



**Vorlesung  
am FB Informatik  
der Universität Oldenburg**

**Vorlesung 12**

**Dietrich Boles**

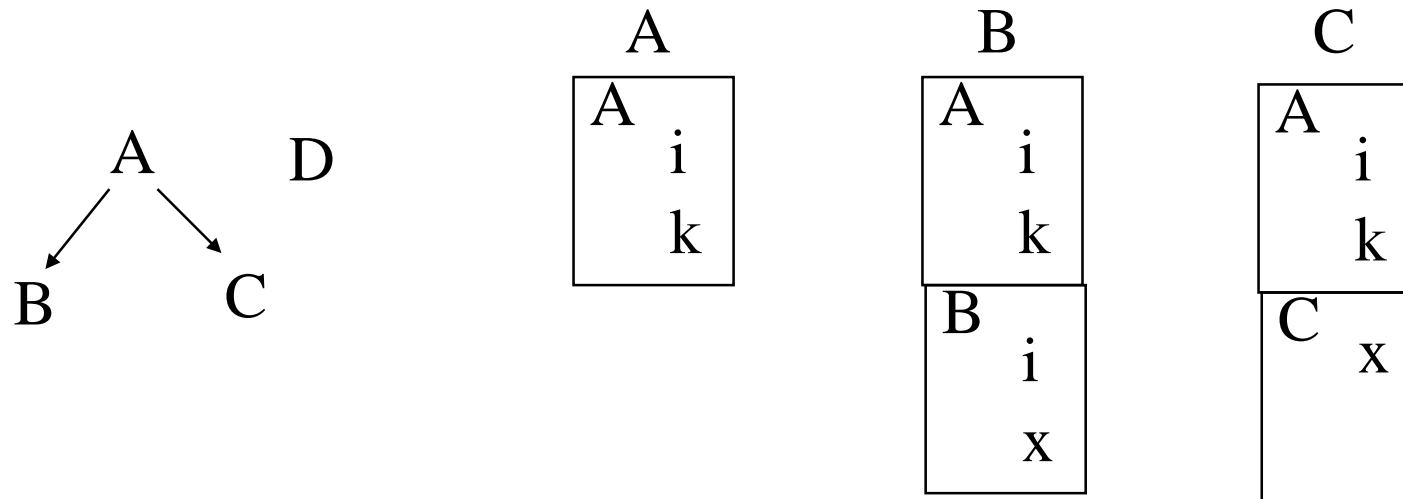
- Wiederholung
- Abstrakte Klassen
  - Motivation
  - Definition
  - Beispiel
- Interfaces
  - Definition
  - Beispiel
  - Vergleich zu abstrakten Klassen
- Subobjekte
  - Beispiel
  - Komposition / Aggregation
  - Delegation
  - „Mehrfachvererbung“
- Beispiele

```
class A {  
    int i, k;  
    public void f() { System.out.println(this.i + this.k); }  
}
```

```
class B extends A {  
    int i, x;  
    public void f() {  
        System.out.println(this.i + this.x + this.k + super.i);  
    }  
    public void g() { ... }  
}
```

```
class C extends A {  
    int x;  
    public void f() {  
        System.out.println(this.x + this.i + this.k);  
    } }  
}
```

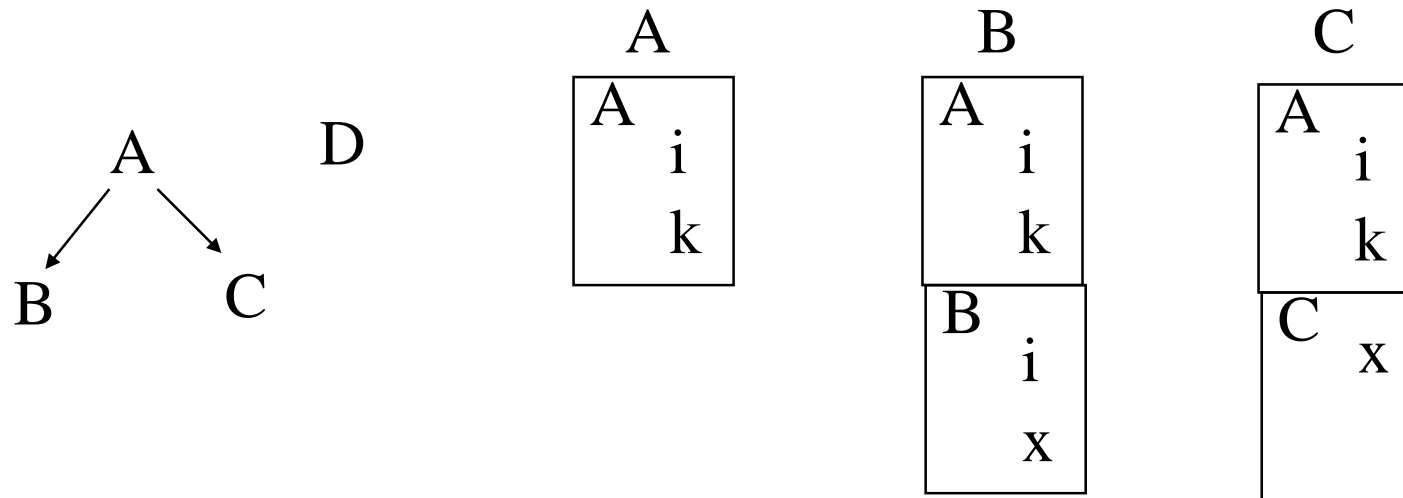
# Wiederholung / Vererbung und Polymorphie



```

A a = new A();
A b = new B();
A c = new C();
A d = new D(); // geht nicht!
B d = new C(); // geht nicht!
  
```

# Wiederholung / Vererbung und Typ-Cast



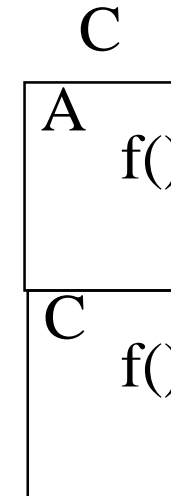
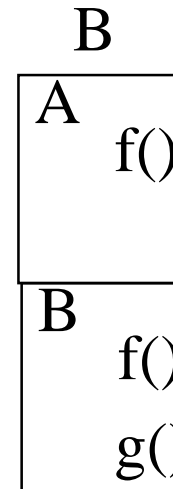
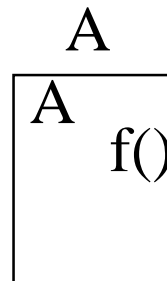
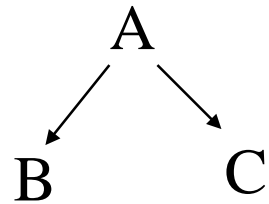
```
A a = new B();
a.i = 3;
B b = new B();
b.i = 4;
```

```
((B)a).i = 5; // B b2 = (B)a; b2.i = 4;
```

```
((C)a).x = 5; // Laufzeitfehler
```

```
((D)d).x = 6; // Compilefehler
```

# Wiederholung / Polymorphie und Dyn. Binden



```

A a = new A();
A b1 = new B();
B b2 = new B();
A c = new C();
  
```

```

a.f();
b1.f();
b1.g(); // geht nicht
((B)b1).g();
b2.g();
  
```

# Abstrakte Klassen / Motivation

---

- Prinzip der Vererbung:
  - Aus existierenden Klassen können spezialisierte Klassen abgeleitet werden
- Prinzip der abstrakten Klassen:
  - Aus mehreren ähnlichen Klassen kann eine gemeinsame Oberklasse abstrahiert werden
  - Sinn und Zweck: Ausnutzung der Polymorphie!
- Beispiel:

```
public class Rechteck { void draw() {...} ...}  
public class Kreis   { void draw() {...} ...}  
public class Linie   { void draw() {...} ...}
```

```
public abstract class Graphik { void draw(); }
```

# Abstrakte Klassen / Definition

---

- Definition: Abstrakte Klasse
  - Eine abstrakte Klasse ist eine bewusst **unvollständige** Oberklasse, in der von einzelnen Methodenimplementierungen abstrahiert wird (**„abstrakte“ Methoden!**)
  - Fehlende Methodenrumpfe werden erst in abgeleiteten Unterklassen implementiert.
  - Die Instantiierung abstrakter Klassen ist nicht möglich.
  - Java: Schlüsselwort **abstract**

```
public abstract class Graphik {  
    public abstract void draw();  
}
```

## Abstrakte Klassen / Beispiel

---

```
public abstract class Graphik {
    String name;
    public Graphic(String str) { this.name = str; }
    public String getName() { return this.name; }
    public abstract void draw();
}

public class Rechteck extends Graphik {
    float width, height;
    public Rechteck(String str, float w, float h) {
        super(str); this.width = w; this.height = h;
    }
    // geerbt: getName
    // weitere Methoden ...
    // implementiert:
    public void draw() {
        System.out.println("Rechteck:" + this.name);
    } }
}
```

## Abstrakte Klassen / Beispiel

---

```
public class Kreis extends Graphik {
    float radius;
    public Kreis(String str, float r) {
        super(str); this.radius = r;
    }
    // geerbt: getName
    // weitere Methoden ...
    // implementiert:
    public void draw() {
        System.out.println("Kreis:" + name + radius);
    } }
public class Linie extends Graphik {
    ...
    // implementiert:
    public void draw() {
        System.out.println("Linie:" + this.name);
    } }
```

## Abstrakte Klassen / Beispiel

---

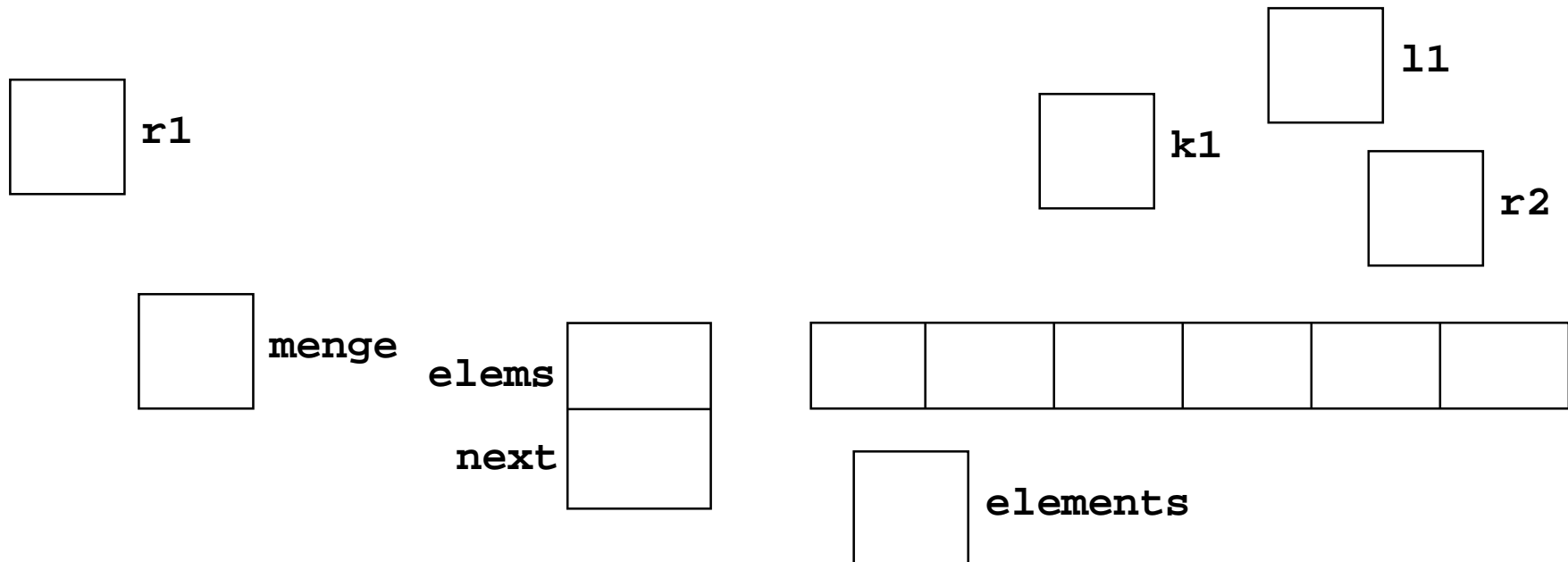
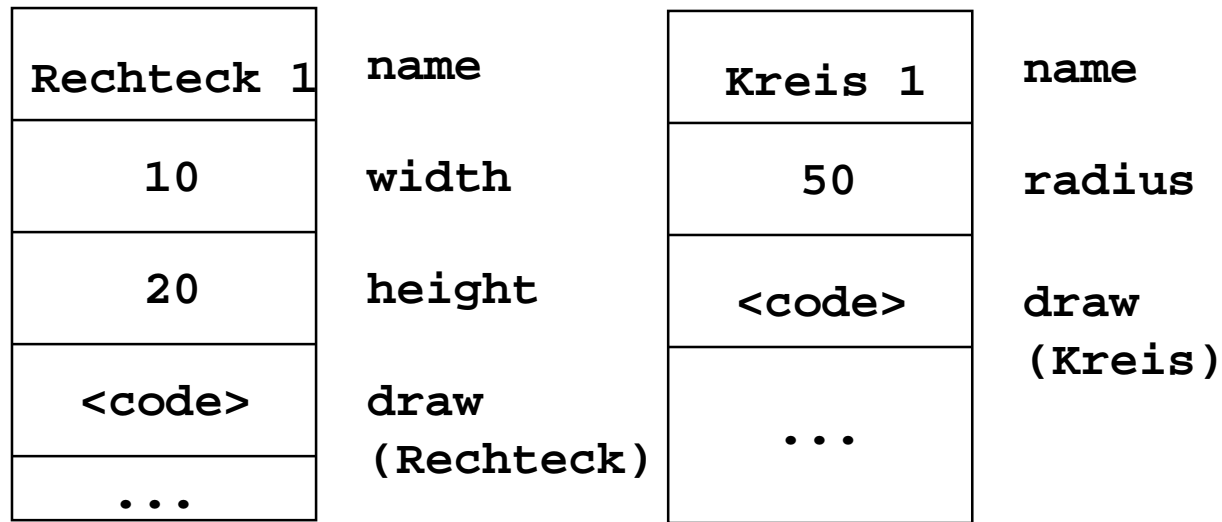
```
public class Set {
    Graphic[] elems;
    int next;
    public Set(int size) {
        this.elems = new Graphic[size];
        this.next = 0;
    }
    public void add(Graphic obj) {
        if (this.next < this.elems.length)
            this.elems[this.next++] = obj;
    }
    public Graphic[] get() {
        return this.elems;
    } }
}
```

## Abstrakte Klassen / Beispiel

---

```
public class Probe {
    public static void main(String[] args) {
        Rechteck r1 = new Rechteck("Rechteck 1", 10, 20);
        Kreis k1 = new Kreis("Kreis 1", 50);
        Linie l1 = new Linie("Linie 1", 40);
        Rechteck r2 = new Rechteck("Rechteck 2", 15, 15);
        Graphik g1 = new Graphik("Graphik 1"); // Fehler!
        Set menge = new Set(6);
        menge.add(r1);    // Ausnutzung der Polymorphie!
        menge.add(r2);
        menge.add(k1);
        menge.add(l1);
        Graphic[] elements = menge.get();
        for (int i=0; i<elements.length; i++)
            if (elements[i] != null)
                elements[i].draw(); // Dynamisches Binden!
    } } }
```

# Abstrakte Klassen / Beispiel / Speichermodell



## Interfaces / Definition

---

- Definition: Interface
  - Ein Interface ist eine Art Klasse, die ausschließlich Konstanten und abstrakte Instanzmethoden deklariert!
  - Java: Schlüsselwörter **interface** und **implements**
  - Sinn: Simulation von Mehrfachvererbung

```
public interface Graphik {  
    public void draw();  
}  
public class Linie implements Graphik {  
    public void draw() {  
        System.out.println(...);  
    }  
}
```

## Interfaces / Beispiel

---

```
public interface Landfahrzeug {
    public void fahren();
}
public interface Wasserfahrzeug {
    public void schwimmen();
}
public class Fahrzeug {
    Motor m;
    ...
}
public class PKW
    extends Fahrzeug
    implements Landfahrzeug {
    public void fahren() { ... }
}
```

## Interfaces / Beispiel

---

```
public class MotorBoot
    extends Fahrzeug
    implements Wasserfahrzeug {
        public void schwimmen() { ... }
    }

public class Amphibienfahrzeug
    extends Fahrzeug
    implements Landfahrzeug, Wasserfahrzeug {
        public void fahren() { ... }
        public void schwimmen() { ... }
    }
```

main:

```
<X> f = new Amphibienfahrzeug();
```

X kann sein (polymorph zu):

Amphibienfahrzeug / Fahrzeug / Object /  
Landfahrzeug / Wasserfahrzeug

# Abstrakte Klassen - Interfaces / Vergleich

---

- Abstrakte Klassen:
  - abgeleiteten Klassen soll bereits ein bestimmtes Grundverhalten zur Verfügung gestellt werden (→ Vererbung)
  - (Einfach-)Polymorphie
  
- Interfaces:
  - ausschließliche Definition des Protokolls
  - (Mehrfach-)Polymorphie

```
public class Gehirn {  
    public void abspeichern(Object info) { ... }  
    public Object denken() { ... }  
}
```

```
public class Herz {  
    public void pumpen(int schlaege) { ... }  
}
```

```
public class Arm {  
    public void anwinkeln(float grad) { ... }  
    public void heben(Object gewicht) { ... }  
}
```

## Subobjekte / Beispiel

```

public class Mensch {
    Herz herz;           // Subobjekt
    Gehirn gehirn;      // exklusives Subobjekt
    Arm[] arme;         // mehrere Subobjekte
    public Mensch() {
        this.gehirn = new Gehirn();
        this.herz = new Herz();
        this.arme = new Arm[2];
        this.arme[0] = new Arm();
        this.arme[1] = new Arm();
    }
    public Object denken() {
        return this.gehirn.denken();
    }
    public void sportTreiben() {
        this.herz.pumpen(180);
    }
    public void schlafen() {
        this.herz.pumpen(50);
    }
}

```

- Definition: **Subobjekt**
  - ein Attribut vom Typ einer Klasse
- Definition: **Komposition / Aggregation**
  - Zusammensetzung eines Objektes aus mehreren Subobjekten
  - „part-of-Beziehung“
  - (Vererbung: „is-a-Beziehung“)
- Definition: **Delegation**
  - Prinzip der Implementierung einer Methode durch Weiterreichen eines Methodenaufrufs an ein Subobjekt

## Subobjekte / „Mehrfachvererbung“

---

```
public class Landfahrzeug {
    public void fahren() { ... }
}
public class Wasserfahrzeug {
    public void schwimmen() { ... }
}
public class Amphibienfahrzeug {
    Landfahrzeug land;           // Subobjekte
    Wasserfahrzeug wasser;
    ...
    public void fahren() {
        this.land.fahren();      // Delegation
    }
    public void schwimmen() {
        this.wasser.schwimmen(); // Delegation
    } }
}
```

## Subobjekte / „Mehrfachpolymorphie“

```

public class A { public void f() { ... } ... }
public class B { public void g() { ... } ... }
public class C extends A, B { ... } // geht nicht!
public class D extends B { // Hilfsklasse
    C cobj;
    public D(C c) { this.cobj = c; }
    public void g() { ... } ...
}
public class C extends A {
    D dobj;
    public C() { this.dobj = new D(this); }
    public void f() { ... }
    public B getObject() { return dobj; }
}
// Polymorphie von C zu A
A a = new C(); a.f(); ...
// Polymorphie von C zu B
B b = new C().getObject(); b.g(); ...

```

## Interfaces und Polymorphie / Beispiel 1

---

```
public interface Funktion {
    // Berechnung von f(x)
    public double getY(double x);
}

public class Math {
    public static
        double nullstelle(Funktion f,
                           double x_i, double x_im1) {
        // Iterationsverfahren ...
        double x_ip1 = x_i - (f.getY(x_i)*(x_im1-x_i)/
                              (f.getY(x_im1)-f.getY(x_i)));
        ...
        return x_ip1;
    }
    public static sqrt(double x) { ... }
    ...
}
```

## Interfaces und Polymorphie / Beispiel 1

---

```
class F1 implements Funktion {
    public double getY(double x) {
        return x*x - 1;
    }
}
class F2 implements Funktion {
    public double getY(double x) {
        return x*x*x + 0.9*x*x - 3.1*x + 2.3;
    }
}
public class Berechnungen {
    public static void main(String[] args) {
        double n = Math.nullstelle(new F1(), 2, 3);
        System.out.println(n);
        Funktion f;
        n = Math.nullstelle(f = new F2(), 4, 5);
        System.out.println(n);
        n = Math.nullstelle(f, -3, -4);
    }
}
```

# Interfaces und Polymorphie / Beispiel 1

---

```
// Nutzung von anonymen Klassen
```

```
public class Berechnungen {  
    public static void main(String[] args) {  
        double n = Math.nullstelle(  
            new Funktion() // anonyme Klasse  
                public double getY(double x) {return x*x-1;}  
            }, 2, 3);  
        System.out.println(n);  
        n = Math.nullstelle(  
            new Funktion() // anonyme Klasse  
                public double getY(double x) {return 3*x*x-2;}  
            }, 2, 3);  
        System.out.println(n);  
    } }  
}
```

```
// Auflistung mehrerer Objekte (für Speicherklassen)
public interface Enumeration {

    // sind noch weitere Elemente vorhanden
    public boolean hasMoreElements();

    // liefere naechstes Element
    public Object nextElement();
}
```

## Interfaces und Polymorphie / Beispiel 2

```

public class Vector { // wachsendes Array
    Object[] elems;      int noof_elems;
    ...
    public void add(Object obj) { ... }
    public Enumeration elements() {
        return new VectorEnumerator(this);
    }
}
class VectorEnumerator implements Enumeration {
    Vector vector;      int count;
    VectorEnumerator(Vector v){vector = v; count = 0;}
    public boolean hasMoreElements() {
        return count < vector.noof_elems;
    }
    public Object nextElement() {
        if (count < vector.noof_elems)
            return vector.elems[count++];
        return null;
    }
}

```

## Interfaces und Polymorphie / Beispiel 2

---

```
public class HashTable {
    ...
    public void add(Object key, Object obj) { ... }
    public Enumeration elements() {
        return new HashTableEnumerator(this);
    }
}
```

```
class HashTableEnumerator implements Enumeration {
    ...
    HashTableEnumerator(HashTable t) {...}
    public boolean hasMoreElements() {
        return ...
    }
    public Object nextElement() {
        if (...)
            return ...
        return null;
    }
}
```

# Interfaces und Polymorphie / Beispiel 2

---

```
public class Probe {
    public static void main(String[] args) {
        Vector vector = new Vector();
        vector.add(new String("ich"));
        vector.add(new String("du"));
        vector.add(new String("er"));
        ...
        Enumeration enum = vector.elements();
        while (enum.hasMoreElements()) {
            String str = (String)(enum.nextElement());
            System.out.println(str);
        }
    }
}
```

## Interfaces und Polymorphie / Beispiel 3

---

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

/* public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
public class Button extends Component {
    public Button(String label) { ... }
    public void setLabel(String label) { ... }
    public String getLabel() { ... }
    public void addActionListener(ActionListener l)
    ...
}
public class Frame {
    public Frame(String title) { ... }
    public void add(Component comp) { ... }
    public void setBounds(int x, int y, int w, int h)
    public void show() { ... }
} */
```

```
class ClickAction implements ActionListener {
    Button button;
    public ClickAction(Button b) {
        this.button = b;
    }
    public void actionPerformed(ActionEvent e) {
        if (this.button.getLabel().equals(" Start "))
            this.button.setLabel(" Stop ");
        else
            this.button.setLabel(" Start ");
    }
}
```

## Interfaces und Polymorphie / Beispiel 3

---

```
public class TestWindow extends Frame {  
    public static void main(String[] args) {  
        TestWindow w = new TestWindow();  
        w.setBounds(200,200,100,80);  
        w.show();  
    }  
}
```

```
Button button;
```

```
public TestWindow() {  
    super("Test Window");  
    this.button = new Button(" Start ");  
    this.button.addActionListener(  
        new ClickAction(this.button));  
    add(button);  
}  
}
```