

一、实验名称

第5次上机-第9、10、11章内容

二、实验内容

第9章： page 76, 基础训练1: 异常捕获处理。

目标: 理解什么是异常，异常处理机制的执行特点。

第9章： page 77, 基础训练2: 自定义异常的定义与抛出。

目标: 理解自定义异常的定义与抛出方式、捕获处理。

第10章： page 84, 基础训练1: 绘制同心圆。

目标: 熟悉如何在不同图形部件中进行图形绘制。

第10章： page 86, 基础训练2: 绘制有重叠区域的不同颜色的圆。

目标: 理解覆盖绘图方式和异或绘图方式的差异

第11章： page93, 基础训练1: 单击按钮随机改变窗体的背景颜色。

目标: 了解图形界面的布局、事件处理特点

第11章： page94, 基础训练2: 在一个画布中绘制圆，通过窗体中按钮控制圆的颜色随机变化

目标: 了解如何在两个类之间传递信息

三、实验环境

调试工具: Visual Studio Code Version: 1.98.0 Commit:

6609ac3d66f4eade5cf376d1cb76f13985724bcb OS: Ubuntu 24.04.1 LTS

服务配置: openjdk 21.0.6 2025-01-21 OpenJDK Runtime Environment (build 21.0.6+7-Ubuntu-124.04.1) OpenJDK 64-Bit Server VM (build 21.0.6+7-Ubuntu-124.04.1, mixed mode, sharing)

四、实验设计

第9章：page 76, 基础训练1

(1) 设有一个数组存储一批英文单词，从键盘输入一个数n，输出数组中元素序号为n的单词。

按照实验指导书上 `ExceptionTest.java` 编码，编译运行，依次尝试 `0 1 2 3 4 5 -1 a` 得到不同的输出结果如下

```

@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
good
@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
bad
@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
ok
@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
bye
@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
    at Lab05.ExceptionTest.main(ExceptionTest.java:10)
@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 4
    at Lab05.ExceptionTest.main(ExceptionTest.java:10)
@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 4
    at Lab05.ExceptionTest.main(ExceptionTest.java:10)
@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
Exception in thread "main" java.lang.NumberFormatException: For input string: "a"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:588)
    at java.base/java.lang.Integer.parseInt(Integer.java:685)
    at Lab05.ExceptionTest.main(ExceptionTest.java:9)
@29701 on E:/Code/Java/HW/src/Lab05
#
  
```

因为word数组长度只有4，索引从0 ~ 3，所以当输入为 `4 5 -1` 时超出索引范围造成报错：out of bounds

`Integer.parseInt()` 函数无法将字符串 `a` 转换为整数。因为 `a` 不是一个数字。当 `Integer.parseInt()` 无法解析输入字符串为整数时，它会抛出一个 `NumberFormatException` 异常。

(2) 为了控制异常的报错处理，利用 `try...catch` 进行异常处理。

按照书上增加异常处理的代码，编译运行后尝试使用 `-1` 和 `a` 作为输入，可以看到终端输出正确地抛出异常

```

@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
要求输入整数
@29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
数组访问出界
  
```

(3) 将以上两个catch部分内容删除，改用一个catch，其中，捕获的异常为Exception类，观察程序的运行变化。

按照书上将两个catch部分内容换为一个 `catch (Exception e)` 之后再次尝试使用 `-1` 和 `a` 作为输入, 两次终端输出结果均为 出现异常

所以 `ArrayIndexOutOfBoundsException` 和 `NumberFormatException` 异常类都应当是继承于 `Exception`

(4) 在程序的异常处理代码中, 加入finally部分, 检查其代码在什么情况下执行。

在catch后增加了finally块之后编译运行, 尝试使用正常的 `0` 和异常的 `-1` 和 `a` 作为输入, 发现均执行了finally块

```
⑧ 29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
good
执行了finally块
⑧ 29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
出现异常
执行了finally块
⑧ 29701 on E:/Code/Java/HW/src/Lab05
# java ExceptionTest.java
出现异常
执行了finally块
⑧ 29701 on E:/Code/Java/HW/src/Lab05
#
```

(5) 异常排序问题。

```
import javax.swing.*;

public class ExceptionTest {
    public static void main(String[] args) {
        try {
            String word[] = {"good", "bad", "ok", "bye"};
            String s = JOptionPane.showInputDialog("请输入一个数: ");
            int n = Integer.parseInt(s);
            System.out.println(word[n]);
        } catch (Exception e) {
            System.out.println("出现异常");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("数组访问出界");
        } catch (NumberFormatException e) {
            System.out.println("要求输入整数");
        } finally {
            System.out.println("执行了finally块");
        }
    }
}
```

如果将 `Exception` 异常放在最前面在尝试编译时会报错如下

```
ExceptionTest.java:20: error: exception ArrayIndexOutOfBoundsException has already
been caught
        } catch (ArrayIndexOutOfBoundsException e) {
            ^
ExceptionTest.java:22: error: exception NumberFormatException has already been caught
        } catch (NumberFormatException e) {
            ^
2 errors
error: compilation failed
```

因为所有的异常都会首先与 `Exception` 匹配, 然后到最后的 `finally` 块, 结束 `try-catch`。不会经过中间的其他异常捕获

第9章：page 77, 基础训练2

(1) 自己定义一个异常并抛出。

```
public class MyException extends Exception {  
    public String toString() {  
        return "异常啦";  
    }  
    public static void main(String[] args) {  
        throw new MyException();  
    }  
}
```

尝试编译以上代码, 报错如下

```
MyException.java:8: error: unreported exception MyException; must be caught or  
declared to be thrown  
    throw new MyException();  
        ^  
1 error  
error: compilation failed
```

原因在于 `MyException` 这种异常并没有被定义过

(2) 增加 try...catch 代码。

```
public class MyException extends Exception {  
    public String toString() {  
        return "异常啦";  
    }  
    public static void main(String[] args) {  
        try {  
            throw new MyException();  
        } catch (MyException e) {  
            System.out.println(e);  
        }  
    }  
}
```

根据书上的代码尝试编译运行, 终端输出 `异常啦`

抛出和捕获自定义异常的步骤:

1. **定义异常类:** 创建一个继承 `Exception` 的自定义异常类。可以重写 `toString()` 方法, 提供自定义的异常描述信息。

2. **抛出异常：** 使用 `throw new MyException()` 语句在 `try` 块中抛出异常实例。创建异常对象时，可以将错误信息传递给异常类的构造函数，以便在 `toString()` 方法中使用。
3. **捕获异常：** 使用 `catch (MyException e)` 块捕获特定类型的异常。在 `catch` 块中，通过异常对象 `e` 访问异常信息，例如调用 `e.toString()` 获取自定义异常描述，并执行相应的错误处理。

(3) 在方法头声明方法中可能产生异常。

```
public class MyException extends Exception {  
    public String toString() {  
        return "异常啦";  
    }  
    public static void main(String[] args) throws MyException {  
        throw new MyException();  
    }  
}
```

以上代码编译运行终端输出如下

```
Exception in thread "main" 异常啦  
    at Lab05.MyException.main(MyException.java:13)
```

总结方法异常声明流程：

1. **识别潜在异常：** 分析方法中的代码，明确可能引发异常的情况
2. **评估处理：** 确定方法内部是否能够安全有效地处理这些异常。
3. **声明异常：** 如果方法无法处理异常，则使用 `throws` 关键字在方法声明中声明异常类型

(4) 对声明异常的方法调用。

```
public class MyException extends Exception {  
    public String toString() {  
        return "异常啦";  
    }  
    public static void main(String[] args) throws MyException {  
        method();  
    }  
  
    public static void method() throws MyException {  
        throw new MyException();  
    }  
}
```

以上代码在终端中编译运行输出结果如下

```
Exception in thread "main" 异常啦  
    at Lab05.MyException.method(MyException.java:18)  
    at Lab05.MyException.main(MyException.java:14)
```

```
public class MyException extends Exception {  
    public String toString() {  
        return "异常啦";  
    }  
    public static void main(String[] args) throws MyException {  
        try {  
            method();  
            System.out.println("这里执行不到");  
        } catch (MyException e) {  
            System.out.println(e);  
        }  
        System.out.println("这里要执行");  
    }  
  
    public static void method() throws MyException {  
        throw new MyException();  
    }  
}
```

以上代码执行结果如下

```
异常啦  
这里要执行
```

异常处理流程对执行流程影响：

1. **异常抛出:** `throw` 语句引发异常，中断正常执行。
2. **异常捕获捕获:** `try-catch` 块：捕获并处理异常。
3. **流程控制:**
 - 未处理：异常向上层传递，最终可能导致程序终止。
 - 已处理：`catch` 块执行，之后程序继续执行。

第10章： page 84, 基础训练1

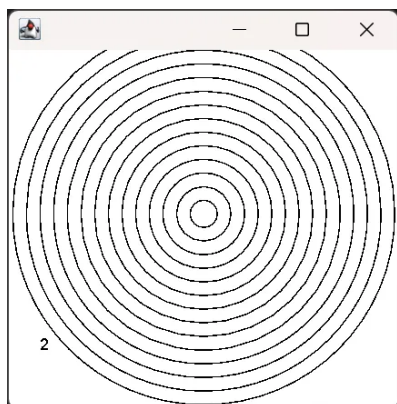
(1) 在窗体内绘制同心圆。

```
import java.awt.*;

public class MyFrame extends Frame {
    public void paint (Graphics g) {
        int x = getWidth();
        int y = getHeight();
        int w = Math.min(x, y);
        g.drawString(""+2, 30, 250);
        for (int k = 1; k < w / 10; k++) {
            g.drawOval(10 * k, 10 * k, w - 20 * k, w - 20 * k);
        }
    }

    public static void main(String[] args) {
        Frame x = new MyFrame();
        x.setSize(300, 300);
        x.setVisible(true);
    }
}
```

以上代码运行得到以下的窗口



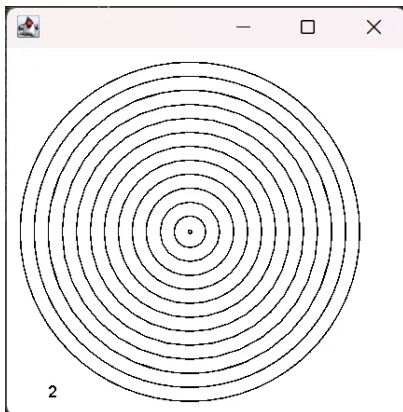
(2) 在画布上绘制同心圆。

```
import java.awt.*;

public class MyCanvas extends Canvas {
    public void paint (Graphics g) {
        int x = getWidth();
        int y = getHeight();
        int w = Math.min(x, y);
        g.drawString(""+2, 30, 250);
        for (int k = 1; k < w / 10; k++) {
            g.drawOval(10 * k, 10 * k, w - 20 * k, w - 20 * k);
        }
    }

    public static void main(String[] args) {
        Frame x = new MyFrame();
        x.add(new MyCanvas());
        x.setSize(300, 300);
        x.setVisible(true);
    }
}
```

以上代码绘制出窗体为



```
import java.awt.*;

public class MyCanvas2 extends Canvas {
    public void paint (Graphics g) {
        // g.setXORMode(getBackground());
        g.setColor(Color.red);
        g.fillOval(20,20,80,80);
        // g.setColor(Color.red);
        // g.fillOval(20,20,80,80);
        g.setColor(Color.blue);
        g.fillOval(80, 40, 80, 80);
        g.setColor(Color.green);
        g.fillOval(40, 50, 80, 80);
    }

    public static void main(String[] args) {
        Frame x = new Frame();
        x.add(new MyCanvas2());
        x.setSize(180, 180);
        x.setVisible(true);
    }
}
```

打开以上代码不同的注释得到以下三个结果

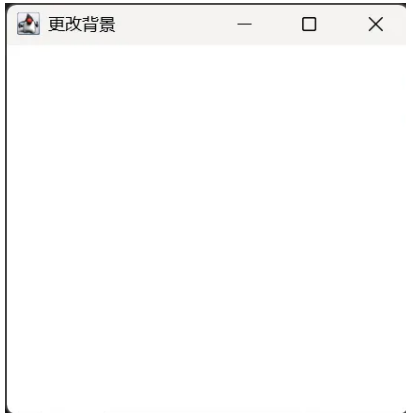


如果是异或绘图方式下在红色圆处再绘制一个红色圆会和不是异或模式下绘制的图像相同, 因为经过了两次异或变成原样.

第11章: page93, 基础训练1

(1) 创建窗体并让窗体可见

书上代码编译运行出现以下窗口



如果讲 `setVisible(true);` 注释掉, 则将不会有任何窗口出现, 但在终端里这个程序依旧在运行.

(2) 在窗体中添加一个按钮。

- `super("更改背景")` 为窗体增加了标题
- `setLayout(new FlowLayout());` 设置了按钮的大小位置
- `Button btn = new Button("更改背景");` 创建了一个按钮对象, 文字内容为"更改背景"
- `add(btn)` 将刚才的创建的按钮对象添加到窗体中

此时单击按钮在终端没有任何输出, 窗体也仅仅只是有按钮有点击的动画

(3) 给按钮注册事件监听者。

添加事件监听后的整体代码如下

```
import java.awt.event.*;
import java.awt.*;

public class ChangeColor extends Frame implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Color c = new Color((int)(Math.random()*256), (int)(Math.random()*256), (int)
(Math.random()*256));
        setBackground(c);
    }

    public ChangeColor() {
        super("更改背景");
        setLayout(new FlowLayout());
        Button btn = new Button("更改背景");
        btn.addActionListener(this);
        add(btn);
        setSize(300,300);
        setVisible(true);
    }

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

此时编译运行, 点击窗体中的按钮, 每次点击窗体都会更改一次背景颜色

【思考】修改事件的监听者, 采用内嵌类或匿名内嵌类实现

1. 使用内嵌类

```
import java.awt.event.*;
import java.awt.*;

public class ChangeColor extends Frame {
    public ChangeColor() {
        super("更改背景");
        setLayout(new FlowLayout());
        Button btn = new Button("更改背景");

        // 使用内嵌类作为 ActionListener
        ColorChanger colorChanger = new ColorChanger();
        btn.addActionListener(colorChanger); // 关联事件源 (按钮) 与监听器

        add(btn);
        setSize(300,300);
        setVisible(true);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0); // 关闭窗口时退出程序
            }
        });
    }

    // 内嵌类, 实现了 ActionListener 接口
    class ColorChanger implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            Color c = new Color((int)(Math.random()*256), (int)(Math.random()*256),
(int)(Math.random()*256));
            setBackground(c);
        }
    }

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

2. 使用匿名内嵌类

```

import java.awt.event.*;
import java.awt.*;

public class ChangeColor extends Frame {
    public ChangeColor() {
        super("更改背景");
        setLayout(new FlowLayout());
        Button btn = new Button("更改背景");

        // 使用匿名内嵌类作为 ActionListener
        btn.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                Color c = new Color((int)(Math.random()*256), (int)
(Math.random()*256), (int)(Math.random()*256));
                setBackground(c);
            }
        }); // 注意这里需要加分号，因为这实际上是一个语句

        add(btn);
        setSize(300,300);
        setVisible(true);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0); // 关闭窗口时退出程序
            }
        });
    }

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

```

第11章： page94, 基础训练2

(1) 编写一个个性化的画布，将其作为MyFrame的属性， 字样可方便在MyFrame窗体中访问和操纵画布对象

```
import java.awt.*;
import java.awt.event.*;

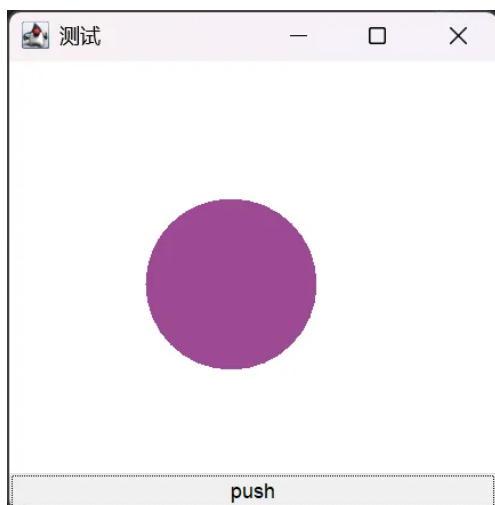
public class MyFrame2 extends Frame implements ActionListener {
    MyCanvas3 a = new MyCanvas3();
    public MyFrame2(String str) {
        super(str);
        this.add("Center", a);
        Button b = new Button("push");
        add("South", b);
        b.addActionListener(this);
        setSize(300, 300);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        a.c = new Color((int)(Math.random() * 0xffffffff));
        a.repaint();
    }

    public static void main(String[] args) {
        new MyFrame2("测试");
    }
}

class MyCanvas3 extends Canvas {
    Color c;

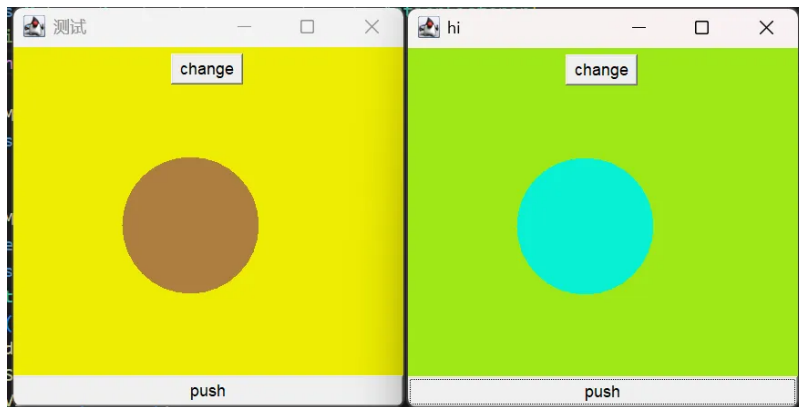
    public void paint(Graphics g) {
        g.setColor(c);
        g.fillOval(80, 80, 100, 100);
    }
}
```



将 `MyCanvas3 extends Canvas` 改成 `MyCanvas3 extends Button` 可以发现圆所在的位置变成了一个按钮

(2) 将画布改成面板，在面板中加入一个按钮，由按钮控制面板颜色随机变化

按照书中参考程序，得到两个窗体，两个窗口中的 `change` 按钮可以改变背景的颜色，下方的 `push` 按钮可以改变圆形的颜色。



```
import java.awt.*;
import java.awt.event.*;

public class MyFrame3 extends Frame implements ActionListener{
    static My a = new My();

    public MyFrame3() {
        this("hi");
    }

    public MyFrame3(String str) {
        super(str);
        this.add("Center", a);
        Button b = new Button("push");
        add("South", b);
        b.addActionListener(this);
        setSize(300, 300);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        a.c = new Color((int)(Math.random() * 0xffffffff));
        a.repaint();
    }

    public static void main(String[] args) {
        new MyFrame3("测试");
        new MyFrame3();
    }
}

class My extends Panel implements ActionListener {
    Color c;

    public My() {
        Button b1 = new Button("change");
        add(b1);
        b1.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        c = new Color((int)(Math.random() * 0xffffffff));
        setBackground(c);
    }

    public void paint(Graphics g) {
        g.setColor(c);
        g.fillOval(80, 80, 100, 100);
    }
}
```

```
}  
}
```

思考：将 `MyFrame3` 中的 `a` 属性改成 `static My a = new My();` 可以再次尝试编译运行，发现成功通过编译。然而在 `测试` 窗体中没有了圆和 `change` 按钮，而且 `测试` 窗体中的 `push` 按钮可以修改 `hi` 窗体中圆的颜色，这在修改之前是做不到的。`hi` 窗体中一切正常

`static` 修饰符在这里的作用

- 将 `a` 声明为 `static` 后，所有 `MyFrame3` 实例共享同一个 `My` 对象。
- 这导致多个窗体（如 `hi` 和 `测试`）操作的是同一个 `a` 对象。
- `测试` 窗体中没有独立的圆和 `change` 按钮，因为 `a` 已被添加到第一个窗体中。
- `测试` 窗体中的 `push` 按钮可以修改 `hi` 窗体中圆的颜色，因为两者共享同一个 `a`。

此处不适合添加`static`关键字。

- 每个窗体应有独立的 `My` 对象以确保内容独立显示。
- 使用 `static` 导致多个窗体共享同一个对象，违背了设计意图，并引发功能异常。

五、实验心得

通过本次实验，我深入学习了Java异常处理机制、图形界面编程基础和事件处理模型，获得了以下收获：

异常处理机制

- 异常捕获流程：掌握了try-catch-finally结构的执行顺序，理解了异常处理对程序流程的影响
- 异常继承体系：明确了Java异常的层次结构，如`ArrayIndexOutOfBoundsException`和`NumberFormatException`都继承自`Exception`
- 异常匹配规则：体会到异常捕获遵循"从特殊到一般"的原则，父类异常应放在后面捕获
- 自定义异常：学会了创建自定义异常类并通过throws声明或try-catch处理的方式

图形界面绘制

- 组件绘制机制：理解了`paint(Graphics g)`方法在各种容器中的作用及重写方式
- 画布与窗体：掌握了在`Canvas`和`Frame`中进行图形绘制的区别
- 绘图坐标系统：熟悉了Java 2D坐标系统和基本绘图方法如`drawOval`、`fillOval`等

- 绘图模式差异：体验了覆盖绘图与异或绘图(XORMode)的不同效果和应用场景

事件处理模型

- 事件监听机制：理解了Java事件驱动编程模型中的事件源、事件监听器和事件处理方法的关系
- 接口实现方式：掌握了通过实现ActionListener接口来响应用户交互事件
- 内嵌类监听器：学习了使用内嵌类和匿名内嵌类两种方式实现事件监听，提高了代码组织灵活性
- 组件间通信：理解了如何通过类的属性关系实现不同组件之间的信息传递和交互

遇到的问题

在尝试修改事件监听者，采用内嵌类或匿名内嵌类实现的时候，没有思考清楚创建匿名内嵌类的关系，导致没有成功创建对象

对此我做出以下总结

匿名类的用法

- **定义**：匿名类是没有名字的类，通常用于实现接口或继承抽象类。
- **适用场景**：适用于需要一次性使用的简单逻辑实现
- **例子**：在事件监听器中使用匿名类。

```
btn.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("按钮被点击");  
    }  
});
```

匿名内嵌类的用法

- **定义**：匿名内嵌类是定义在另一个类内部的匿名类，它可以访问外部类的所有成员（包括私有成员）。
- **适用场景**：适用于需要访问外部类上下文的一次性任务。
- **例子**：在包含类内部定义一个匿名内嵌类作为事件监听器。

```
public class OuterClass {
    public OuterClass() {
        Button btn = new Button("点击我");
        btn.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("按钮被点击! " + OuterClass.this.someMethod());
            }
        });
    }

    private String someMethod() {
        return "Hello from OuterClass";
    }
}
```