Java Refresher
Part III

Alexandre Bergel
http://bergel.eu
21/03/2018

Goal of this lecture

Understand what *this* and *super* are and what are they good for

Understanding some of the *design rules* that govern *inheritance*

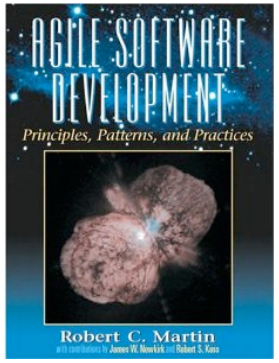
Application of *Java Interfaces*

See a *bit* of *theory*

Recommended Texts

Agile Software Development, Principles, Patterns, and Practices

Robert C. Martin “Uncle Bob”, 2002



Outline

1. This and super pseudo variables
2. Liskov principle
    1. theory
    2. concrete applications
3. Java Interfaces

Outline

1. **This and super pseudo variables**
2. Liskov principle
    1. theory
    2. concrete applications
3. Java Interfaces

Questions

What is *this*?

What is *super*?

This and Super

the *this* pseudo-variable always refers to the object receiver

the *super* pseudo-variable always refers to the object receiver

a message sent to *super* makes the lookup begins in the superclass of the class in which the call is written

The Java syntax prevents one from using super without being followed by ".identifier"

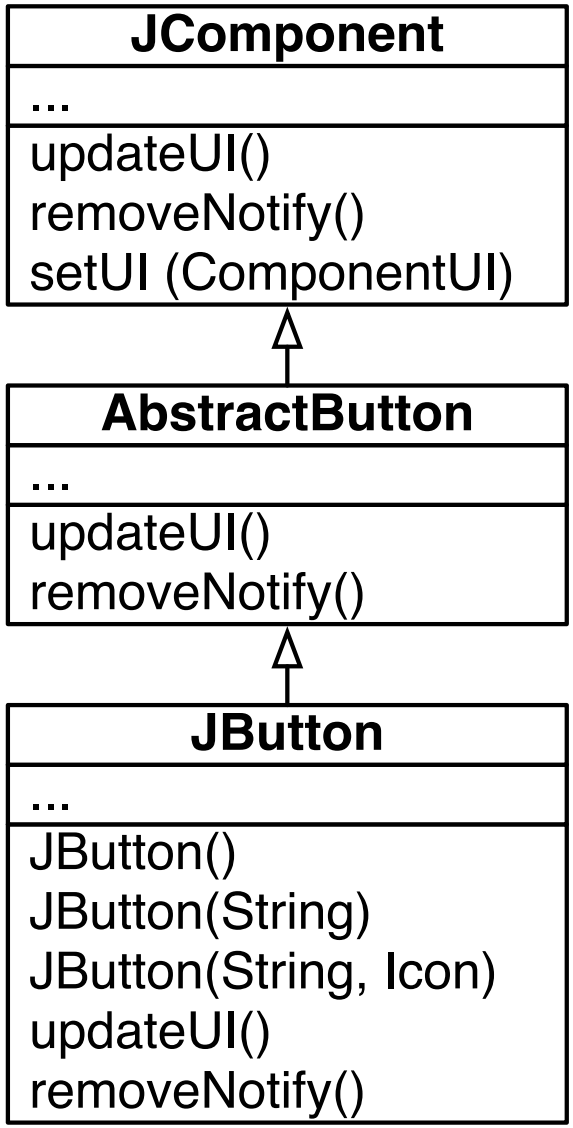
Class inheritance principle

Sending a message to an object triggers a *lookup* along the *class hierarchy* of the class of the object

In a statically typed languages (e.g., Java, C#, C++), the lookup *always* find an appropriate method

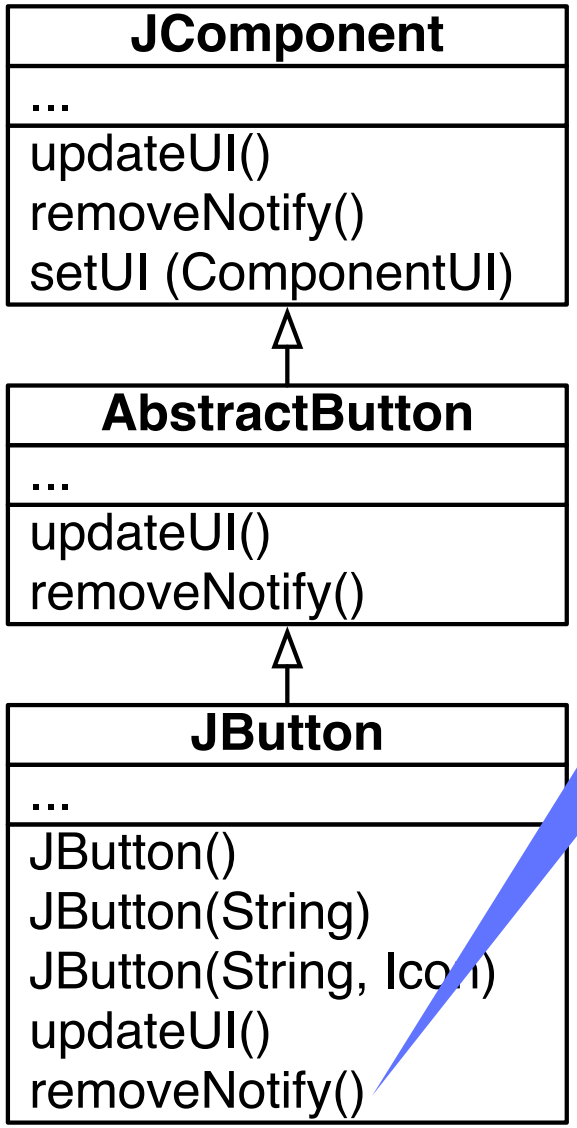
This may not be the case in a dynamically typed language (e.g., Python, Ruby, Pharo, VisualWorks, JavaScript)

Class inheritance in action!




```
JButton button = new JButton("OK");
button.removeNotify();
```

Class inheritance in action!

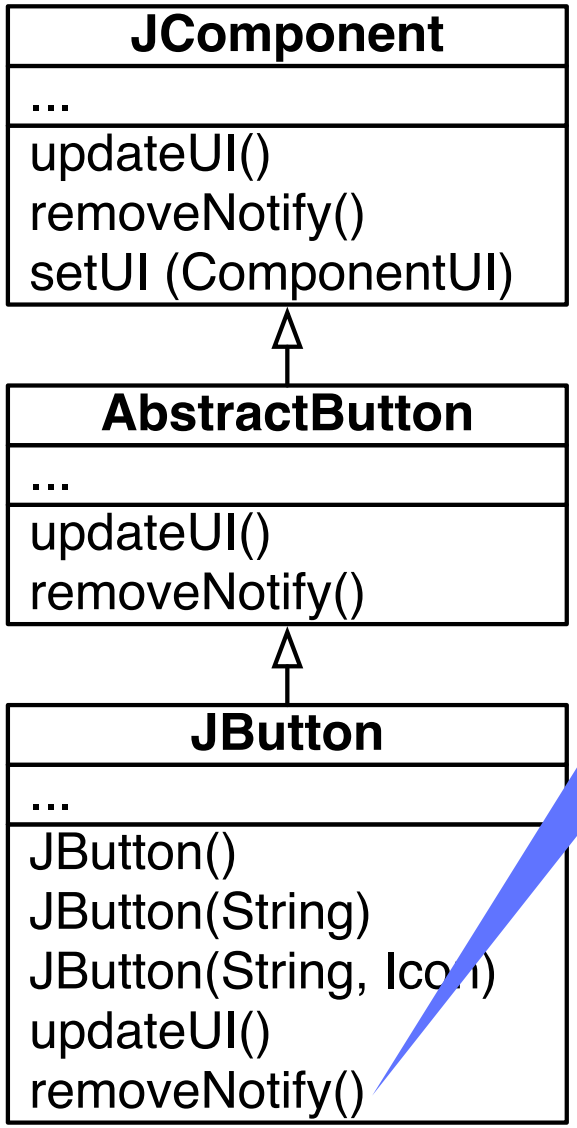



```
public void removeNotify() {
   JRootPane root =
SwingUtilities.getRootPane(this);
  if (root != null &&
root.getDefaultButton() == this){
    root.setDefaultButton(null);
  }

  super.removeNotify();
}
```

```
JButton button = new JButton("OK");
button.removeNotify();
```

Class inheritance in action!



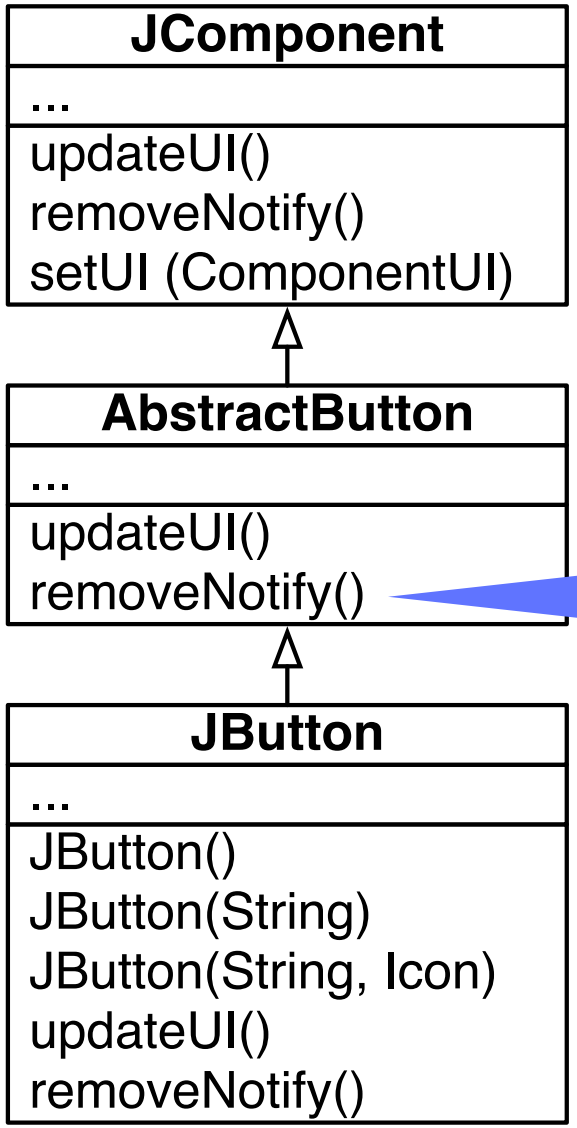

```
public void removeNotify() {
    JRootPane root =
SwingUtilities.getRootPane(this);
   if (root != null &&
root.getDefaultButton() == this){
     root.setDefaultButton(null);
   }

   super.removeNotify();
}
```

send removeNotify to button

```
JButton button = new JButton("OK");
button.removeNotify();
```

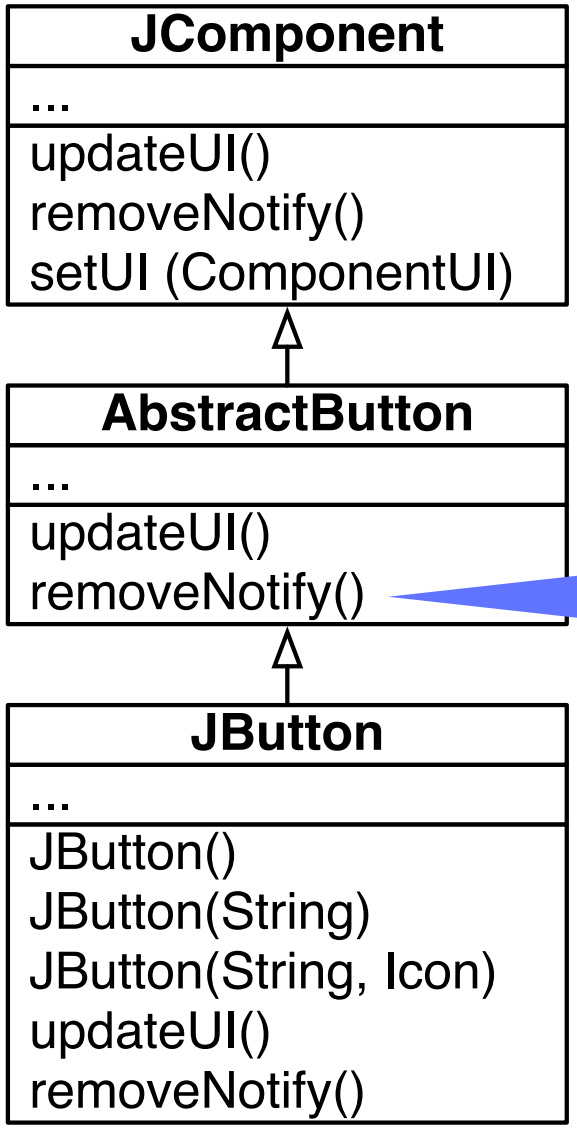
Class inheritance in action!




```
public void removeNotify() {
    super.removeNotify();
    if(isRolloverEnabled()) {
        getModel().setRollover(false);
    }
}
```

```
JButton button = new JButton("OK");
button.removeNotify();
```

Class inheritance in action!

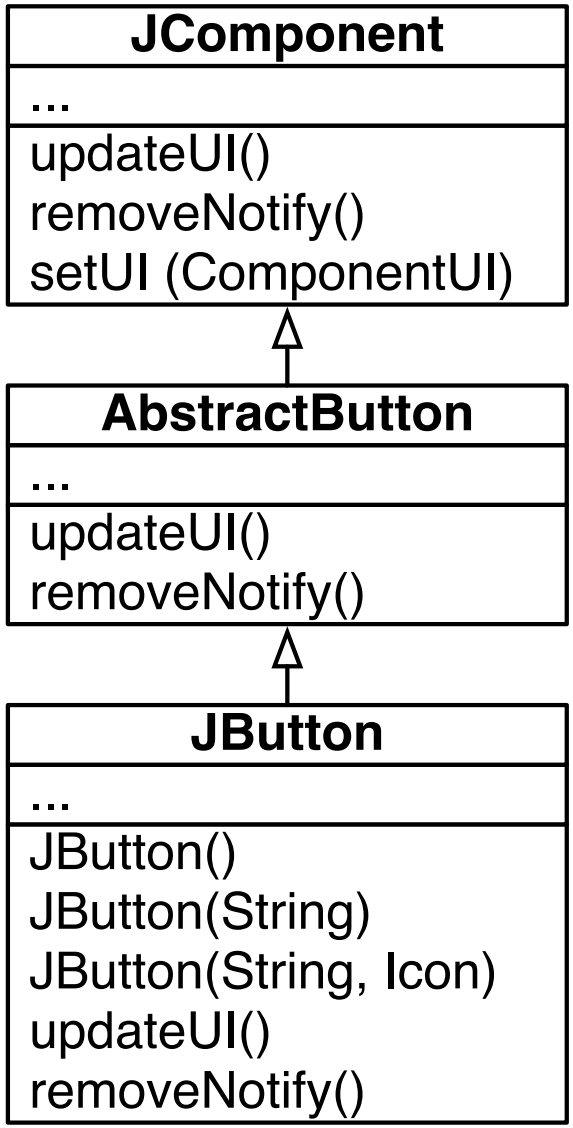



```
public void removeNotify() {
    super.removeNotify();
    if(isRolloverEnabled()) {
        getModel().setRollover(false);
    }
}
```

send removeNotify to button, but the lookup starts in JComponent

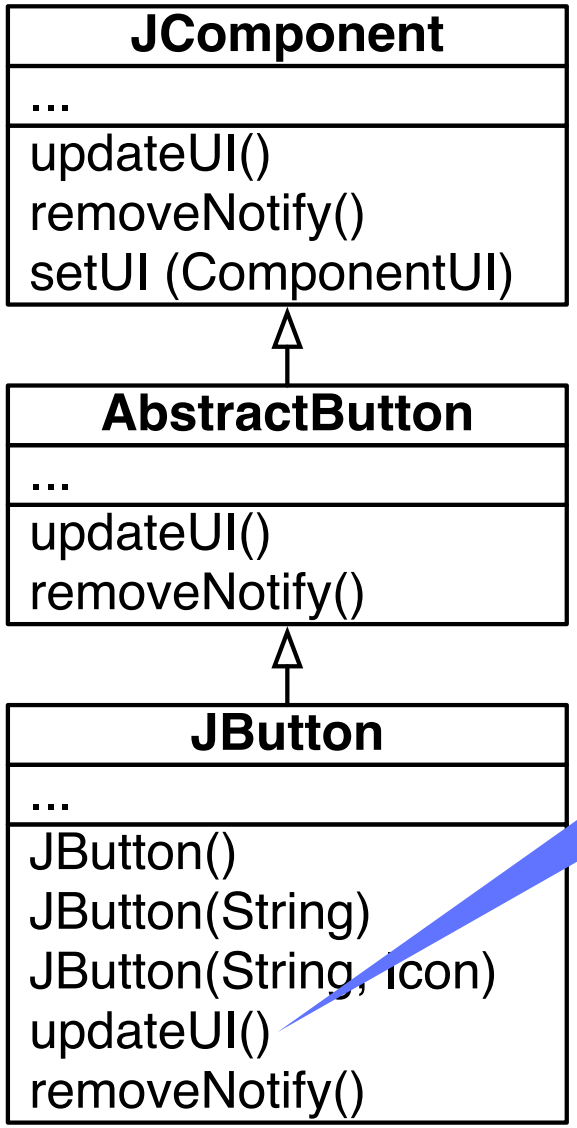
```
JButton button = new JButton("OK");
button.removeNotify();
```

Class inheritance in action!




```
JButton button = new JButton("OK");
button.updateUI();
```

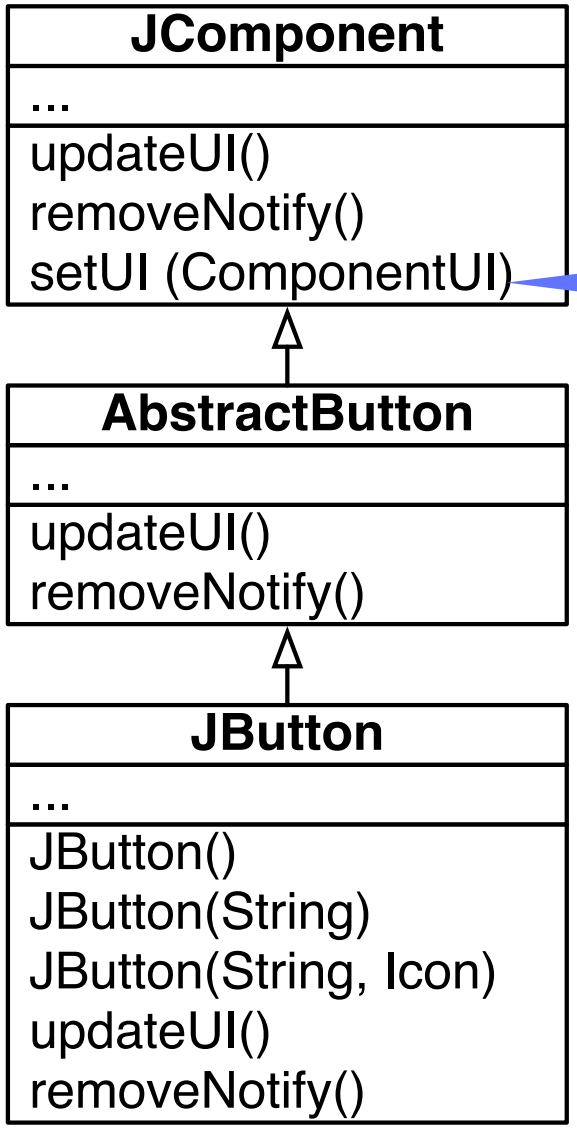
Class inheritance in action!




```
public void updateUI() {
  setUI((ButtonUI)UIManager.
            getUI(this));
}
```

```
JButton button = new JButton("OK");
button.updateUI();
```

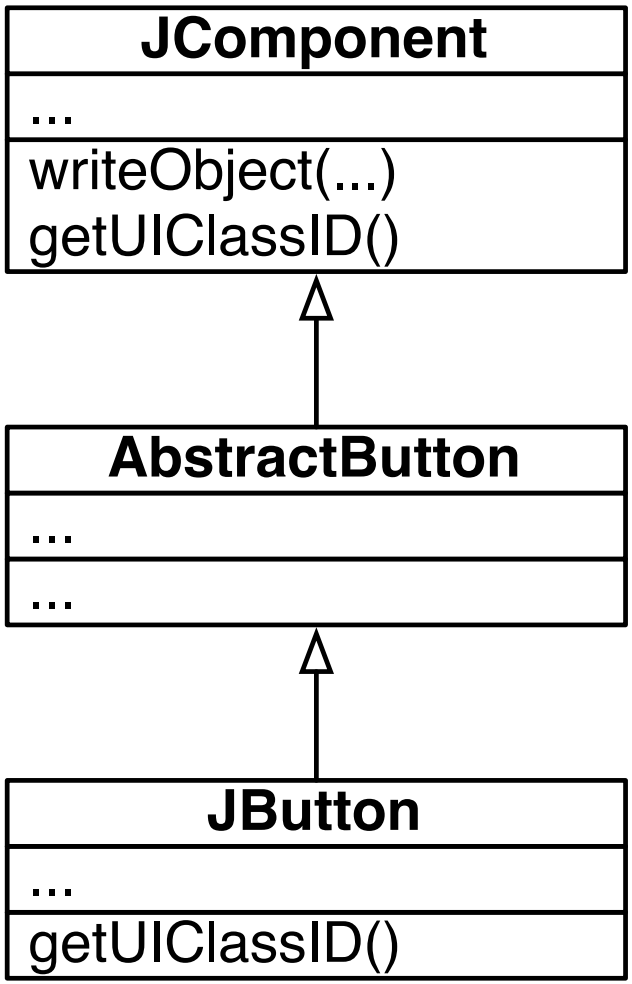
Class inheritance in action!




```
protected void setUI(ComponentUI
newUI) {
  /* We do not check that the UI
instance is different
  * before allowing the switch in
order to enable the
  * same UI instance *with different
default settings*
  * to be installed.
  */
  if (ui != null) {
      ui.uninstallUI(this);
    //clean UIClientPropertyKeys
  ...
```

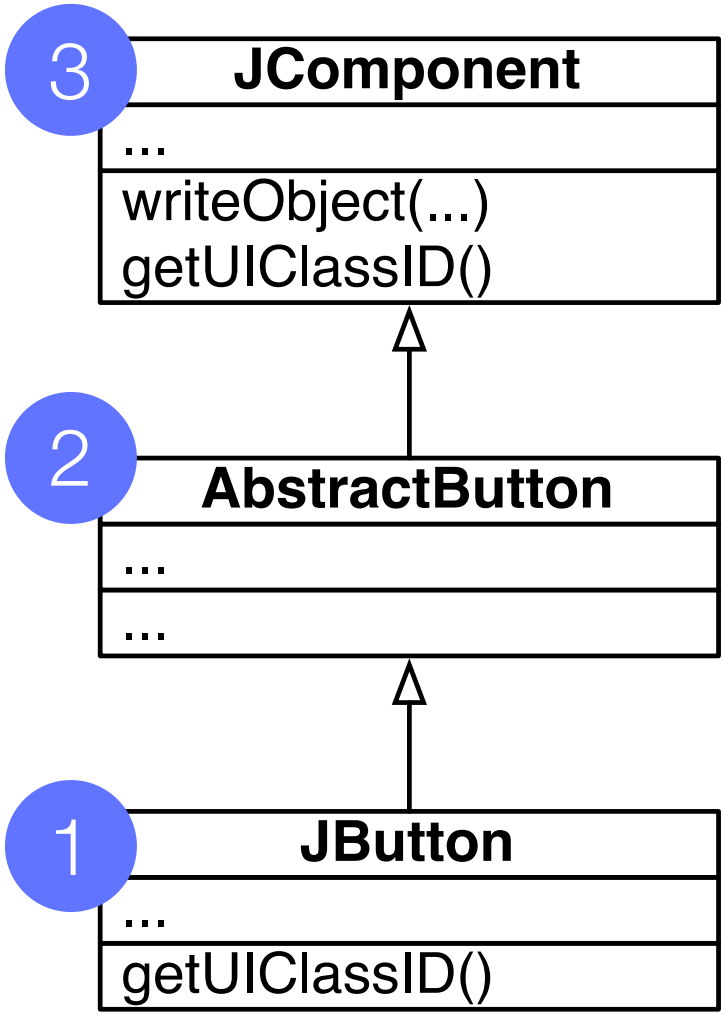
```
JButton button = new JButton("OK");
button.updateUI();
```

this always refers to the receiver



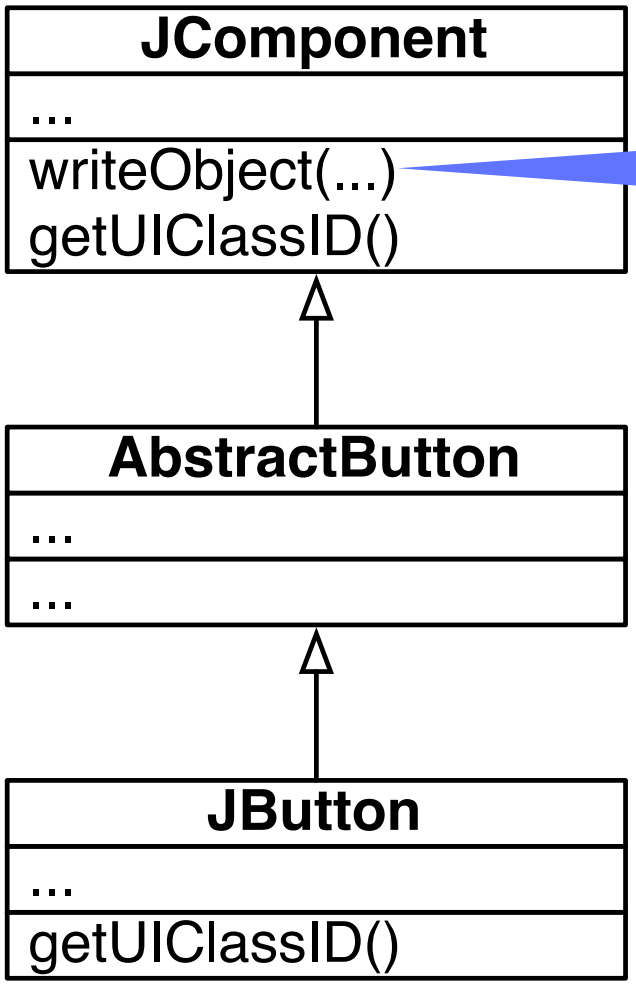

```
JButton button = new JButton(“OK”);
button.writeObject(stream);
```

writeObject(...) is found in JComponent




```
JButton button = new JButton("OK");
button.writeObject(stream);
```

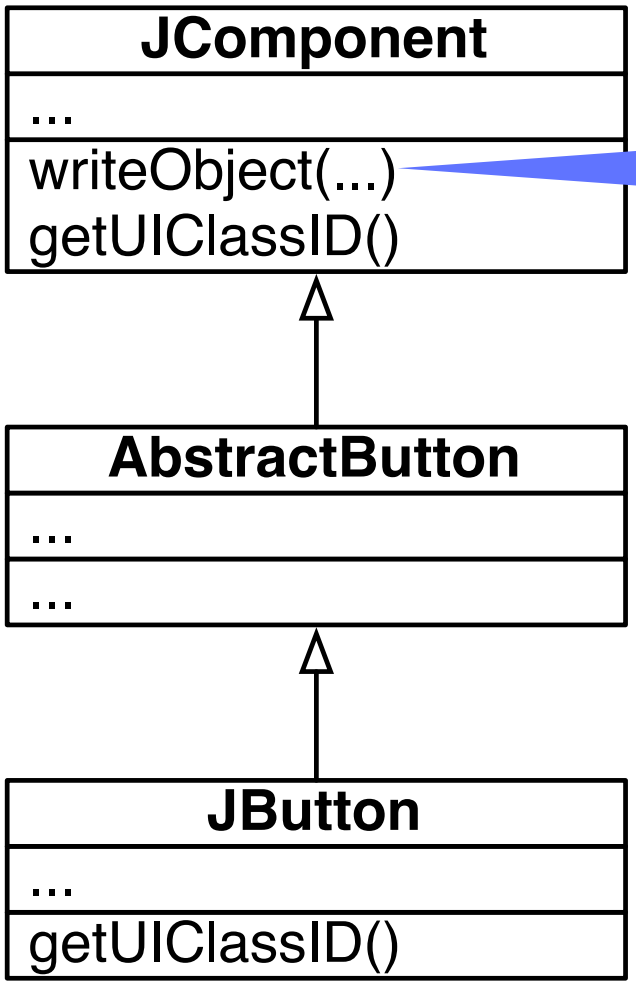
writeObject(...) is found in JComponent




```
public void writeObject(ObjectOutputStream s)
throws IOException {
        s.defaultWriteObject();
        if (getUIClassID().equals(uiClassID)) {
        ...
```

```
JButton button = new JButton("OK");
button.writeObject(stream);
```

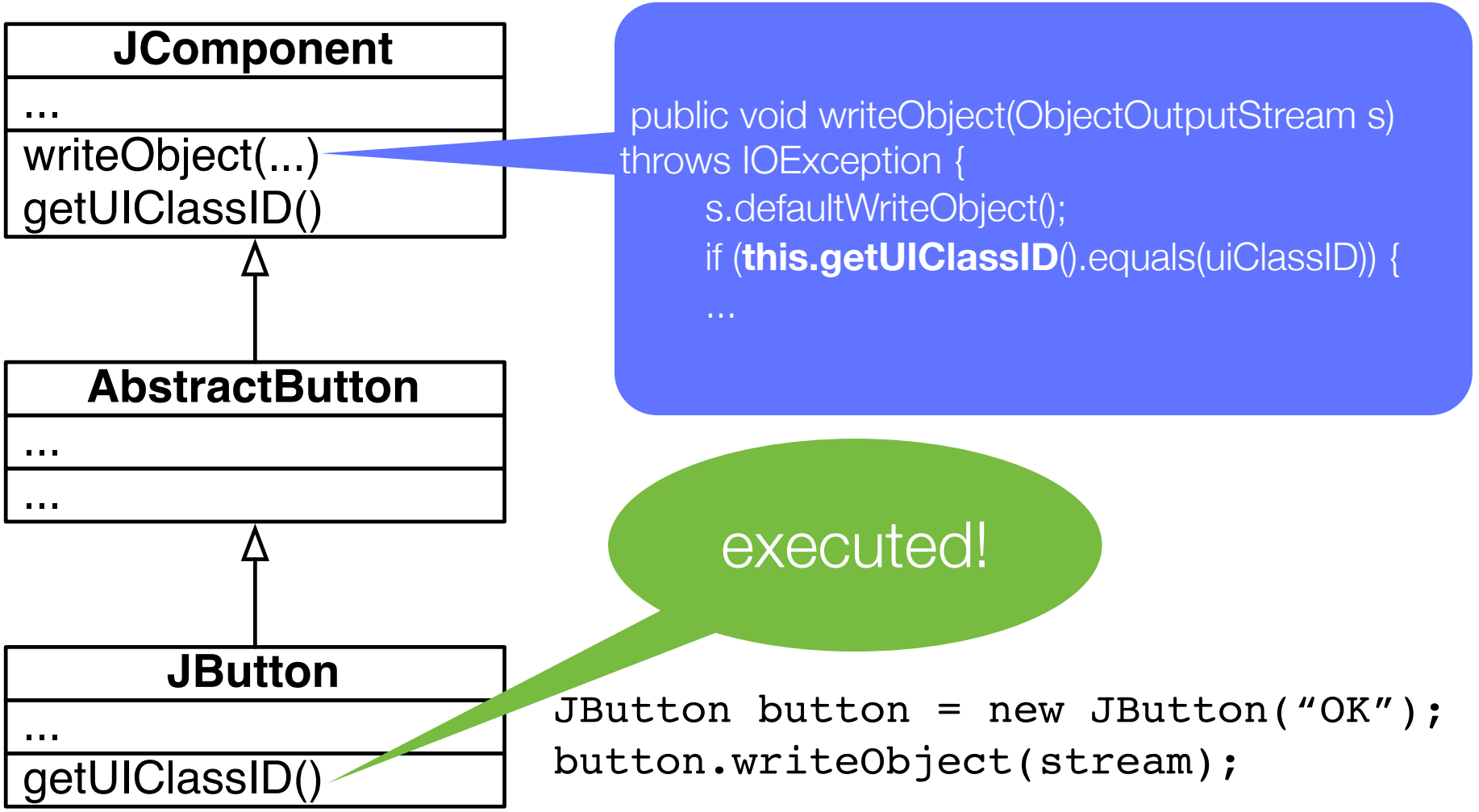
writeObject(...) is found in JComponent




```
public void writeObject(ObjectOutputStream s)
throws IOException {
        s.defaultWriteObject();
        if (this.getUIClassID().equals(uiClassID)) {
        ...
```

```
JButton button = new JButton("OK");
button.writeObject(stream);
```

writeObject(...) is found in JComponent



Small exercise

```
class A{
 void foo(){
  System.out.println("A.foo()");
  this.bar ();
 }
 void bar (){
  System.out.println("A.bar()");
 }
}
```

```
class B extends A {
 void foo() {
  super.foo();
 }
 void bar (){
  System.out.println("B.bar()");
  }
}
```

what `new B().foo()` prints?

Small exercise

```
class A{
    boolean test1(){
        return super.equals(this);
    }

    A yourself(){
        return this;
    }
}
```

```
class B extends A {
  boolean test2() {
    return super.yourself().
                        equals(this);
  }
  boolean test3() {
    return super.
           equals(super.yourself());
  }
}
```

```
new B().test1(), new B().test2(), new B().test3()
```

??

Small exercise

```
class A{
    boolean test(){
        return super.getClass() ==
                this.getClass();
    }
}
class B extends A {
    public static void main(String[] argv) {
        System.out.println(new B().test());
    }
}
```

Outline

1. This and super pseudo variables
2. **Liskov principle**
    1. **theory**
    2. **concrete applications**
3. Java Interfaces

Liskov substitution principle

Initially introduced in 1974 by Barbara Liskov

Formulated in 1994 with Jeannette Wing as follows:

Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T.

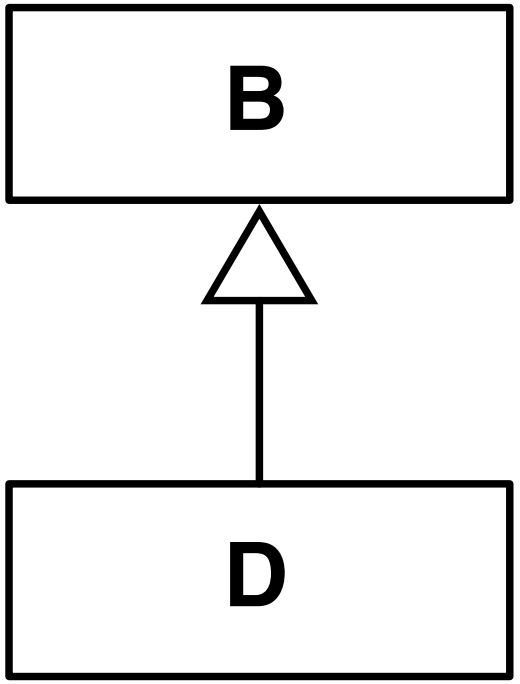
Barbara Liskov received the Turing Award in 2008

Liskov principle vulgarized

Subtypes must be substitutable for their base types

Liskov principle vulgarized

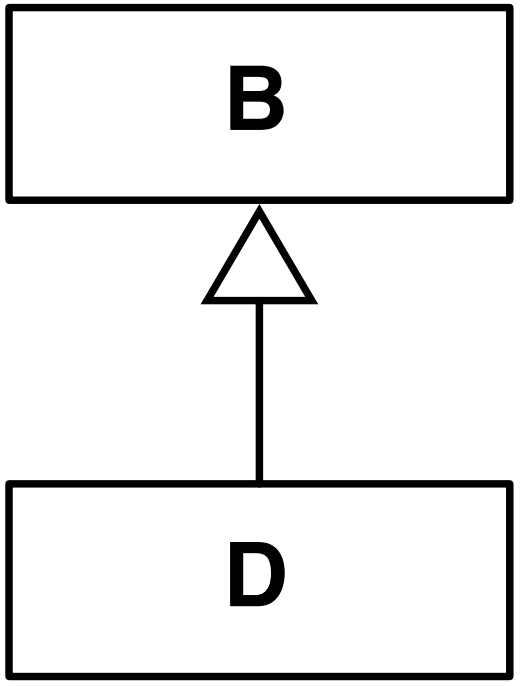
```
void f (B object) {
   ...
}
```



Liskov principle vulgarized

```
void f (B object) {
   ...
}
```

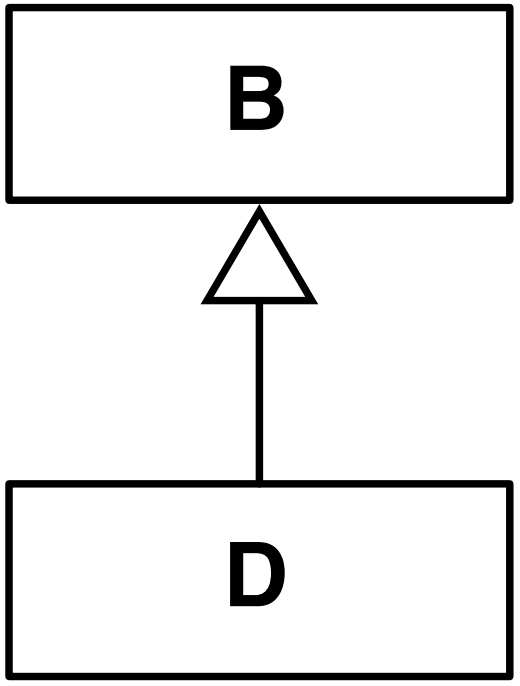
if `f(new B())` behaves correctly, `f(new D())` has to correctly behave as well



Fragile class

```
void f (B object) {
   ...
}
```

if `f(new B())` behaves correctly and `f(new D())` not, then we say that `D` is fragile in the presence of `f`



Some practical illustrations

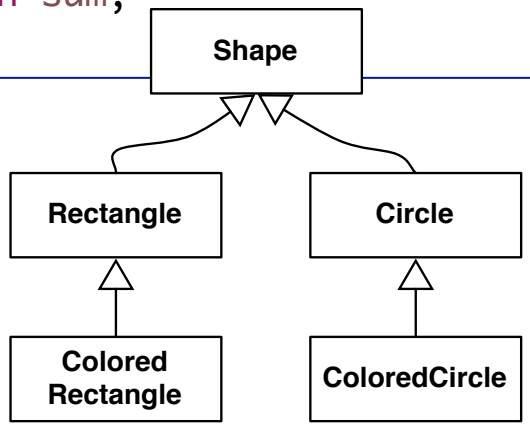
Procedural coding style

Object initialization

Access privileges cannot be weakened

Procedural coding style

```
public static long sumShapes(Shape[] shapes) {
    long sum = 0;
    for (int i=0; i<shapes.length; i++) {
        if (shapes[i] instanceof Rectangle) {
            Rectangle r = (Rectangle)shapes[i];
            sum += (r.width * r.height);
            break;
        }
        if (shapes[i] instanceof Circle) {
            Circle r = (Circle)shapes[i];
            sum += (Math.PI * r.radius * r.radius);
            break;
        }
        // more cases
    }
    return sum;
}
```

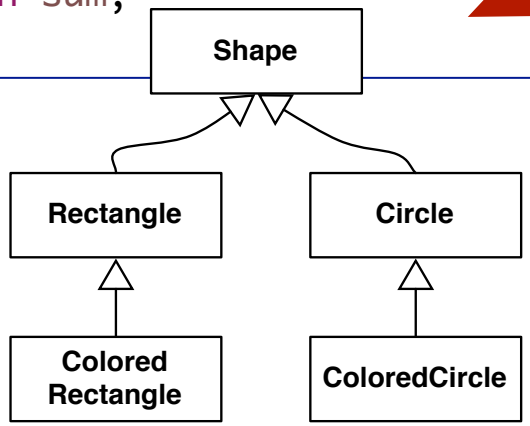


???

Procedural coding style

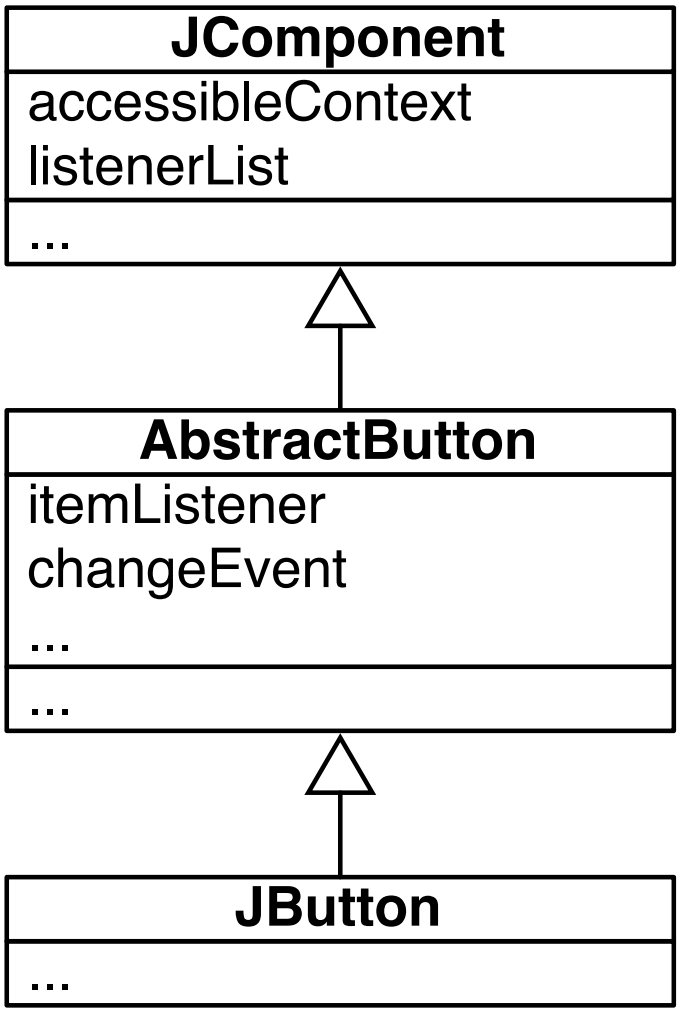
```
public static long sumShapes(Shape[] shapes) {
    long sum = 0;
    for (int i=0; i<shapes.length; i++) {
        if (shapes[i] instanceof Rectangle) {
            Rectangle r = (Rectangle)shapes[i];
            sum += (r.width * r.height);
            break;
        }
        if (shapes[i] instanceof Circl
            Circle r = (Circle)shapes
            sum += (Math.PI *
            break;
        }
        // more cases
    }
    return sum;
}
```

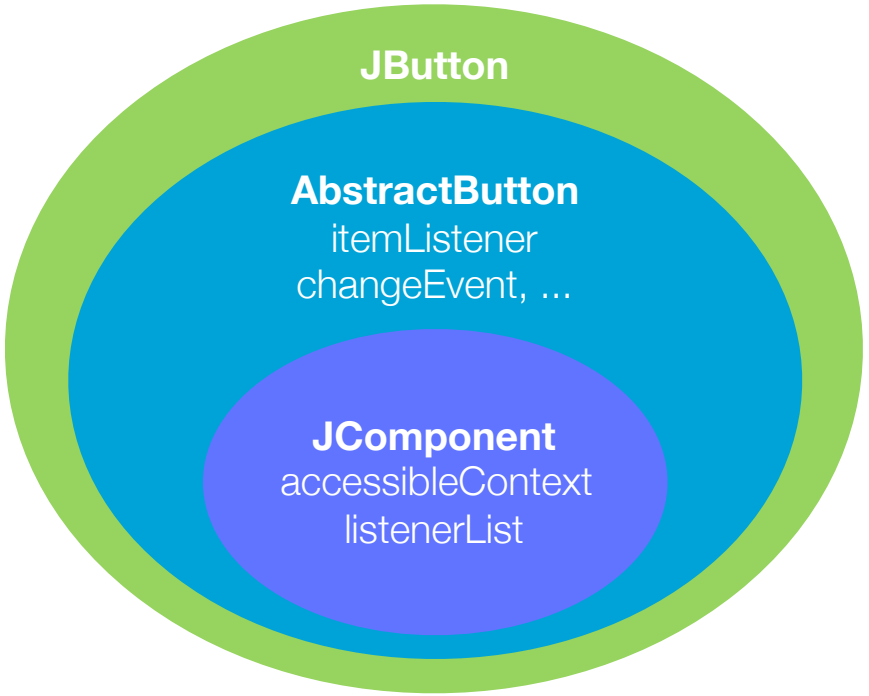
Simple violation of the Liskov principle



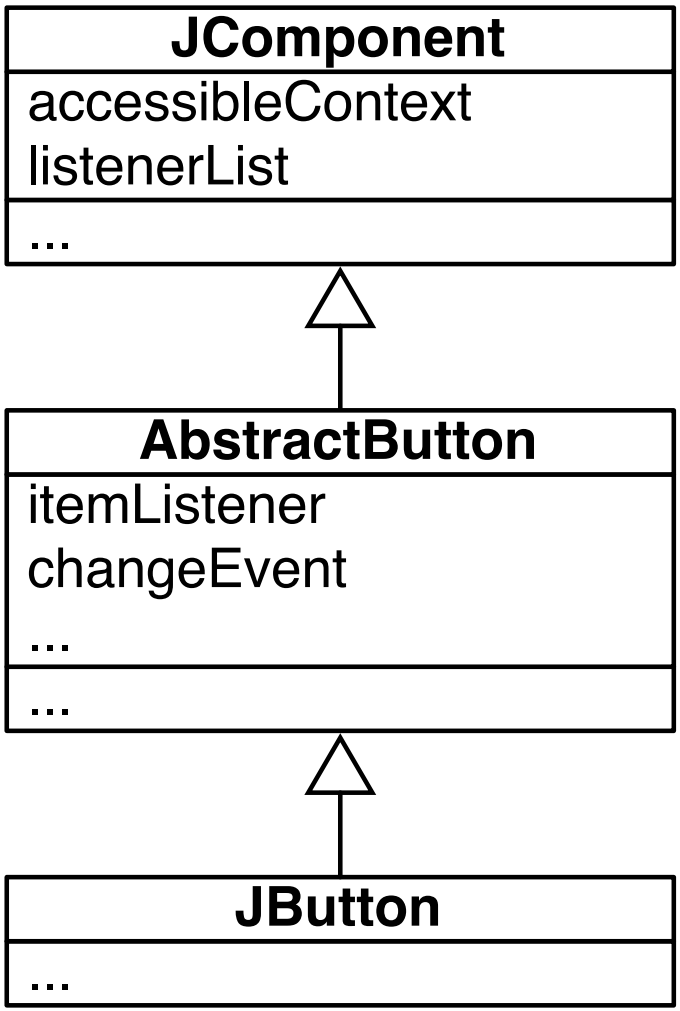
???

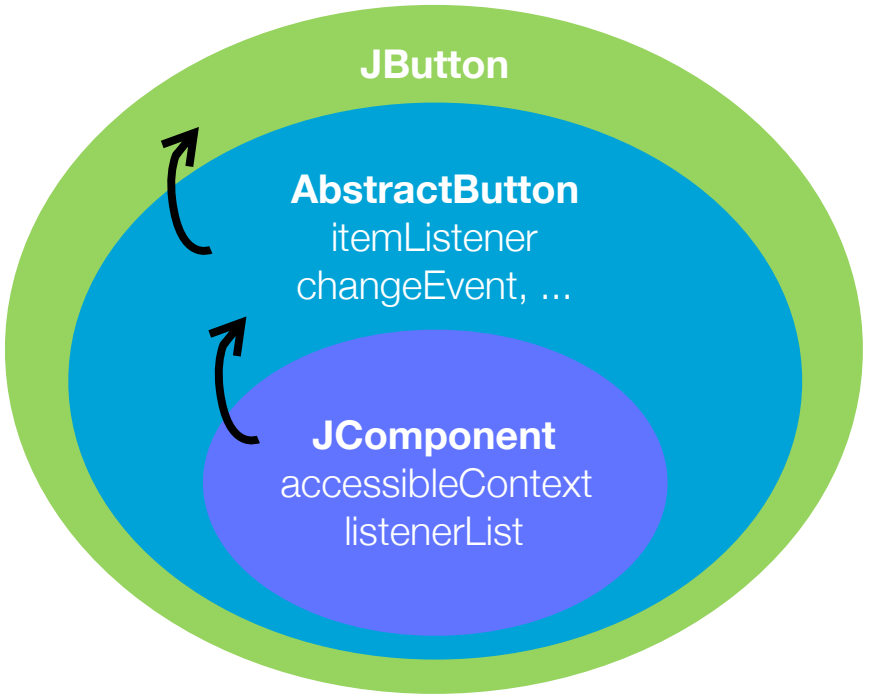
Object initialization





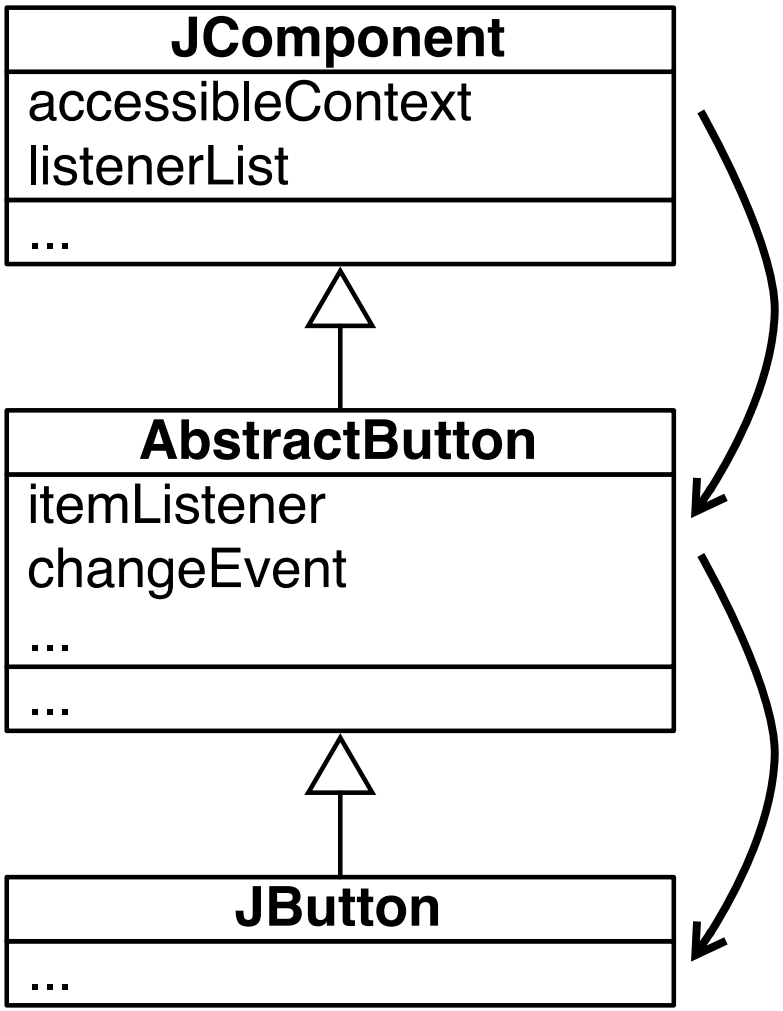
Object initialization

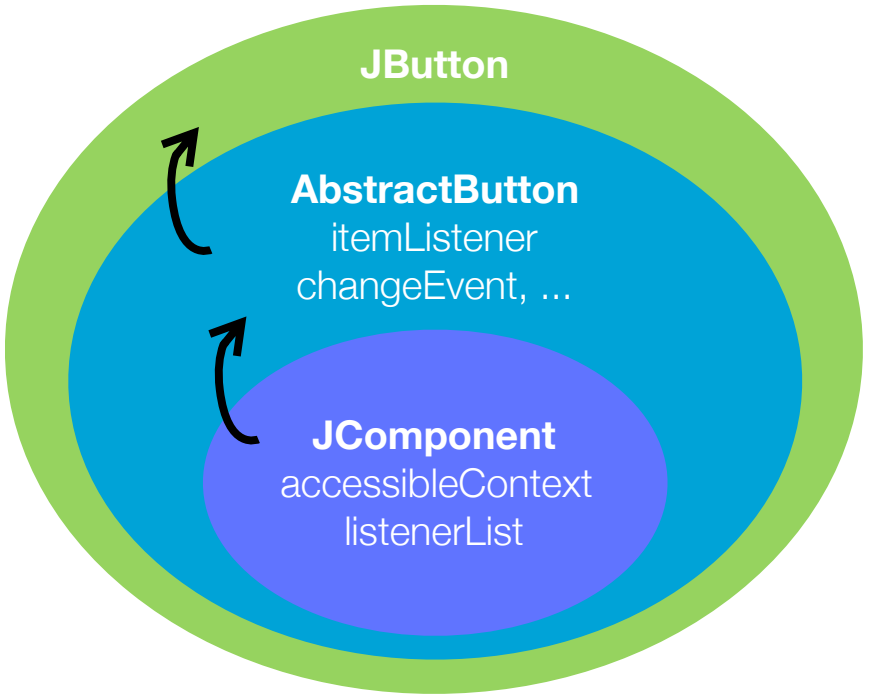




→ order of object initialization, enforced by the super(...) at the beginning of each constructor

Object initialization





→ order of object initialization, enforced by the super(...) at the beginning of each constructor

Privilege access

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Access privileges apply to class definition
and class members (e.g., field, method, inner class)

More on http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Access privileges can only be widened

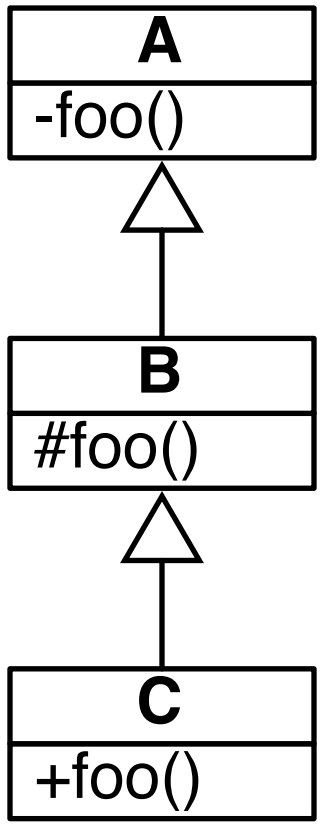
```
class A {
    private void foo () {
    }
}

class B extends A {
    protected void foo () {

    }
}

class C extends B {
    public void foo () {

    }
}
```



Would it be okay to have this?

```
class A {
    public void foo () {
    }
}

class B extends A {
    protected void foo () {

    }
}

class C extends B {
    private void foo () {

    }
}
```

Outline

1. This and super pseudo variables
2. Liskov principle
    1. theory
    2. concrete applications
3. **Java Interfaces**

Java Interfaces

An interface is a group of related methods, and defines an abstract type

```
interface Readable {
  public int read();
}
class Stream implements Readable {
  public int read() { … }
}
```

One can now write:

```
Readable r = new Stream();
```

Java Interfaces

A class may implements more than one interfaces:

```
public abstract class AbstractButton
    extends JComponent
    implements ItemSelectable,SwingConstants { … }
```

Implementing an interface allows a class to become more formal about the behavior it promises to provide.

Interfaces form a *contract between the class and the outside world*, and this contract is enforced at build time by the compiler.

Java Interfaces

“Methods form the object's interface with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.” — docs.oracle.com

In practice, Java interfaces are often used to *abstract a domain variation*

Simple rule: Whenever you need more than one kind of objects, then you need to use interfaces

Java Interfaces

Interfaces are not instantiated, but rather implemented

In practices, it often happens that abstract classes implements interfaces

An interface may have 0, 1 or more super interfaces

```
interface LotsOfColors extends
RainbowColors, PrintColors { … }
```

Inheritance

We have seen many aspects related to inheritance and sub-typing

abstract classes, extends keyword, interfaces, super keyword, …

We have studied only the syntactical aspects of them

Properly using them is particularly difficult, and most of the remaining of the semester is exactly about that.

What you should know!

The difference between the *this* and *super* pseudo-variables

What is the Liskov principle?

How the Liskov principle affects the design of a programming language

Can you answer to these questions?

Why *this* and *super* are called pseudo-variables?

Why the effect of sending a message to *super* cannot be “the method lookup begins in the class of the object receiver”?

License

http://creativecommons.org/licenses/by-sa/2.5

creative commons

COMMONS DEED

Attribution-ShareAlike 2.5

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.