



Essential of Object Oriented Programming

Alexandre Bergel

<http://bergel.eu>

21/09/2020

Goal of this lecture

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

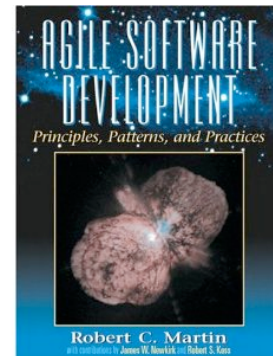
Method lookup with inheritance

Java interfaces

Recommended Texts

Agile Software Development, Principles, Patterns, and Practices

Robert C. Martin “Uncle Bob”, 2002



Outline

- 1.This and super pseudo variables
- 2.Java Interfaces

Outline

1.This and super pseudo variables

2.Java Interfaces

```
public class Tank {  
    private int healthPoint;  
  
    public Tank() {  
        this.healthPoint = 175;  
    }  
  
    public void receiveDamage(int amount) {  
        this.healthPoint -= amount;  
    }  
}  
  
public class InfestedTank extends Tank {  
    public void receiveDamage(int amount) {  
        super.receiveDamage(amount * 2);  
    }  
}
```

```
public class Tank {  
    private int healthPoint;  
  
    public Tank() {  
        this.healthPoint = 175;  
    }  
  
    public void receiveDamage(int amount) {  
        this.healthPoint -= amount;  
    }  
}  
  
public class InfestedTank extends Tank {  
    public void receiveDamage(int amount) {  
        super.receiveDamage(amount * 2);  
    }  
}
```

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

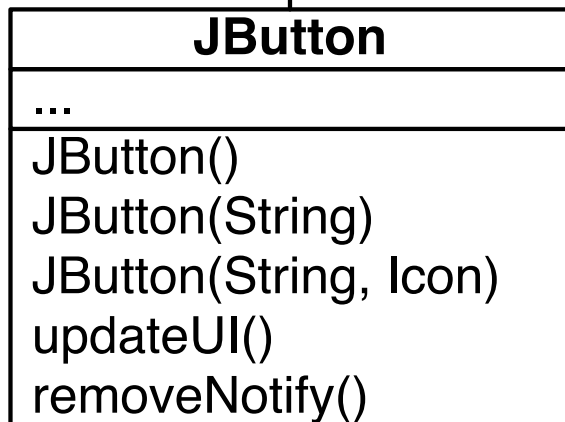
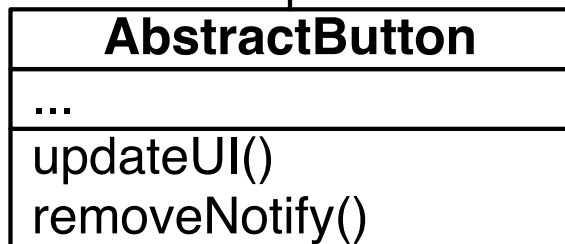
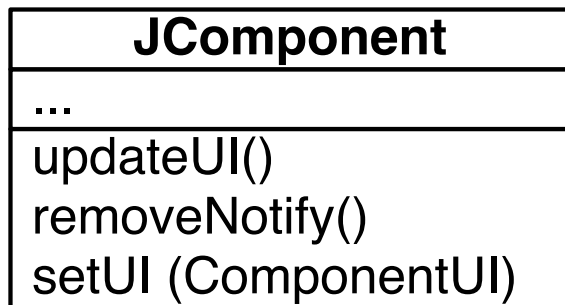
The Java syntax prevents one from using *super* without being followed by “.identifier”

Class inheritance principle

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

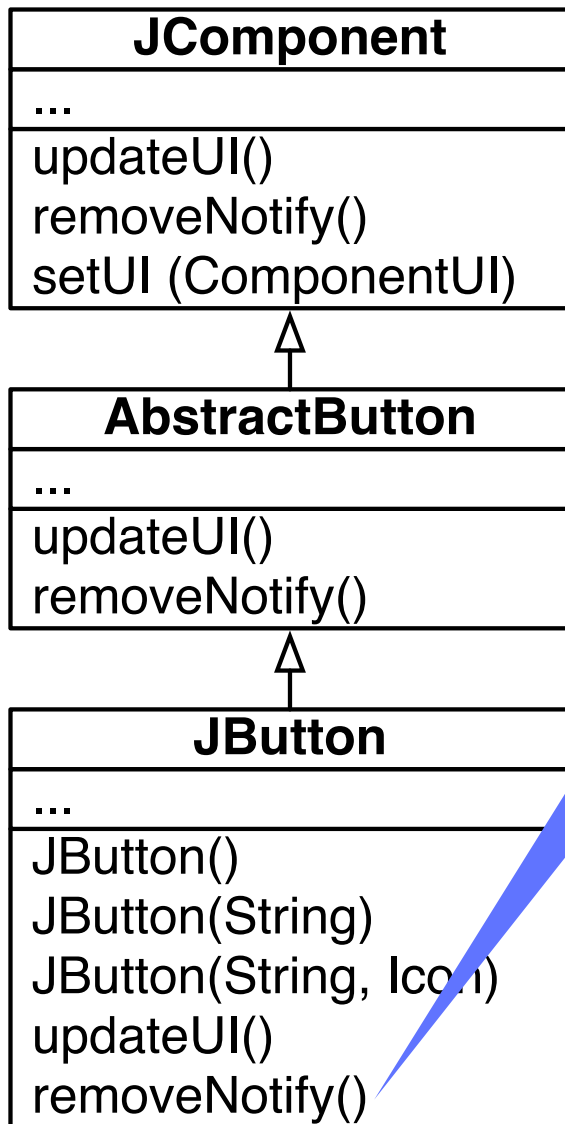
In a statically typed languages (e.g., Java, C#, C++), the lookup *always* find an appropriate method

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



```

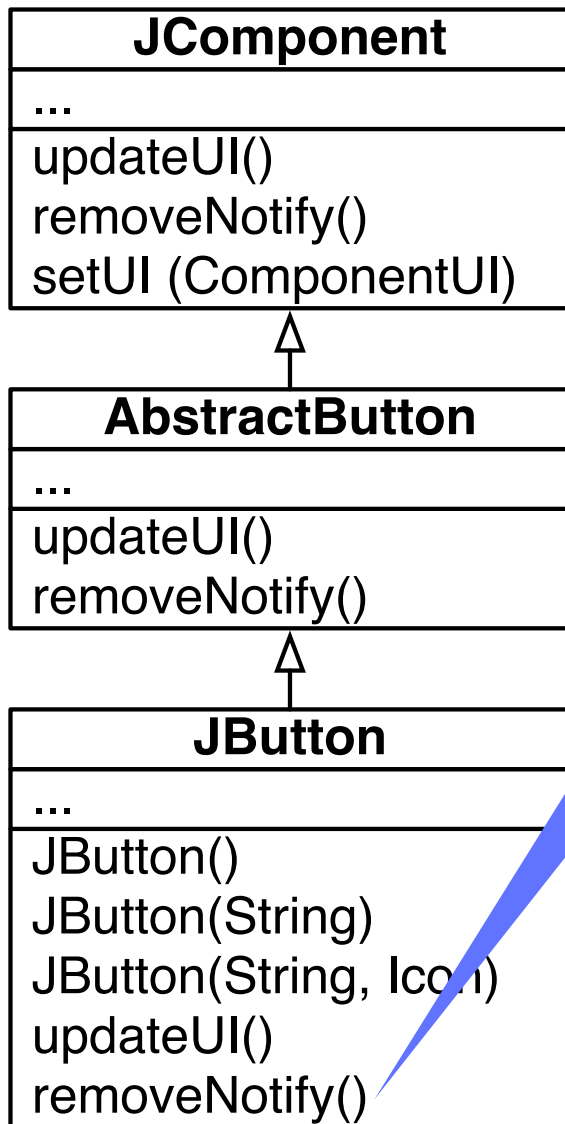
JButton button = new JButton("OK");
button.removeNotify();
  
```



```
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();
```

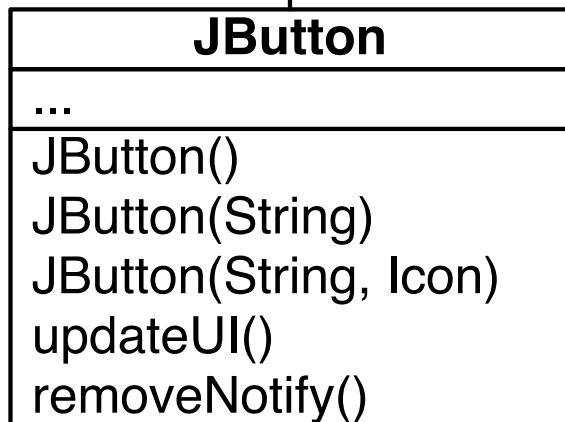
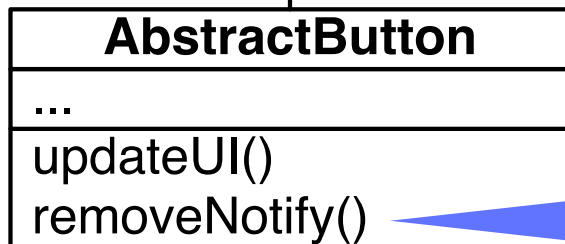
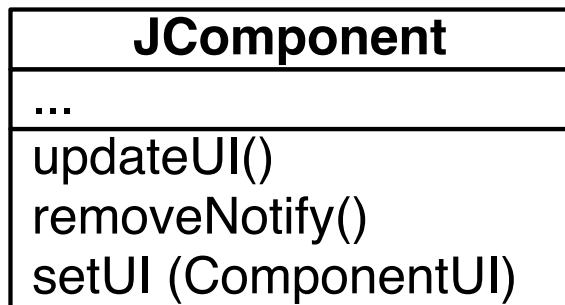


```
public void removeNotify() {
    JRootPane root =
    SwingUtilities.getRootPane(this);
    if (root != null &&
    root.getDefaultButton() == this){
        root.setDefaultButton(null);
    }

    super.removeNotify();
}
```

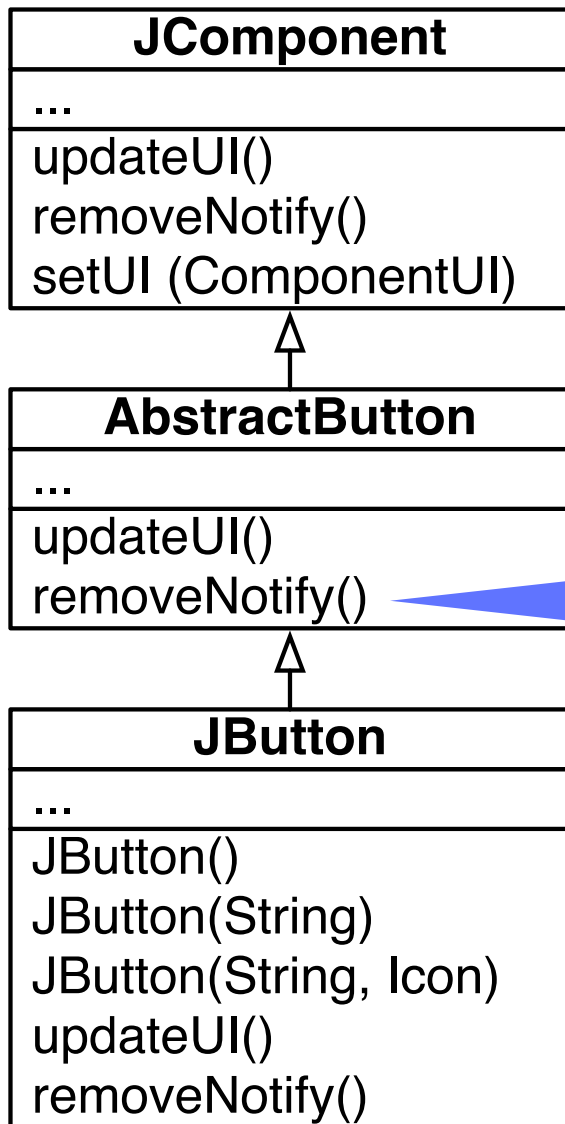
send
removeNotify to
button

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



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

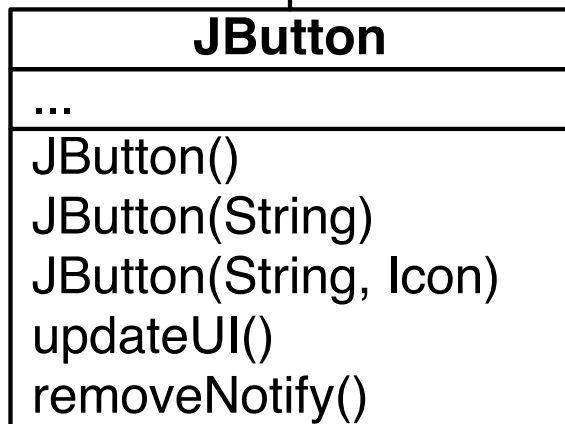
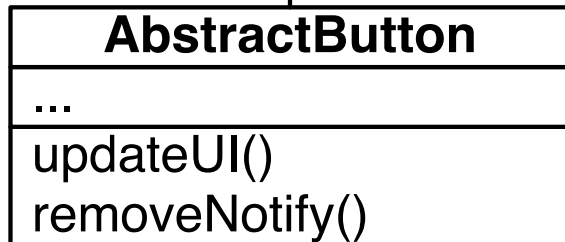
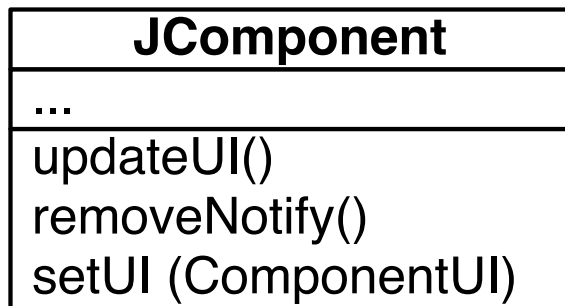
```
JButton button = new JButton("OK");  
button.removeNotify();
```



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

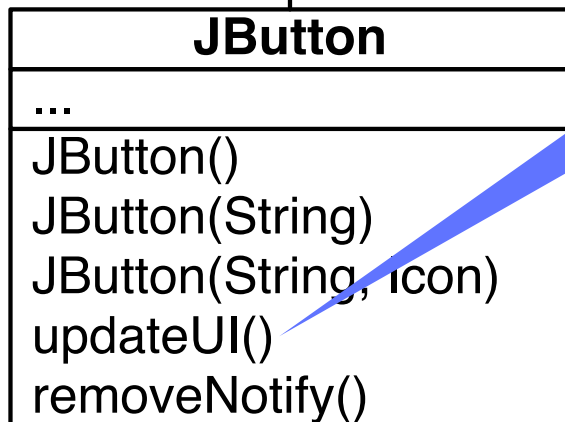
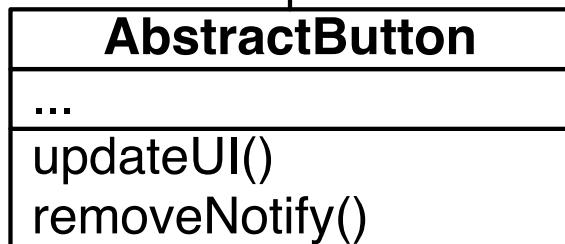
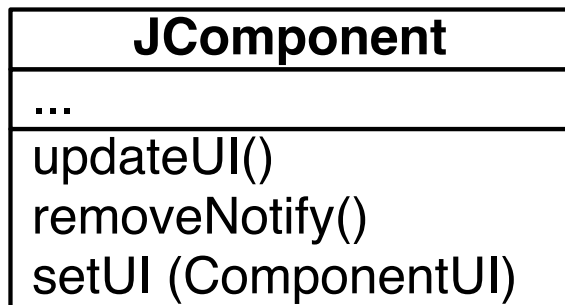
send removeNotify to
button, but the lookup starts
in JComponent

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



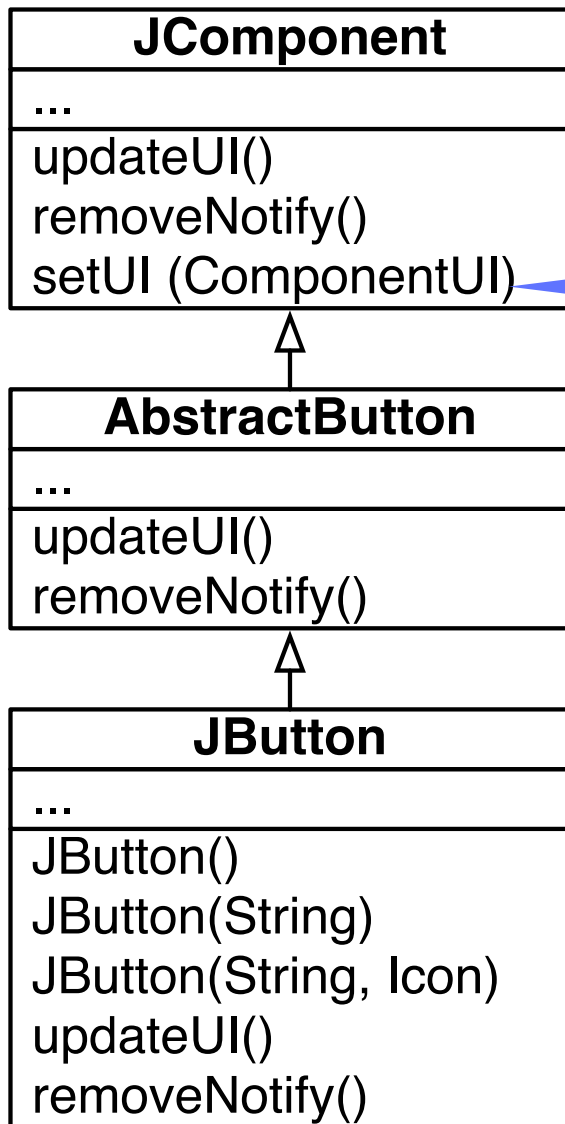
```

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



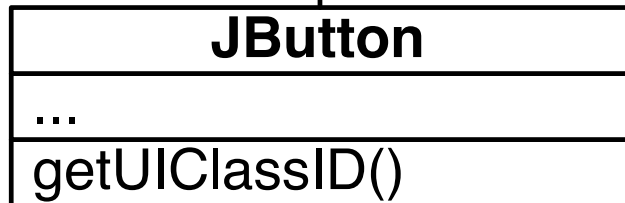
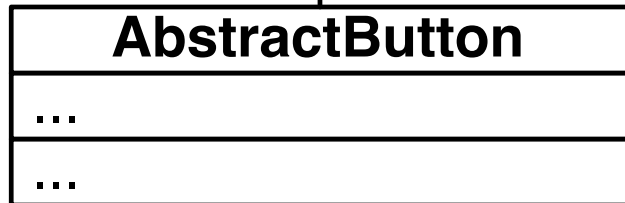
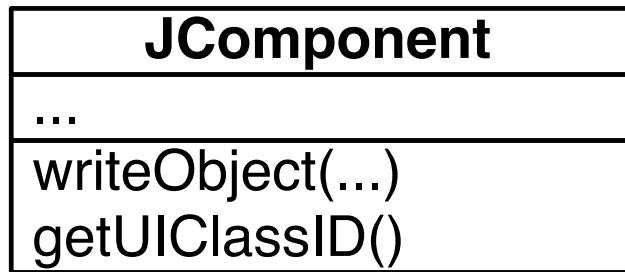
```
public void updateUI() {  
    setUI((ButtonUI)UIManager.  
        getUI(this));  
}
```

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

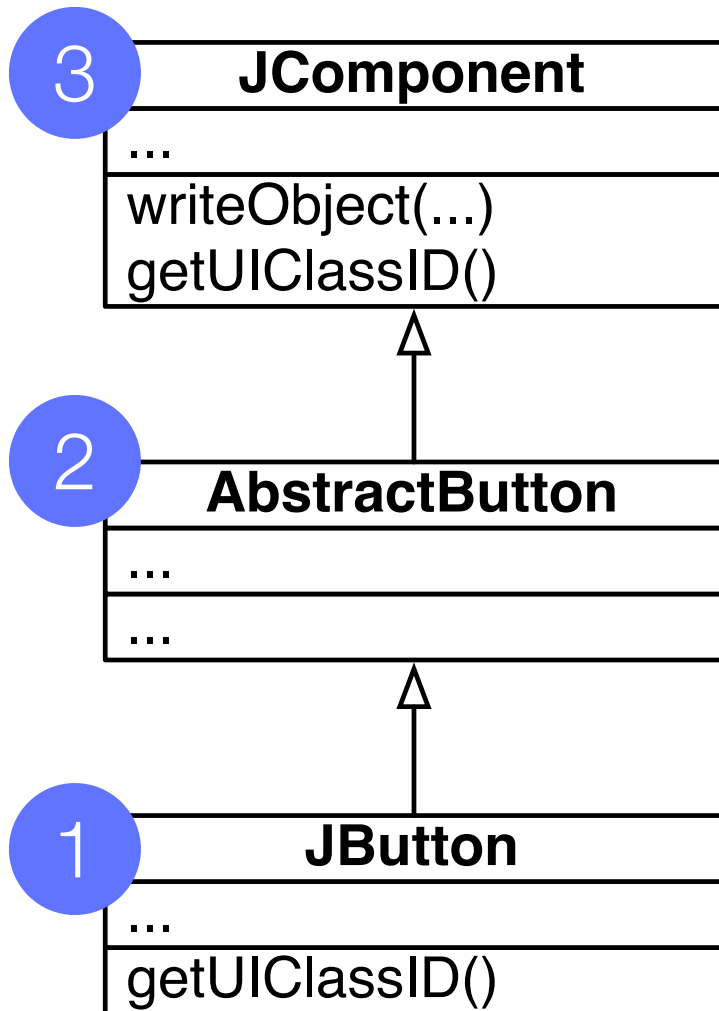


```
protected void setUI(ComponentUI
newUI) {
    /* We do not check that the UI
instance is different
    * before allowing the switch in
order to enable the
    * same UI instance *with different
default settings*
    * to be installed.
    */
    if (ui != null) {
        ui.uninstallUI(this);
  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

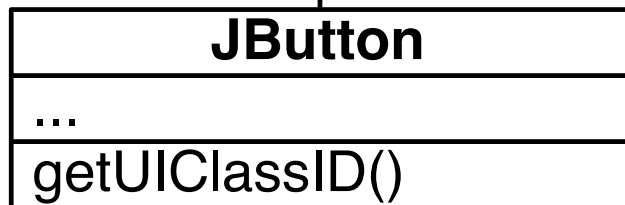
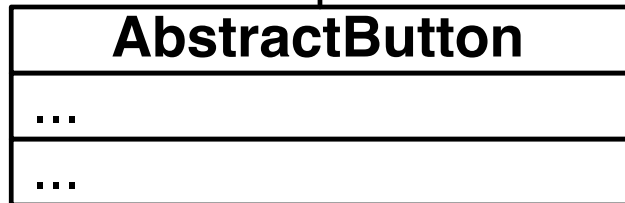
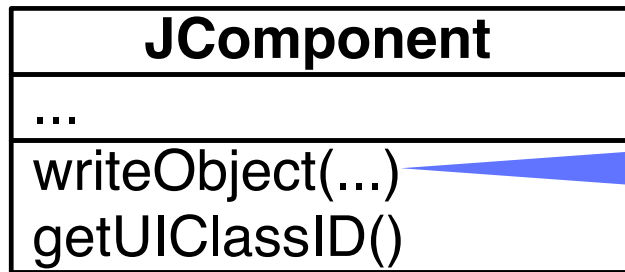
```
JButton button = new JButton("OK");
button.updateUI();
```



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

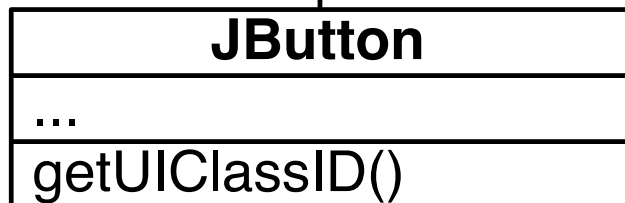
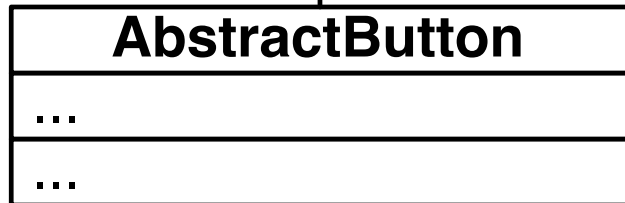
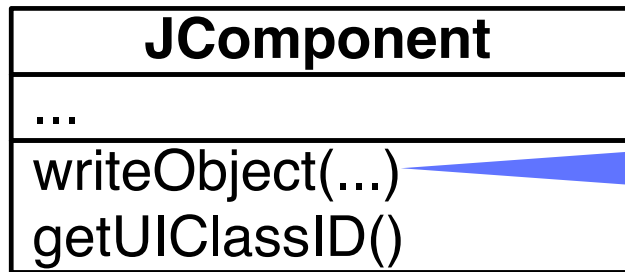


```
JButton button = new JButton("OK");  
button.writeObject(stream);
```



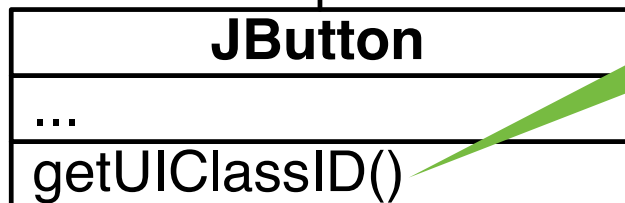
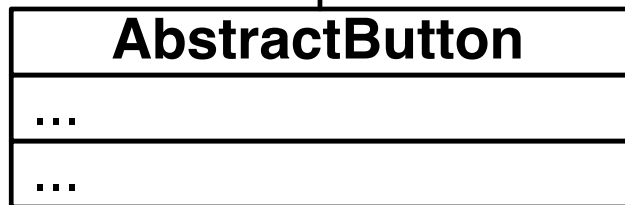
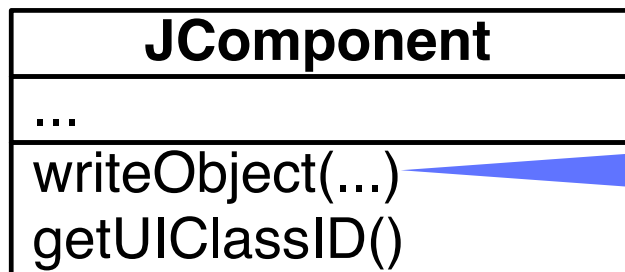
```
public void writeObject(ObjectOutputStream s)
throws IOException {
    s.defaultWriteObject();
    if (getUIClassID().equals(uiClassID)) {
        ...
    }
}
```

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



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

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



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

executed!

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

```
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?

```
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()` ??

```
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.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:

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

```
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, ...

Note that 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 an interface in Java?

How does an interface relate to an abstract class?

Can you answer to these questions?

Why very often abstract classes should implement an interface?

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



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

- Share: copy and redistribute the material in any medium or format
- Adapt: remix, transform, and build upon the material for any purpose, even commercially

The licensor cannot revoke these freedoms as long as you follow the license terms



Attribution: you must give appropriate credit



ShareAlike: if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original

Complete license: <https://creativecommons.org/licenses/by-sa/4.0/>



dcc

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

f @ in / DCCUCHILE