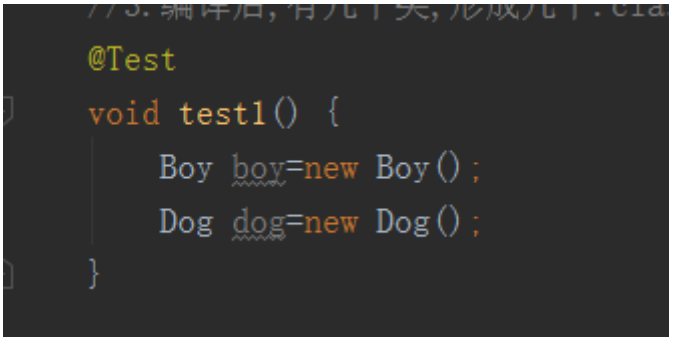
一个文件中可以写多个类,内部类,外部类,匿名内部类

测试一个文件中可以写多个类

1. 一个文件只能有一个public修饰的类

2. 文件名要和public修饰的类相同

3. 编译后,有几个类,形成几个.class文件



测试外部类和内部类

1. 内部类为外部类服务的,一般很少创建内部类

2. 要实例化内部类,需要先实例化外部类

匿名内部类:定义在方法内

```
@Test
void test3() {
 Integer[] array=new Integer[10];
 Random random=new Random();
 for (int i = 0; i < array.length; i++) {
 array[i]= random.nextInt(10)+33;
 }
 System.out.println(Arrays.toString(array));
 //数组排序
  //升序
  Arrays.sort(array);
  System.out.println(Arrays.toString(array));

```

```
14
15  //降序
```

```
public class MyRule implements Comparator<Integer> {


    public int compare(Integer o1,Integer o2) {
        if (o1>o2){
            return -1;
        }else if (o1<o2){
            return 1;
        }else {
            return 0;
        }

    }
}
```

```
1  Comparator<Integer> comparator=new MyRule();
2  Arrays.sort(array);
3  System.out.println(Arrays.toString(array));
```

```
1  //创建了一个匿名内部类的格式
2  /*
3  new 类名/接口(){
4  重写的方法
5  }
6  */
```

另一种写法：

```
1  Comparator<Integer>comparator1=new Comparator<Integer>() {
2  public int compare(Integer o1, Integer o2) {
3  return -o1.compareTo(o2);
4  }
5  };
6  Arrays.sort(array);
7  System.out.println(Arrays.toString(array));
8
```

```
Arrays.sort(array, new Comparator<Integer>() {
 public int compare(Integer o1, Integer o2) {
 return 0;
 }
 });
```

三. 反射:通过反射的技术,可以获取一个类的属性和方法

获取类对象,实质获取字节码文件

以小汽车为例,在类中,小汽车的价格为私有的,通过映射我们可以获取小汽车的私有构造方法和自己写的方法

```
public String brand;
private double price;

public Car() {
}

private Car( double price) {
 this.price = price;
}

public Car(String brand, double price) {
 this.brand = brand;
 this.price = price;
}

public void run(int speed){
 System.out.printf("%s以%d的速度再跑!\n",brand,speed);
}
private void stop(){
 System.out.println("停车!!!");
}

@Override
public String toString() {
 return "Car{" +
 "brand='" + brand + '\'' +
```

```
27      ", price=" + price +
28      '}';
}
```

```
    //获取类对象,实质获取字节码文件
    @Test
    void test() throws ClassNotFoundException {
        //方式一:类.class
        Class<Car> carClass = Car.class;
        System.out.println(carClass);

        //方式二:对象.getClass
        Car car = new Car();
        Class<? extends Car> aClass = car.getClass();
        System.out.println(aClass);

        //方式三:Class.forName("全类名")
        Class<?> aClass1 = Class.forName("com.lanou.Car");
        System.out.println(aClass1);
    }
```

```
@Test
void test3() throws ClassNotFoundException, NoSuchMethodException, InvocationTargetEx
    Class<?> aClass = Class.forName("com.lanou.Car");
    //获取方法
    Method method = aClass.getMethod( name: "run", int.class);
    //调用方法
    method.invoke(car,  ...args: 120);
    Method method1 = aClass.getDeclaredMethod( name: "stop");
    method1.setAccessible(true);
    method1.invoke(car);
}

//通过反射获取构造方法,并实例化对象
```

```java
@Test
void test2() throws ClassNotFoundException, NoSuchMethodException, IllegalAccessException, InvocationTargetExcept
    //获取类对象
    Class aClass = Class.forName("com.lanou.Car");
    //获取无参构造方法
    Constructor constructor = aClass.getConstructor();
    System.out.println(constructor);
    //通过构造方法创建对象
    Object o = constructor.newInstance();
    System.out.println(o instanceof Car);
    //获取有参构造方法
    Constructor constructor1 = aClass.getConstructor(String.class, double.class);
    System.out.println(constructor1);
    //通过构造方法创建对象
    Object o1 = constructor1.newInstance( ...initargs: "五菱宏光", 50000);
    System.out.println(o1);
    //获取有参构造方法
    Constructor constructor2 = aClass.getDeclaredConstructor(double.class);
    //设置是否可以访问
    constructor2.setAccessible(true);
    Object o2 = constructor2.newInstance( ...initargs: 12345);
    System.out.println(o2);

    /*
```

aClass.getConstructor();获取某个public的构造方法

aClass.getConstructors();获取全部public的构造方法

aClass.getDeclaredConstructor();获取某个构造方法

aClass.getDeclaredConstructors();获取全部构造方法

```java
//属性
    @Test
    void test4() throws ClassNotFoundException, NoSuchFieldException, IllegalAccessException {
        Class<?> aClass = Class.forName("com.lanou.Car");
        //获取属性
        Field brandField = aClass.getField( name: "brand");
        //修改属性的值
        brandField.set(car,"法拉第");
        //获取属性的值
        System.out.println(brandField.get(car));
        Field priceField = aClass.getDeclaredField( name: "price");
        priceField.setAccessible(true);
        priceField.set(car,555555555);
        System.out.println(priceField.get(car));
        System.out.println(car);
```

四,lambda表达式

格式:

(参数列表)->执行的代码

lambda表达式用于替代匿名内部类

```
@Test
void test() {
 new Thread(new Runnable() {
 public void run() {
 System.out.println("睡");
 }
 }).start();
 new Thread(() -> System.out.println("睡")).start();
}
```

lambda表达式用于遍历集合

```
@Test
void test2() {
 List<String> list = Arrays.asList("Java", "PHP", "C", "C#", "Python");
```

方式一

```
 for (int i = 0; i < list.size(); i++) {
 System.out.println(list.get(i));
 }
```

方式二,内部修改不了集合

```
 for (String s : list) {
 System.out.println(s);
 }
```

方式三

```
 for (Iterator<String> iterator = list.iterator(); iterator.hasNext(); )
{
 String next = iterator.next();
 System.out.println(next);
 }
```

方式四

```
 list.forEach(new Consumer<String>() {
 @Override
 public void accept(String s) {
 System.out.println(s);
```

```
5   }
6   });
```

方式5

```
1   list.forEach((x)-> System.out.println(x));
2   list.forEach(x-> System.out.println(x));
```

方式6

```
1   list.forEach(System.out::println):方法的引用
```

格式:

类::方法

把方法当成参数.传递进去,每个元素依次调用这个方法

```
1   list.forEach(System.out::println);
```

```
1  @Test
2  void test3() {
3   List<Integer> list = Arrays.asList(10, 20, 30, 40);
4   for (Integer a : list) {
5   System.out.println(a+10);
6   }
7   //Stream:高级版迭代器,
8   list.stream().map(x->x+10).forEach(System.out::println);//映射
9   list.stream().filter(x->x>10).forEach(System.out::println);//过滤
```

}

五.对接口做了扩展:static,default

接口中可以添加static方法

接口的方法如果为default,可以给方法一个默认实现,实现类这个接口的类,不强制重写default

接口fly

```java
public interface Fly {
    void fly();
    default void stop(){//默认给他一个方法.不需要重写
        System.out.println("停");
    }
    static int max(int a,int b) { return a>b?a:b; }
}
```

测试框架中

```java
        @Override
        public void fly() {
            System.out.println("飞");
        }
    };
    fly.fly();
    fly.stop();

    Fly fly1=()-> System.out.println("飞飞");//只能写一个方法
    fly1.fly();
    fly1.stop();

    Fly fly3=()-> {
        System.out.println("飞飞");
        System.out.println("啦啦啦");
    };
    fly3.fly();
    fly3.stop();
```

```java
//使用匿名内部类的写法,完成创建两个线程,并在线程中打印"Hello"
    @Test
    void test4() {
        Thread thread1=new Thread(){
            @Override
            public void run() {
                System.out.println("Hello");
            }
        };
        thread1.start();

        Runnable runnable=new Runnable() {
            public void run() { System.out.println("hello"); }
        };
        Thread thread2=new Thread(runnable);
        thread2.start();
      new Thread(new Runnable() {
          public void run() { System.out.println("hello"); }
      }).start();

    }
```