

Essential of Object Oriented Programming

Alexandre Bergel

<http://bergel.eu>

07/09/2020

Goal of this lecture

Understand some important concepts related to *polymorphism* and *class inheritance*

See some *particularities* of Java

Note that we are not discussing about appropriate usage of inheritance. We will do this in the coming weeks

Outline

Defining Inheritance

example in the JDK

constructors

abstract classes

method overriding and overloading

Using class inheritance

We will pick an example from the standard Java class library

The class documentations are available online

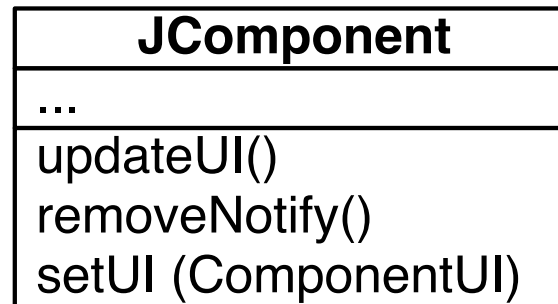
<https://docs.oracle.com/en/java/javase/11/docs/api/index.html>

Source code is freely available

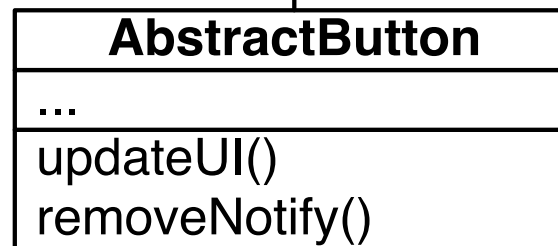
<http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java>

Source code of OpenJava is open source

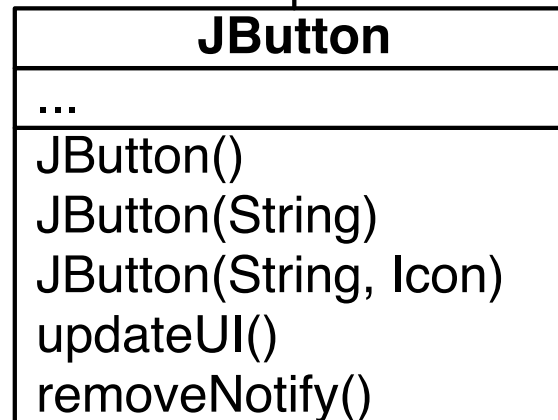
Example in the JDK



5471 LOC

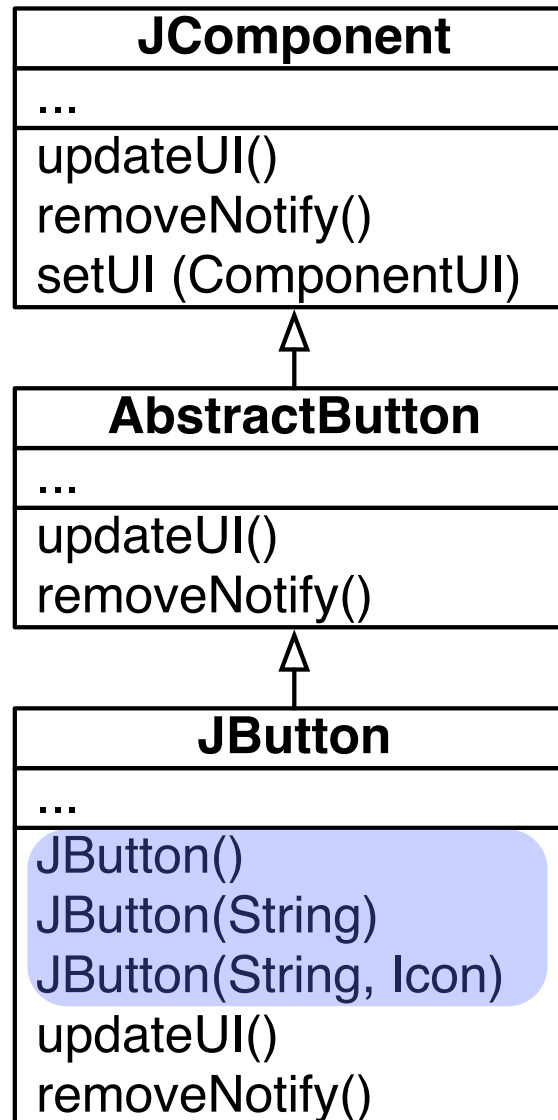


3079 LOC

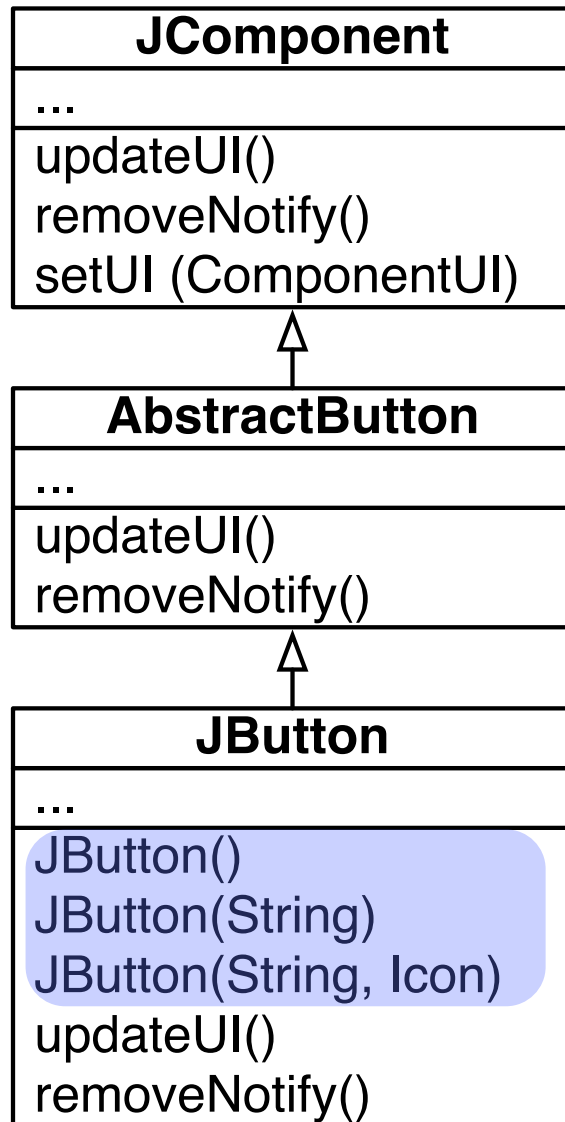


305 LOC

Class constructor



Class constructor



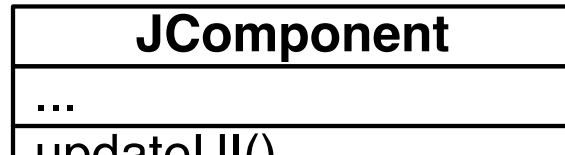
A constructor is responsible to properly initialize an object

When you write **new JButton("OK")**:

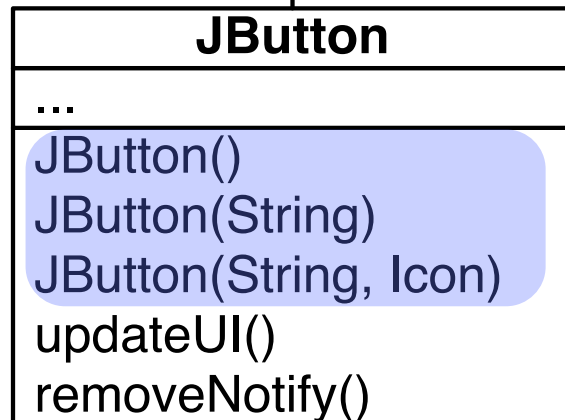
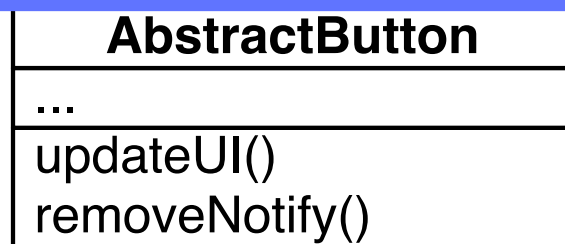
- 1 - the memory is allocated
(object creation)
- 2 - the new object is initialized

Pay attention that having a construction does not mean your object will be well initialized.

Class constructor



Done automatically
by the virtual machine



constructor is responsible to properly initialize an object

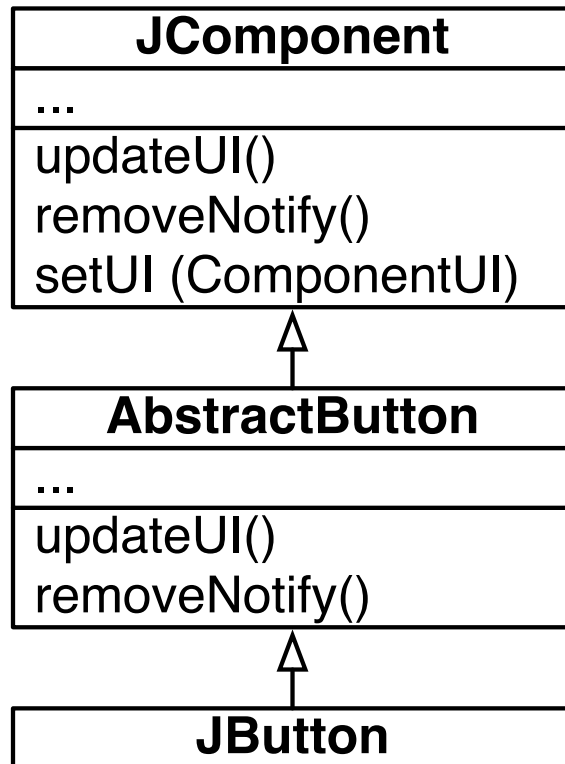
When you write `new JButton("OK")`:

1 - the memory is allocated
(object creation)

2 - the new object is initialized

Pay attention that having a construction does not mean your object will be well initialized.

Class constructor



A constructor is responsible to properly initialize an object

When you write `new JButton("OK")`:

1 - the memory is allocated
(object creation)

2 - the new object is initialized

Specified by the programmer (author of `JButton` and its superclasses)

... mention that having a construction does not mean your object will be well initialized.

Example of incomplete initialization

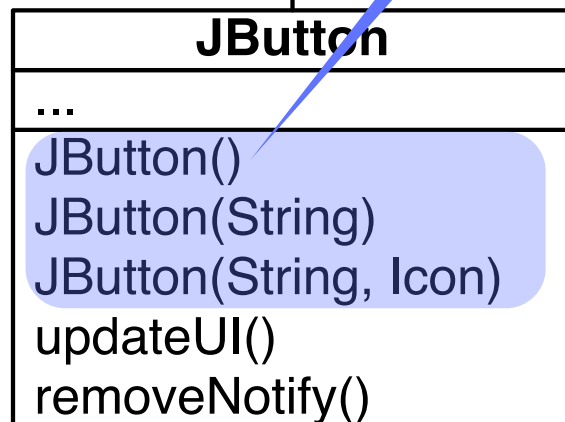
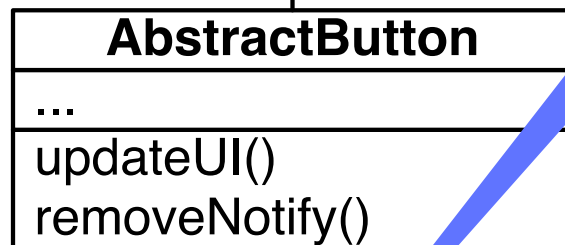
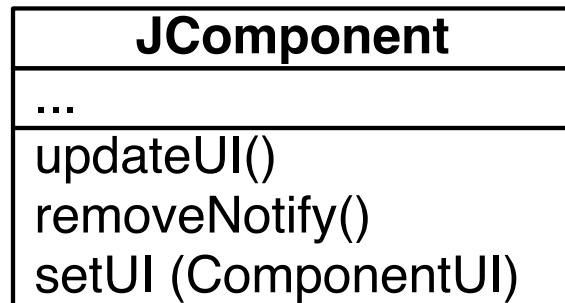
```
public class ColorPoint {  
    private double x, y;  
    private Color color;  
  
    public ColorPoint(double xt, double yt) {  
        x = xt; y = yt;  
    }  
  
    public void setColor(Color aColor) {  
        color = aColor;  
    }  
}
```

Example of incomplete initialization

```
public class ColorPoint {  
    private double x, y;  
    private Color color;  
  
    public ColorPoint(double xt, double yt) {  
        x = xt; y = yt;  
    }  
  
    public void setColor(Color aColor) {  
        color = aColor;  
    }  
}
```

What is the color of new ColorPoint(2, 3) ?
Should probably have a default color

Class constructor

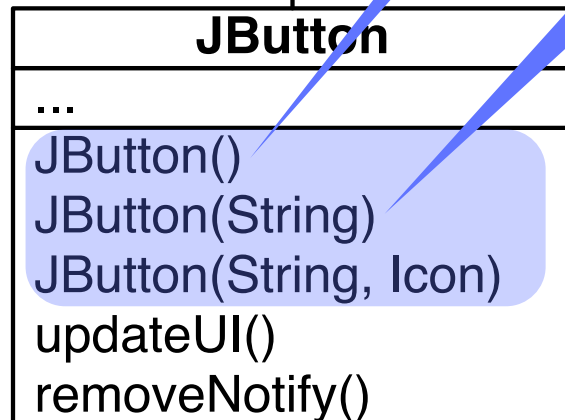
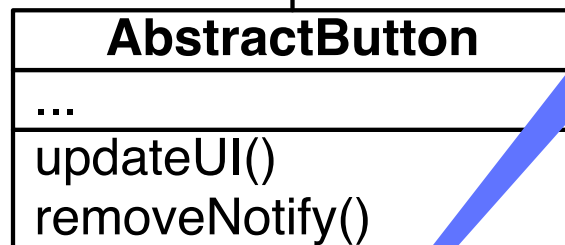
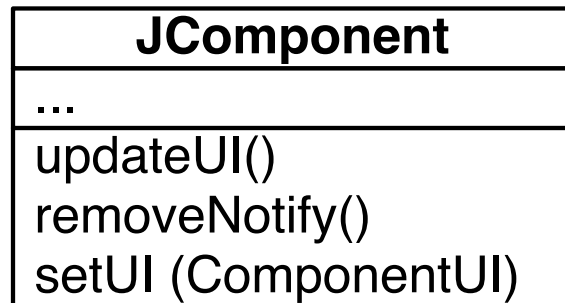


```
public JButton() {  
    this(null, null);  
}
```

Constructor may invoke each other.

The keyword *this* is used for that purpose. Note that this “*this*”, used in to invoke constructor, has nothing to do with the “this” pseudo variable we will later see.

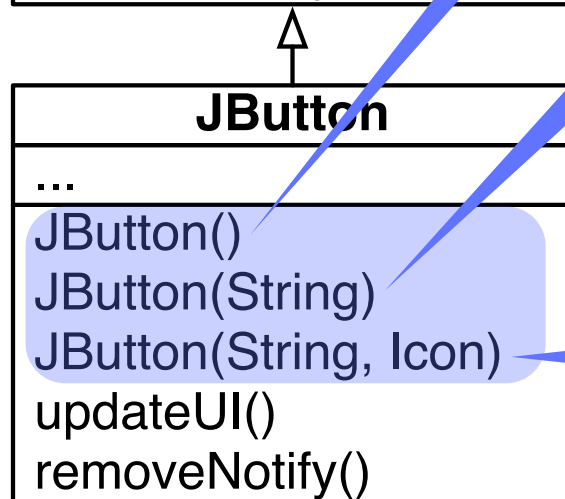
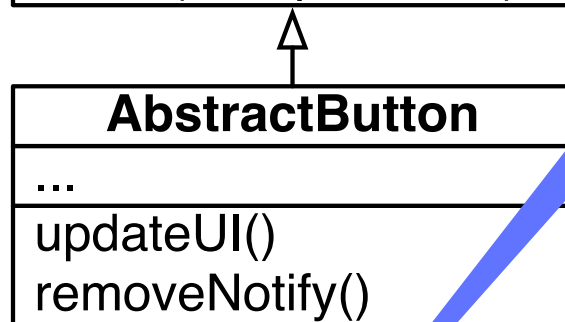
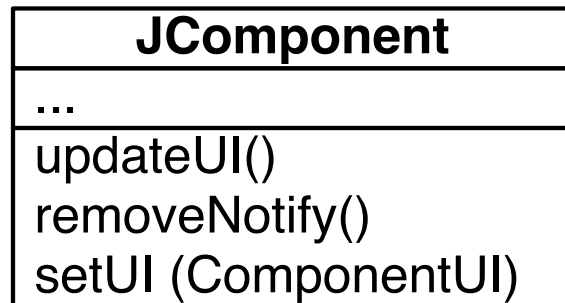
Class constructor



```
public JButton() {  
    this(null, null);  
}
```

```
public JButton(String text) {  
    this(text, null);  
}
```

Class constructor

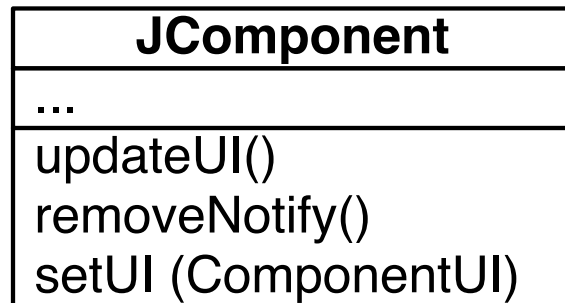


```
public JButton() {  
    this(null, null);  
}
```

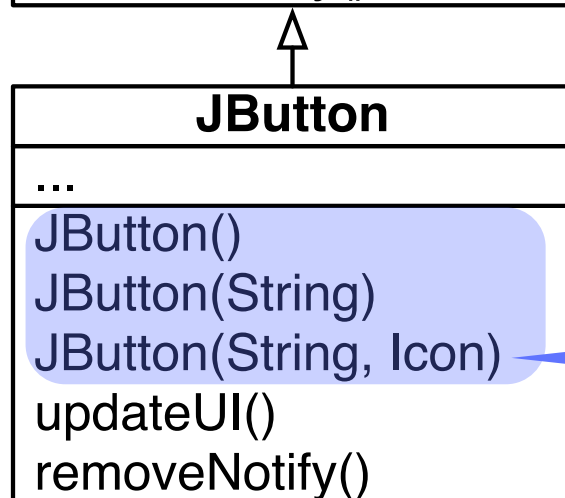
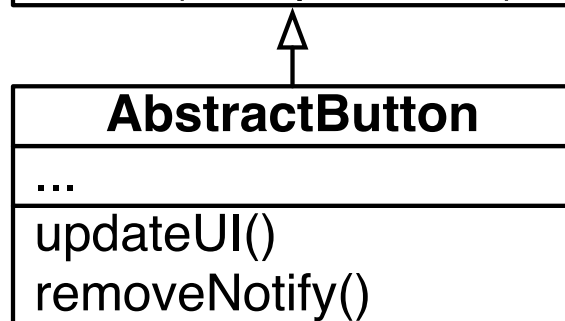
```
public JButton(String text) {  
    this(text, null);  
}
```

```
public JButton(String text, Icon icon) {  
    // Create the model  
    setModel(new DefaultButtonModel());  
  
    // initialize  
    init(text, icon);  
}
```

Class constructor

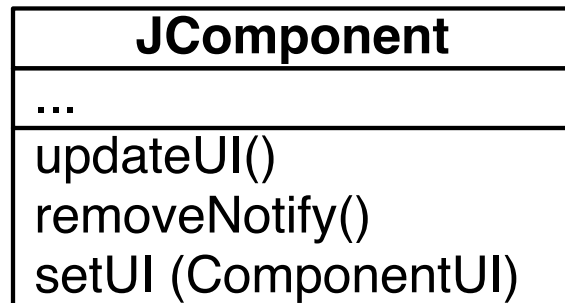


```
public JComponent() {  
    super();  
    enableEvents(AWTEvent.KEY_EVENT_MASK);  
    ...  
}
```

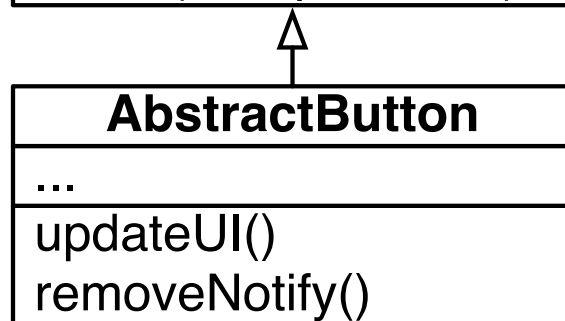


```
public JButton(String text, Icon icon) {  
    // Create the model  
    setModel(new DefaultButtonModel());  
  
    // initialize  
    init(text, icon);  
}
```

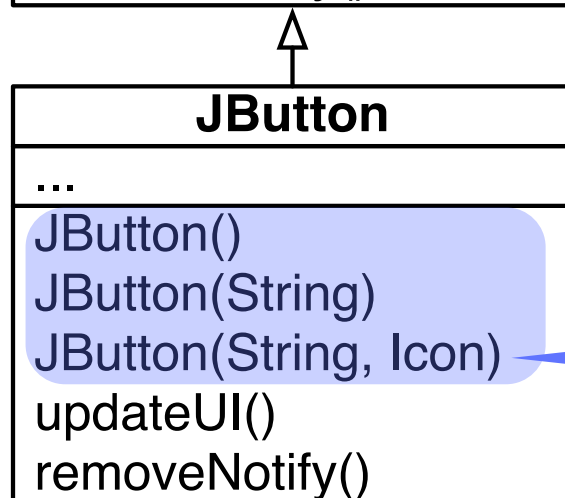
Class constructor



```
public JComponent() {  
    super();  
    enableEvents(AWTEvent.KEY_EVENT_MASK);  
    ...  
}
```

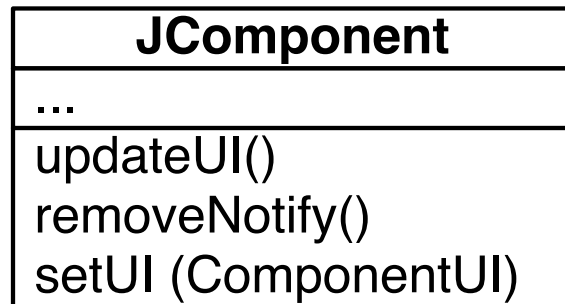


```
public AbstractButton() {  
    super();  
}
```

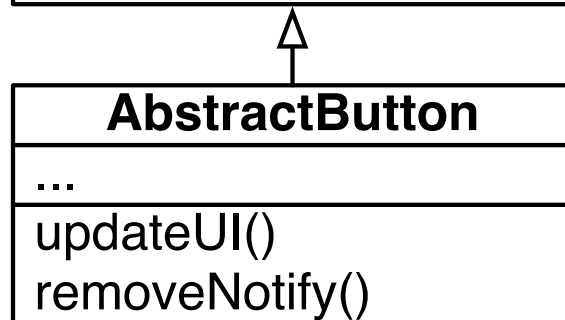


```
public JButton(String text, Icon icon) {  
    // Create the model  
    setModel(new DefaultButtonModel());  
  
    // initialize  
    init(text, icon);  
}
```

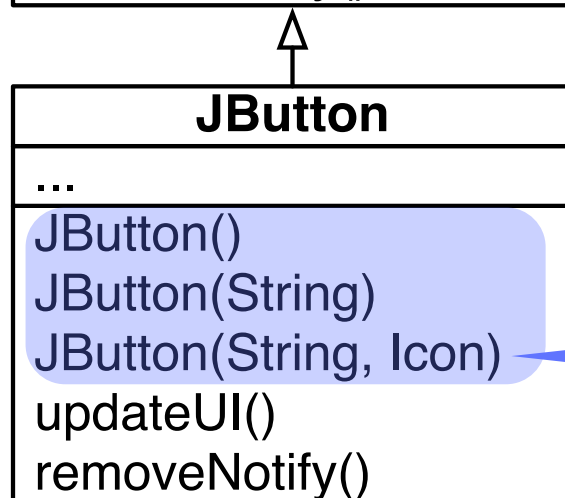

Superclass constructors are sequentially executed



```
public JComponent() {  
    super();  
    enableEvents(AWTEvent.KEY_EVENT_MASK);  
    ...  
}
```



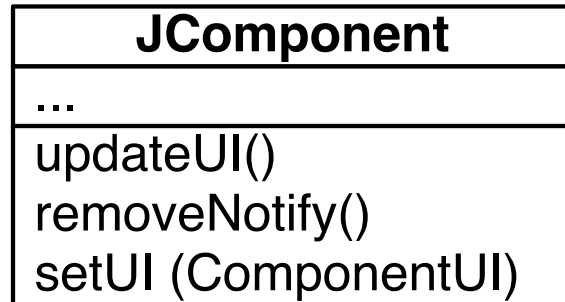
```
public AbstractButton() {  
    super();  
}
```



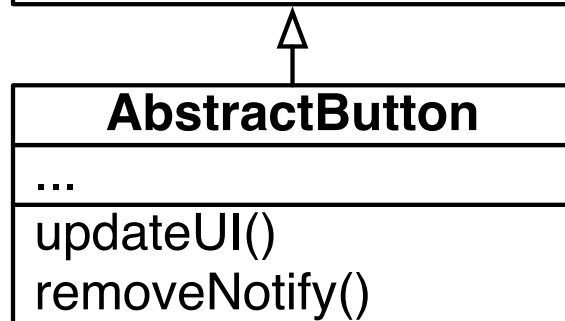
```
public JButton(String text, Icon icon) {  
    super(); “<-- implicit”  
    // Create the model  
    setModel(new DefaultButtonModel());  
  
    // initialize  
    init(text, icon);  
}
```

Error at compilation if no parent construction can be found

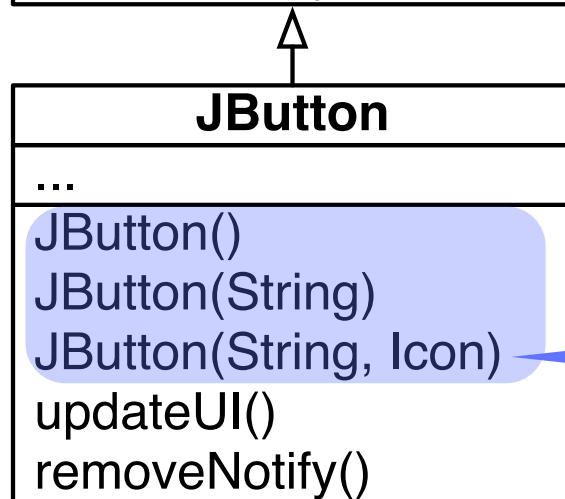
error



```
private JComponent() {  
    super();  
    enableEvents(AWTEvent.KEY_EVENT_MASK);  
    ...  
}
```

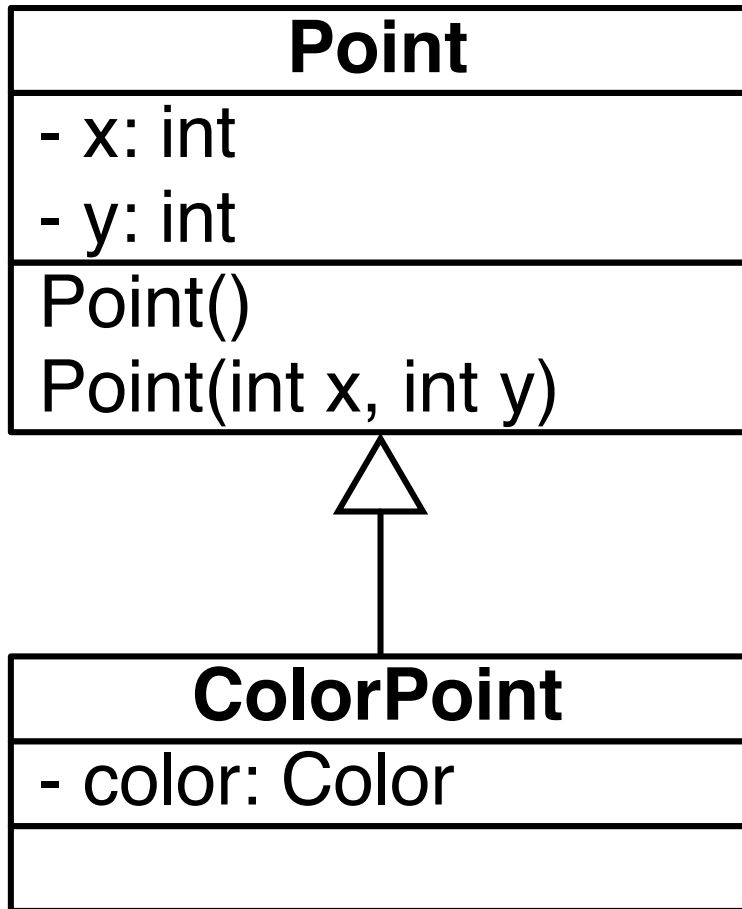


```
public AbstractButton() {  
    super();  
}
```



```
public JButton(String text, Icon icon) {  
    // Create the model  
    setModel(new DefaultButtonModel());  
  
    // initialize  
    init(text, icon);  
}
```

Constructor are not inherited!

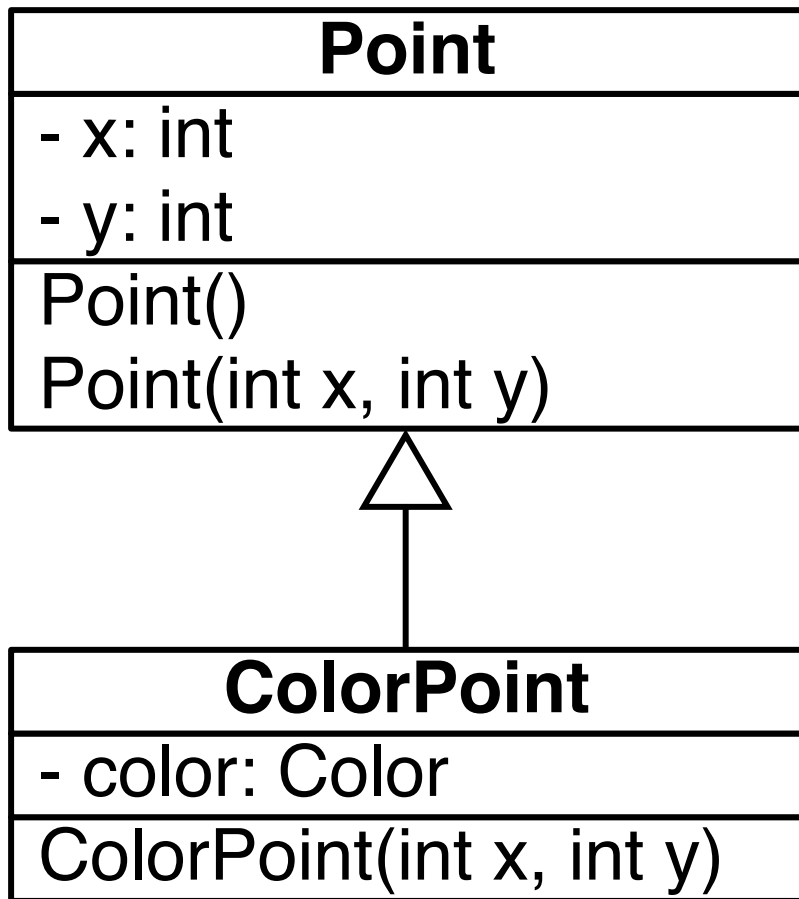


```
new Point()  
new Point(2, 3)  
=> Okay
```

```
new ColorPoint()  
=> Okay (because of the  
default constructor)
```

```
new ColorPoint(2, 3)  
=> Does not compile
```

Missing default constructor

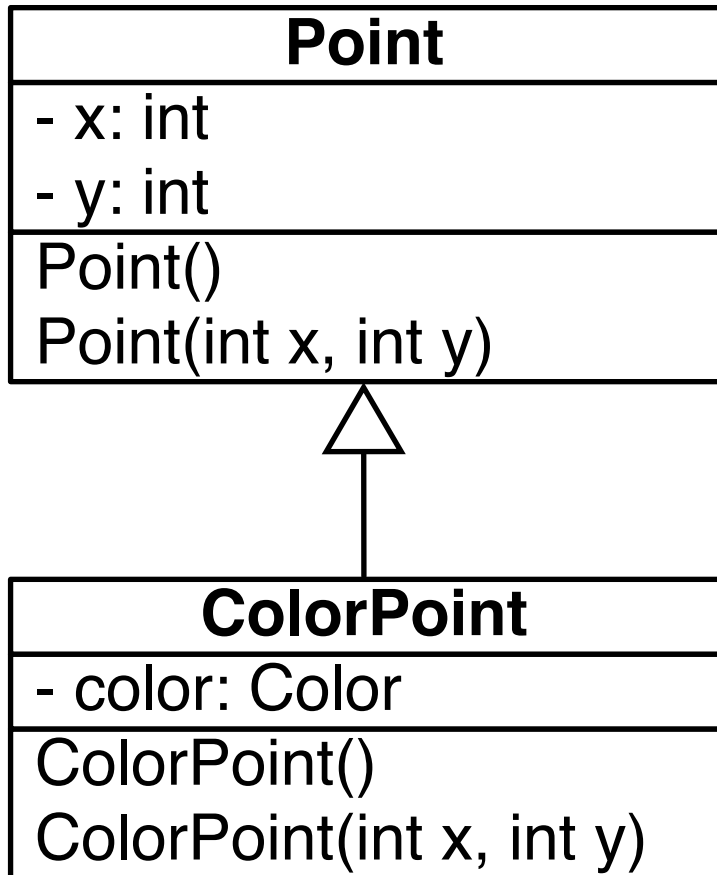


```
new Point()  
new Point(2, 3)  
=> Okay
```

```
new ColorPoint()  
=> Does not compile  
(because there is no  
default constructor)
```

```
new ColorPoint(2, 3)  
=> Okay
```

Missing default constructor

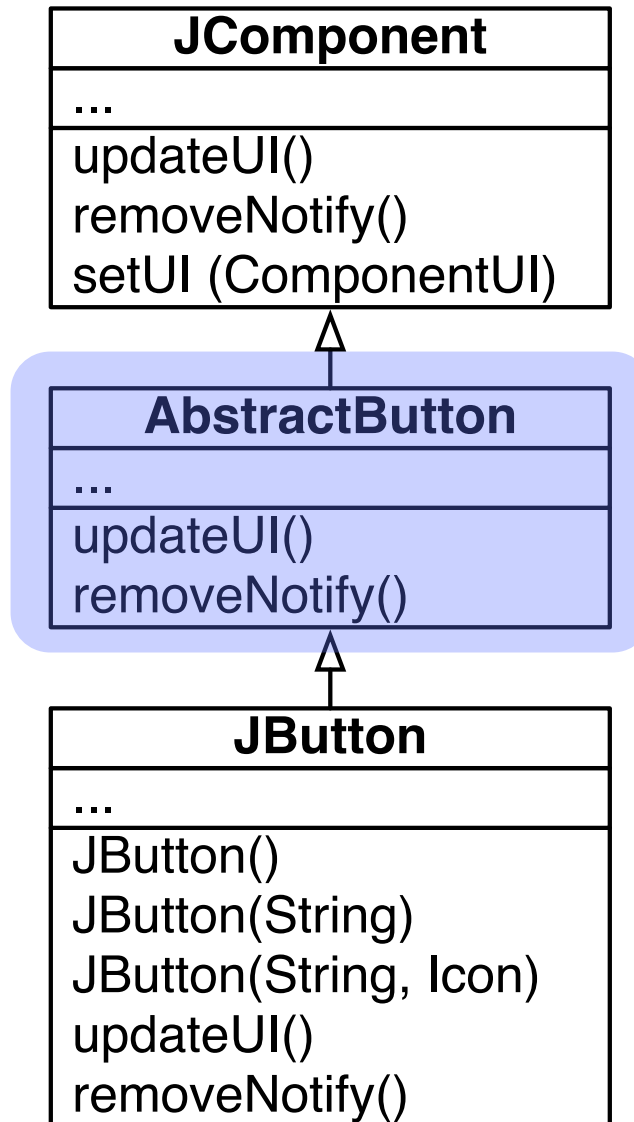


```
new Point()  
new Point(2, 3)  
=> Okay
```

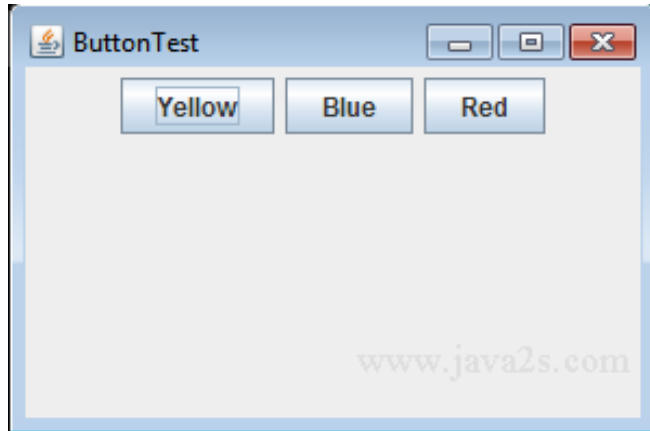
```
new ColorPoint()  
=> Okay
```

```
new ColorPoint(2, 3)  
=> Okay
```

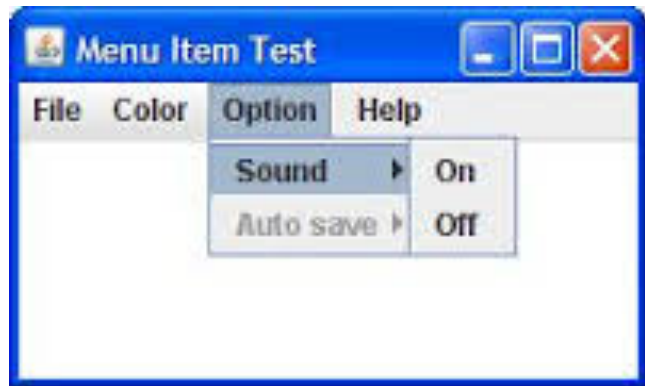
Abstract class



AbstractButton is superclass of 3 classes:



JButton

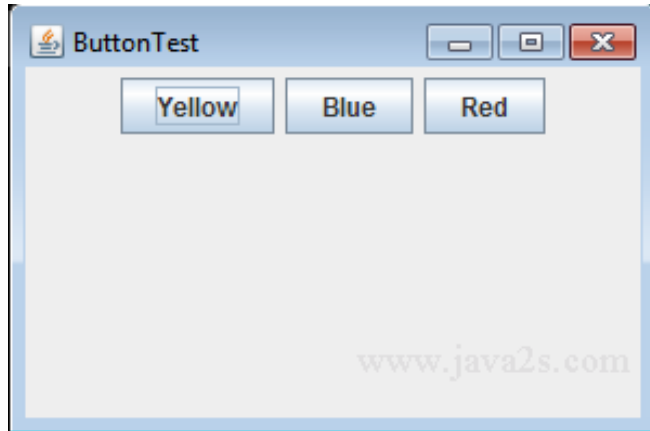


JMenuItem

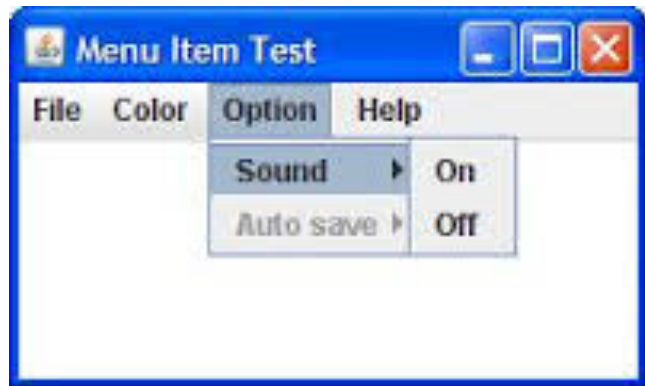


JToggleButton

AbstractButton is superclass of 3 classes:



JButton



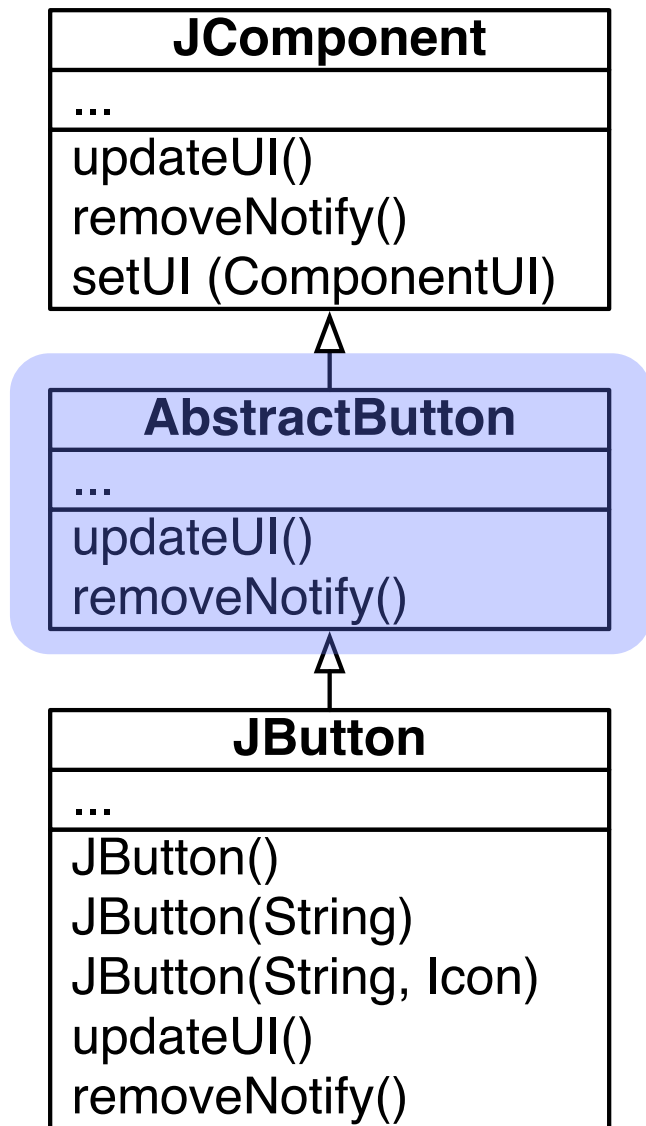
JMenuItem



JToggleButton

AbstractButton is abstract, and each subclass specializes it

Abstract class



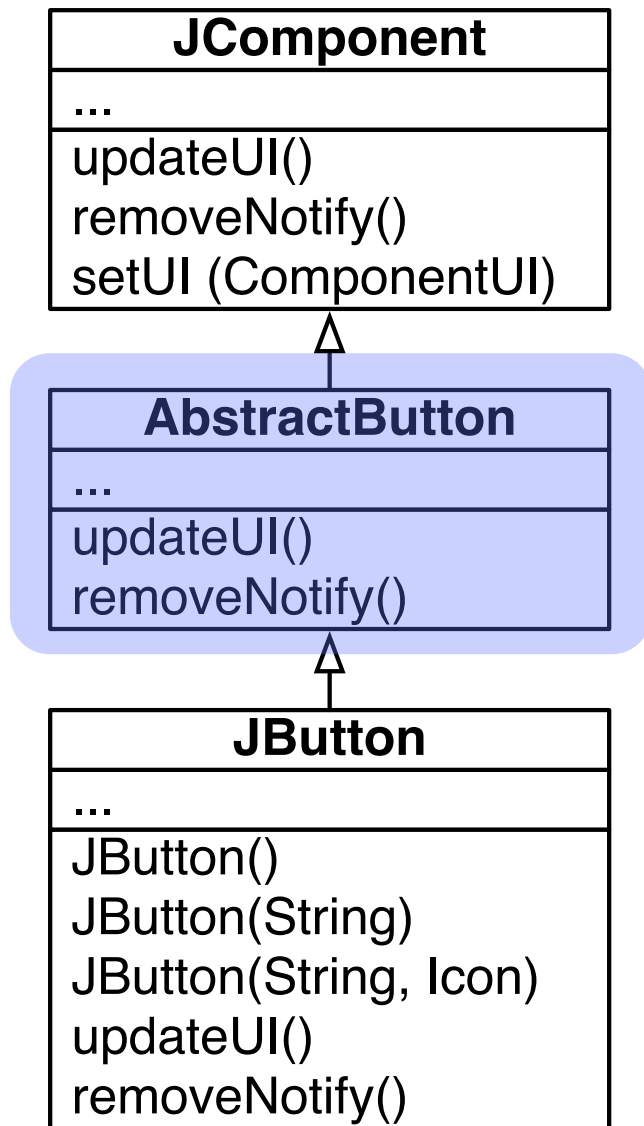
An abstract class is an incomplete class.

It cannot be instantiated therefore.

It is declared as abstract, and may contain abstract methods.

Note on design: abstract classes are not meant to be used as a type. Use interfaces for that purpose!

Abstract class



An abstract class is an incomplete class.

It cannot be instantiated therefore.

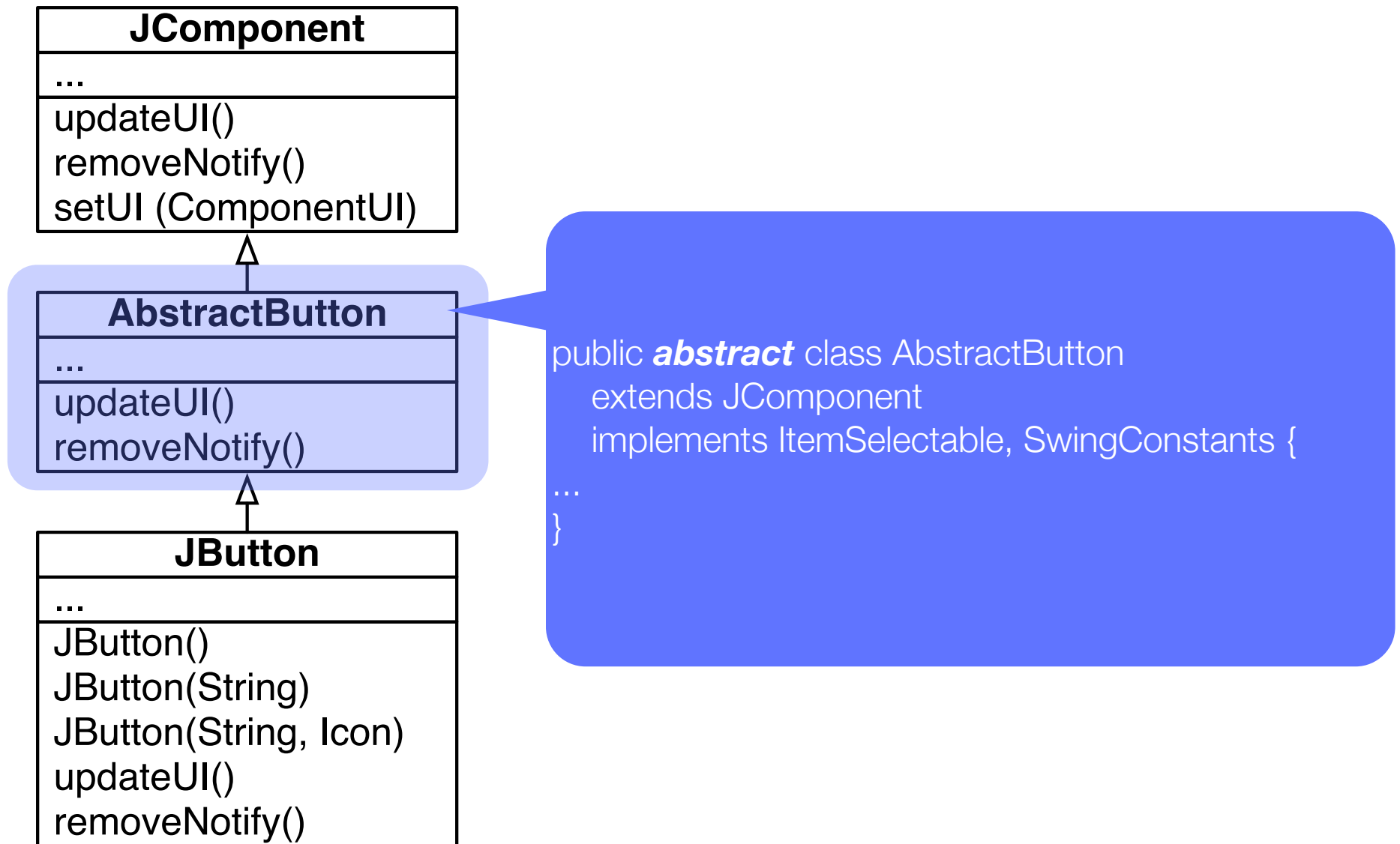
It is declared as abstract, and may contain abstract methods.

Note on design: abstract classes are not meant to be used as a type. Use interfaces for that purpose!

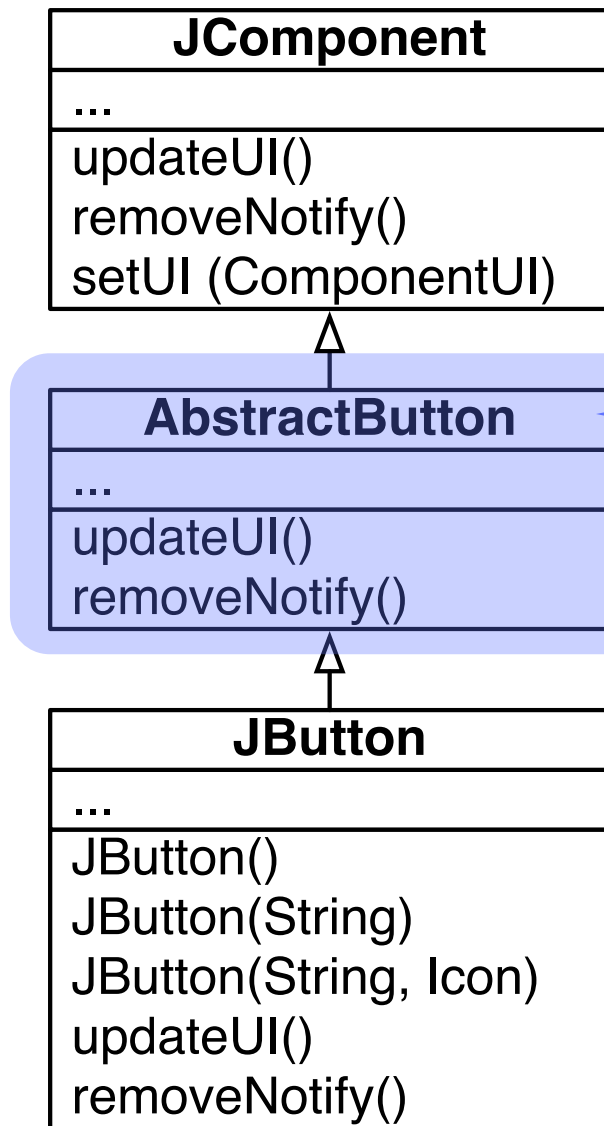
You should **not** write:

```
AbstractButton button = new JButton();
```

Abstract class



Abstract class

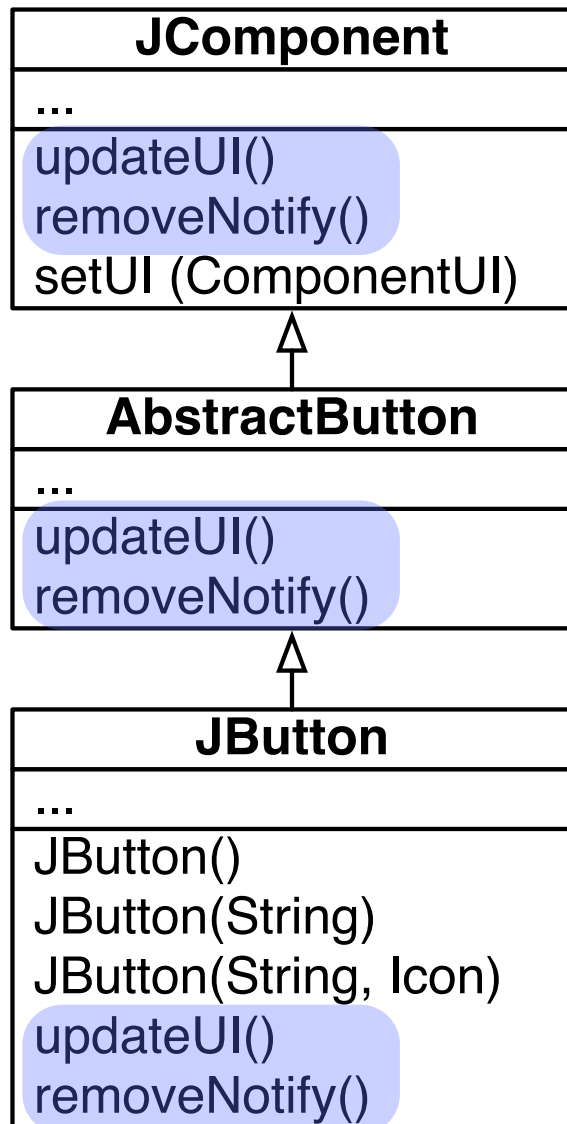


public **abstract** class AbstractButton
extends JComponent
implements ItemSelectable, SwingConstants {
...
}

But we can write:

```
JButton button = new JButton();  
ItemSelectable button = new JButton();
```

Method overriding



Method signature =
name + number of parameters +
type of its parameters

method overriding =
An instance method in a subclass
with the *same signature* as an instance
method in the superclass *overrides*
the superclass's method.

A message send is always associated
to a method signature. The signature
is used to find the method to execute.

Method overloading

JComponent
...
repaint(long tm, int x, int y, int w, int h)
repaint(Rectangle r)

Two methods defined in a hierarchy
can have the same name
but different signatures

Method overloading

JComponent
...
repaint(long tm, int x, int y, int w, int h)
repaint(Rectangle r)

Two methods defined in a hierarchy
can have the same name
but different signatures



**Could generate
complicated
bugs**



A typical situation

Movie
title
authors
...
equals(Movie aMovie)

```
public boolean equals(Movie aMovie) {  
    return aMovie.title.equals(this.title) &&  
        aMovie.authors.equals(this.authors);  
}
```


A typical situation

```
String t = "The Godfather";  
String a = "Coppola"
```

Movie
title
authors
...
equals(Movie aMovie)

```
Movie m1 = new Movie(t, a);  
Movie m2 = new Movie(t, a);
```

```
m1.equals(m2);
```

```
=> true
```

A typical situation

```
String t = "The Godfather";  
String a = "Coppola"
```

Movie
title
authors
...
equals(Movie aMovie)

```
Object m1 = new Movie(t, a);  
Movie m2 = new Movie(t, a);
```

```
m1.equals(m2);
```

```
=> false
```

A typical situation

```
String t = "The Godfather";  
String a = "Coppola"
```

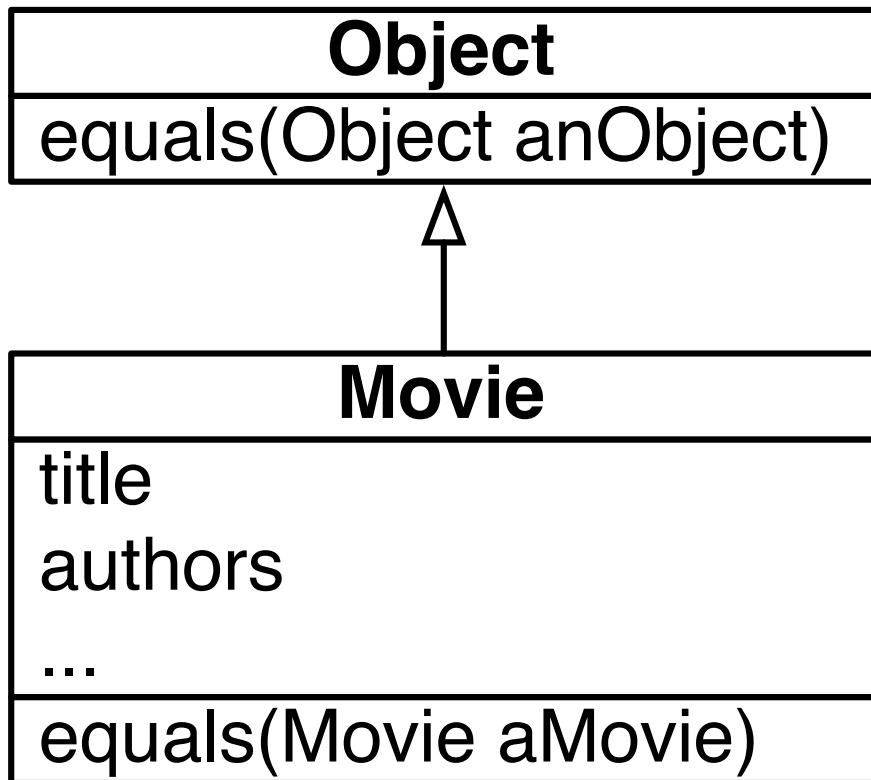
Movie
title
authors
...
equals(Movie aMovie)

```
Movie m1 = new Movie(t, a);  
Object m2 = new Movie(t, a);
```

```
m1.equals(m2);
```

```
=> false
```

Method signature are determined based on the type information



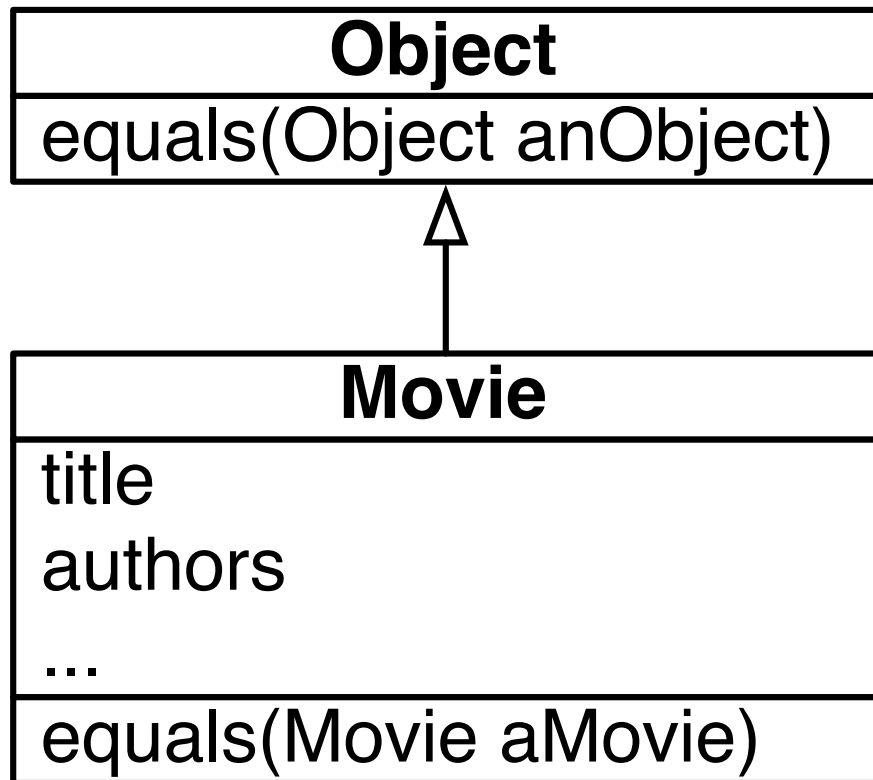
```
String t = "The Godfather";  
String a = "Coppola"
```

```
Movie m1 = new Movie(t, a);  
Object m2 = new Movie(t, a);
```

```
m1.equals(m2);
```

```
=> false
```

Method signature are determined based on the type information



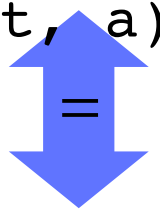
`equals(Object)`

```
String t = "The Godfather";
String a = "Coppola"
```

```
Movie m1 = new Movie(t, a);
Object m2 = new Movie(t, a);
```

```
m1.equals(m2);
```

```
=> false
```



An acceptable version of equals

```
public class Movie {
    private String title;
    private String author;

    public Movie(String title, String author) {
        this.title = title;
        this.author = author;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Movie) {
            return ((Movie)o.title).equals(this.title) &&
                ((Movie)o.author).equals(this.author);
        }
        return false;
    }
}
```

Use method overloading with care!

The example presented earlier shows a typical situation where

defining equals(Movie) is not an *overriding*, but an *overloading*

two different methods are therefore accessible

the method signature to lookup is statically determined (by the compiler)

In Java, the overloading is statically resolved

Method overloading is complex and should be used with care

@Override

This problem of accidental overloading is so important that it is recommended to explicitly declare overriding methods:

```
@Override public boolean equals(Object other) {...}
```


What we have seen today

Constructors

- May invoke themselves

- Constructor of the superclass is always executed before `super()` implicit (if not provided, then the system does it for you)

- If no constructor is provided, then the system defines one

Abstract class

- Incomplete class

- used to be specialized when subclassed

Instance methods (i.e., a method without **static**)

- May be overridden (same signature in a subclass)

- May be overloaded (different signature, but same name)

What we have seen today

A signature is always determined from a method call

The signature is used to look for a method during the program execution

What you should know

How the *method lookup* works

How the *object initialization* works

When should you *use method overloading*

The difference between *overloading* and *overriding*

What an *abstract class* is for

Can you answer to this questions?

Why *method overloading* may be difficult to use?

What are the condition to have *super.getClass() == this.getClass()* return true?

Why *frameworks* provides some *abstract classes*?

What is the difference between a *constructor* and a *method*?

Why a *constructor* always *call* a *parent constructor*?

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