Inner Classes

Inner Classes

- The Java programming language allows you to define a class within another class.

```
class OuterClass {
        ...
        class InnerClass {
                ...
        }
}
```

Why Use Inner Classes?

- There are several reasons for using inner classes:

    1. It is a way of **logically grouping** classes that are only used in one place.

        - If a class is useful to only one other class,
          then it is logical to embed it in that class and keep the two together.

        - Nesting such "helper classes" makes their package more streamlined.

Why Use Inner Classes?

- There are several reasons for using inner classes:

2. It **increases encapsulation**.

    - Consider two top-level classes, A and B, where B needs access to private members of A.
      By hiding class B within class A,
      A's members can be declared private and B can access them.

    - In addition, B itself can be hidden from the outside world.

Why Use Inner Classes?

- There are several reasons for using inner classes:

3. Nested classes can lead to **more readable** and **maintainable** code.

    - Nesting small classes within top-level classes places the code closer to where it is used.

Types of Inner Classes

- There are broadly four types of inner classes:

    1. Normal Member Inner Class
    2. Static Member Inner Class
    3. Local Inner Class (inside method body)
    4. Local Anonymous Inner Class

1. Normal Member Inner Class

```
public class OuterClass{
  private int x ;
  public void myMethod(){
       MyInnerClass m = new MyInnerClass();
       ..
  }

  class MyInnerClass{
       public void aMethod(){
            //you can access private members of the outer class here
             x = 3 ;
       }
  }
}
```

1. Normal Member Inner Class

- An inner class can extend any class and/or implement any interface.
- An inner class can assume any accessibility level:
  - private, (friendly), protected, or public.
- An inner class can have an inner class inside it.
- When you compile the .java file, two class files will be produced:
  - MyClass.class
  - MyClass$MyInnerClass.class
- The inner class has an implicit reference to the outer class.

1. Normal Member Inner Class

The inner class has an implicit reference to the outer class

```
public class MyClass{
   private int x ;
   public void myMethod(){
        MyInnerClass m = new MyInnerClass();
   }
   class MyInnerClass{
        int x ;
        public void aMethod(){
             x = 10 ;  //x of the inner class
             MyClass.this.x = 25 ;  // x of the outer class
        }
   }
}
```

2. Static Inner Class

- You know, the normal inner class implicitly has a reference to the outer class that created it.
  - If you don't need a connection between them, then you can make the **inner class static or a Nested class**.
- **A static inner class** means:
  - You don't need an outer-class object to create an object of a static inner class.
  - You can't access an outer-class object from an object of a static inner class.

2. Static Inner Class

- Static Inner Class:
  - is among the static members of the outer class.
  - When you create an object of static inner class, you don’t need to use an object of the outer class (remember, it’s static!).
  - Since it is static, such inner class will only be able to access the static members of the outer class.

2. Static Inner Class (Example)

```
public class OuterClass{
   int x ;
   static int y;
   InnerClass i = mew InnerClass();
   public static class InnerClass{
        public void aMethod(){
               y = 10;                  // OK
               x = 33;                  // wrong
        }
   }
}
```

```
OuterClass.InnerClass ic = new OuterClass.InnerClass();
```

3. Local Inner Class

```
public class MyClass {
   private int x ;
   public void myMethod(final String str, final int a){
        final int b = 5;
        class MyLocalInnerClass{
            public void aMethod(){
                //you can access private members of the outer class
                //and you can access final local variables of the method
            }
        }
       MyLocalInnerClass myObj = new MyLocalInnerClass();
    }
}
```

3. Local Inner Class

- The object of the local inner class can **only be created** below the definition of the local inner class (**within the same method**).
- The local inner class can access the member variables of the outer class.
- It can also access the local variables of the enclosing method **if they are declared final.**

4. Anonymous Inner Class

```
public class MyClass extends JFrame
{
   int x ;
   public MyClass()
   {
     Thread th = new Thread(new Runnable()
                                {
                                    public void run()
                                    {
                                        ..
                                    }
                                });
     th.start();
   }
}
```

4. Anonymous Inner Class

- The whole point of using an anonymous inner class is to **implement an interface** or **extend a class** and then override one or more methods.

- Of course, it does not make sense to define new methods in an anonymous inner class; how will you invoke them?

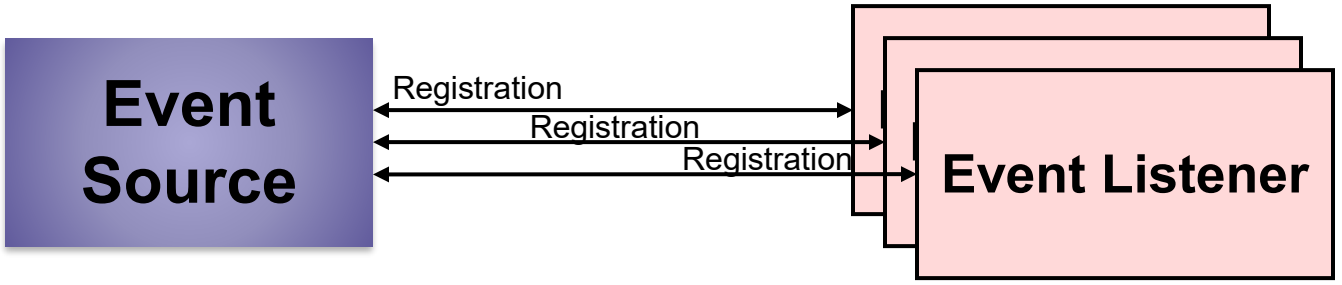
- When you compile the .java file, two class files will be produced:
  - MyClass.class
  - MyClass$1.class

Event Handling

Event Handling

- Event Delegation Model was introduced to Java since JDK 1.1
- This model realizes the event handling process as two roles:
  - Event Source
  - Event Listener.
- The Source:
  - is the object that fired the event,
- The Listener:
  - is the object that has the code to execute when notified that the event has been fired.

Event Handling



- An Event Source may have one or more Event Listeners.
- The advantage of this model is:
  - The Event Object is only forwarded to the listeners that have registered with the source.

Event Handling

- When working with GUI, the source is usually one of the GUI Components (e.g. Button, Checkbox, …etc).
- The following piece of code is a simplified example of the process:

```
Button b = new Button("Ok"); //Constructing a Component (Source)
MyListener myL = new MyListener(); //Constructing a Listener
b.addActionListener(myL); //Registering Listener with Source
```

Event Handling

- Let’s look at the signature of the registration method:

```
void addActionListener(ActionListener l)
```

- In order for a listener to register with a source for a certain event,
  - it has to implement the proper interface that corresponds to the designated event, which will enforce a certain method to exist in the listener.

Event Handling Example

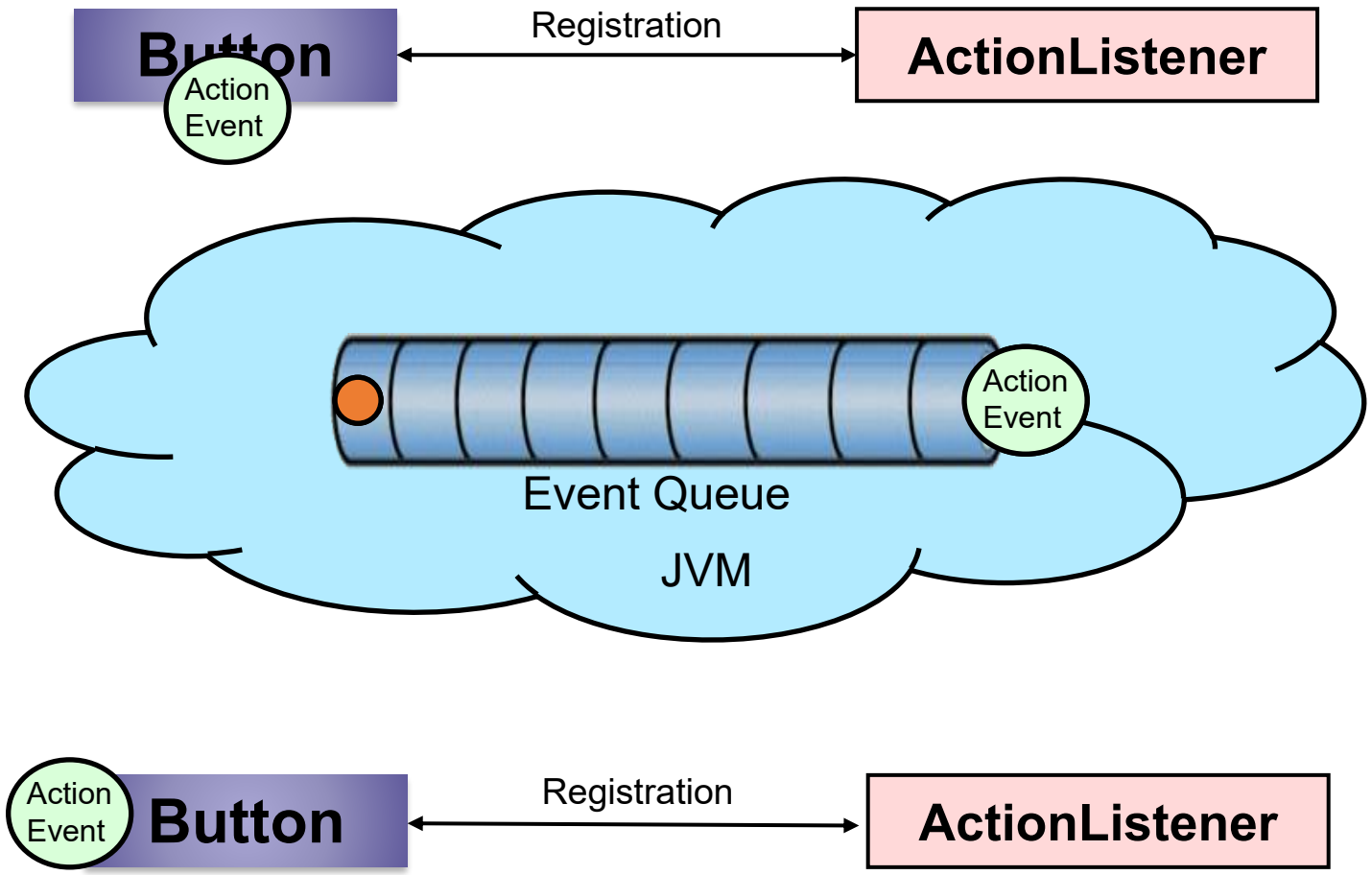
1. Write the listener code:

```
class MyListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
        // handle the event here
        // (i.e. what you want to do
        // when the Ok button is clicked)
   }
}
```

2. Create the source, and register the listener with it:

```
public class MyFrame extends JFrame{
       public MyFrame(){
            Button b = new Button("Ok");
            MyListener myL = new MyListener();
            b.addActionListener(myL);
       }
}
```

Event Dispatching Thread



Event Dispatching Thread

- When examining the previous button example:
  - The button (**source**) is created.
  - The listener is registered with the button,
  - The user clicks on the Ok button.
    - An ActionEvent object is created and placed on the **event queue**.
  - The **Event Dispatching Thread** processes the events in the queue.
    - The Event Dispatching Thread checks with the button to see if any listeners have registered themselves.
  - The Event Dispatcher then invokes the actionPerformed(..) method, and passes the ActionEvent object itself to the method.

Event Handling Example

- Having a JFrame that includes two buttons.
- "++" button: is supposed to increment the value shown on the frame.
- "--" button: is supposed to decrement the value shown on the frame.

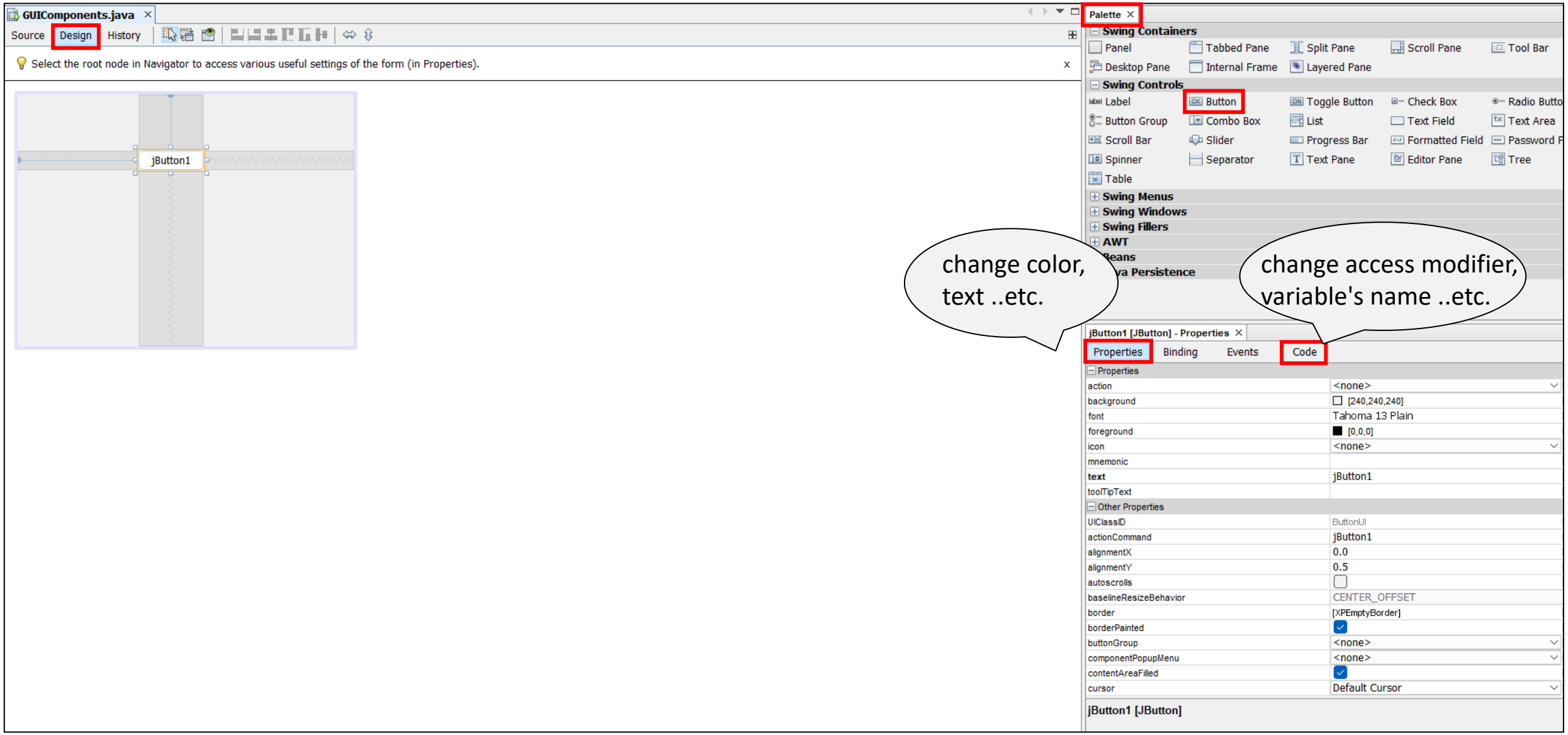
Event Handling Example

```
public class ButtonApp extends JFrame{
   int x;
   Button b;
   public ButtonApp(){
        b = new Button("Click Me");
        b.addActionListener(new MyButtonListener());
        setLayout(new FlowLayout());
        add(b);
   }
   public void paint(Graphics g){
        super.paint(g);
        g.drawString("Click Count is:" + x, 50, 200);
   }
   class MyButtonListener implements ActionListener{
        public void actionPerformed(ActionEvent ev){
            x++ ;
            repaint() ;
        }
   }
}
```

Event Handling Example

```
public class ButtonApp extends JFrame{
   int x;
   JButton b;
   public ButtonApp(){
        b = new JButton("Click Me");
        b.addActionListener(
                new ActionListener(){
                        public void actionPerformed(ActionEvent ev){
                                x++ ;
                                repaint() ;
                        }
                });
        setLayout(new FlowLayout());
        add(b);
   }
   public void paint(Graphics g){
       super.paint(g);
       g.drawString"Click Count is:" + x, 50, 200);
   }
}
```

Adding GUI Components - NetBeans



Lab Exercises

1. Button Count

- Create a JFrame that has two buttons
  → one to increment the counter value
  → and one to decrement this value.

2. Draw Single Line

- Create a JFrame that allows the user to draw one line by dragging the mouse on the JFrame.

- Hint: Use **`MouseMotionListener`**