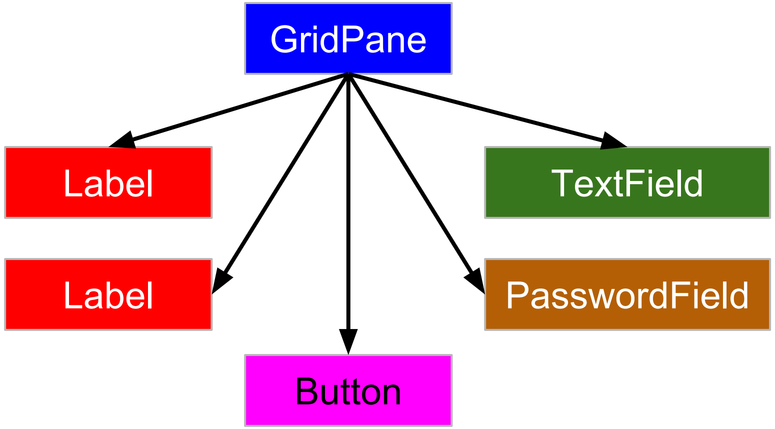
Graphical User Interfaces (GUIs)

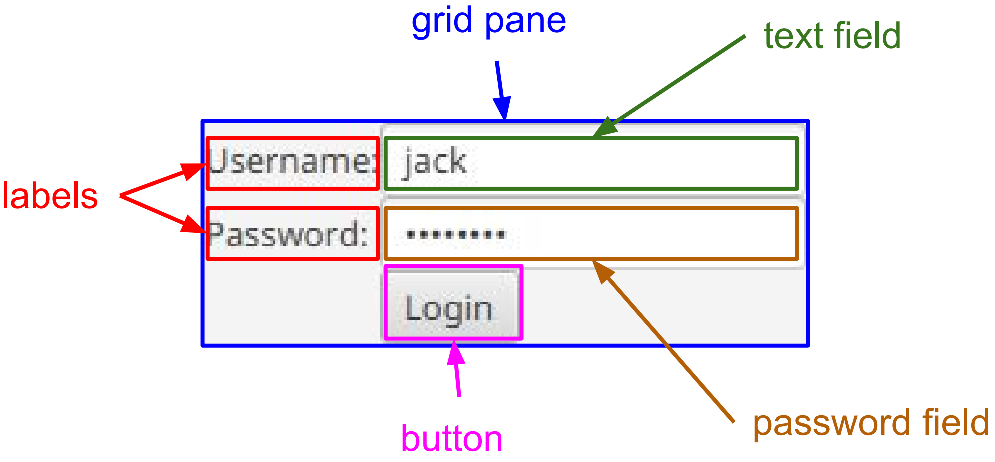
History of GUI technology in Java

- (1995) AWT
- (1998) Swing
- (2011) JavaFX

JavaFX Concepts

- A **node** is a graphical object (e.g. a Button, TextField, Label, GridPane).
- A **scene** is a tree of nodes.





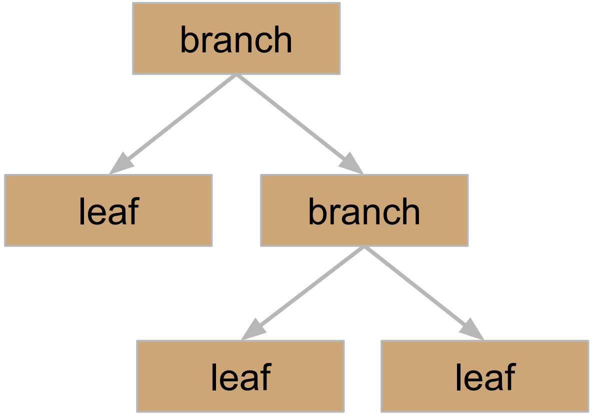
Username: jack

Password: •••••••••

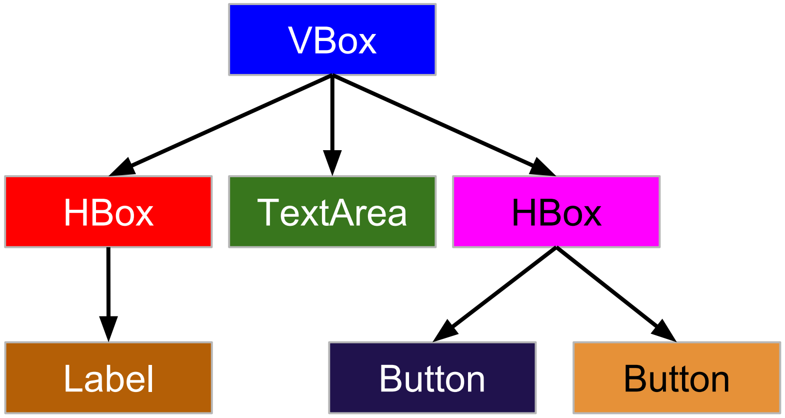
- A **stage** is a place to display a scene (typically a window).
- An **application** has a main method. It sets up and shows the primary stage.

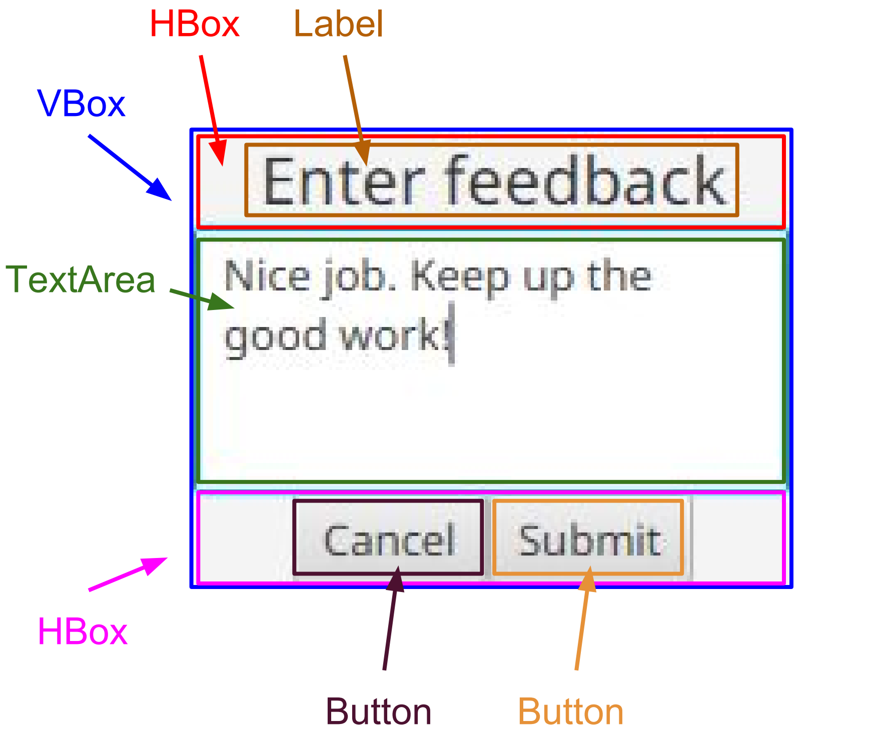
The scene graph

- A scene is a tree of nodes.
- Each node is either a branch or a leaf.
    - A branch node can have children
      e.g. GridPane, HBox, VBox
    - A leaf node cannot have children
      e.g. Button, Label, TextField



Nested branches





Nice job. Keep up the good work!

Packages to import

- **Nodes**:

```
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.text.*;
import javafx.scene.image.*;
```

- **Scene**:

```
import javafx.scene.*;
```

- **Stage**:

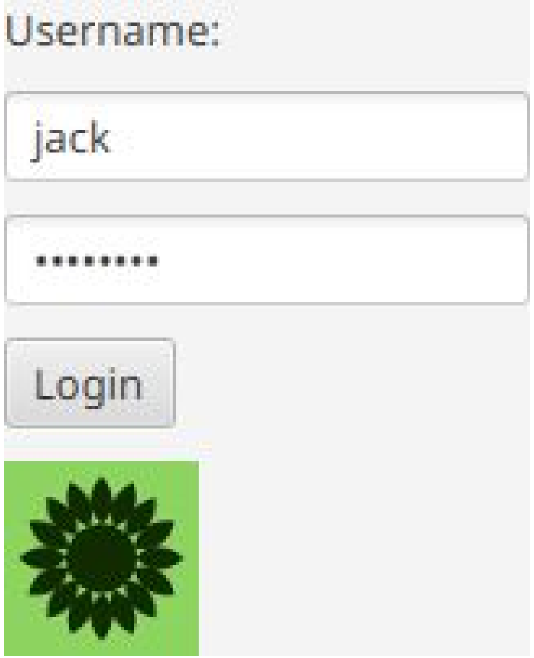
```
import javafx.stage.*;
```

- **Application**:

```
import javafx.application.*;
```

Leaf nodes

```
Label usernameLbl = new Label(“Username:”);

TextField usernameTf = new TextField();

PasswordField passwordPf = new PasswordField();

Button loginBtn = new Button(“Login”);

ImageView flowerIv = new ImageView(“flower.png”);
```



jack

****

Branch nodes - VBox

- A VBox lays out its children in a vertical box.
- Create a VBox with 10 pixel spacing:

```
VBox box = new VBox(10);
```

- Add the the children one by one:

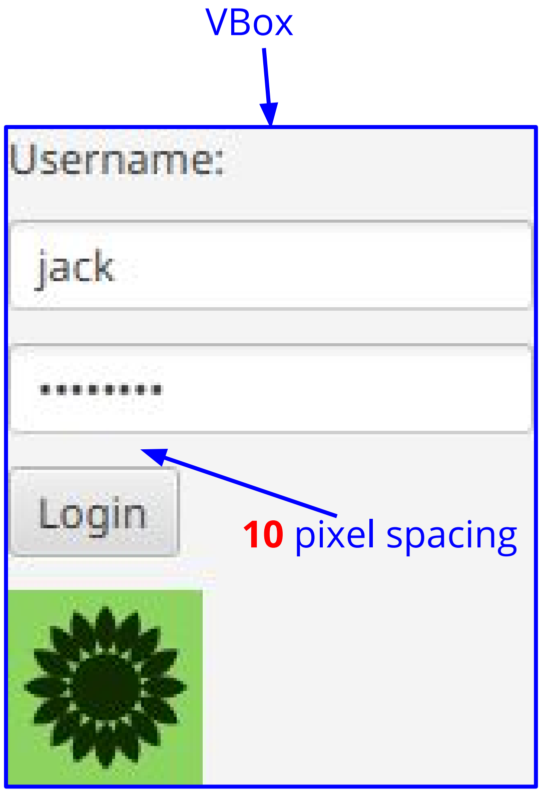
```
box.getChildren().add(usernameLbl);
box.getChildren().add(usernameTf);
box.getChildren().add(passwordPf);
```

- Or add many children at once:

```
box.getChildren().addAll(loginBtn, flowerIv);
```

- Or Create a VBox with children:

```
VBox box = new VBox(10, usernameLbl, usernameTf, passwordPf, loginBtn, flowerIv);
```



VBox

Username:

jack

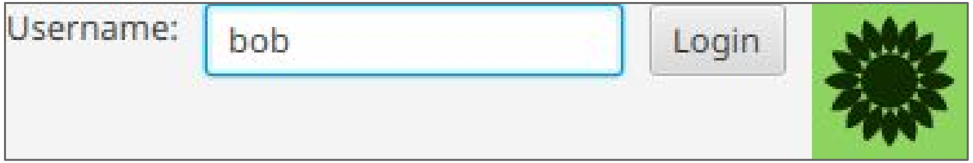
****

10 pixel spacing

Branch nodes - HBox

- An HBox lays out its children in a horizontal box.

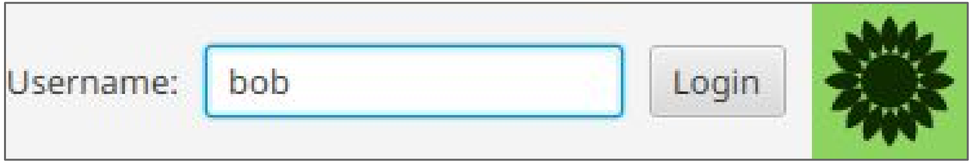
-
  ```
  HBox box = new HBox(10);
  box.getChildren().addAll(usernameLbl, usernameTf, loginBtn, flowerIv);
  ```



Username: bob

- Align with setAlignment:

  ```
  box.setAlignment(Pos.CENTER);
  ```



Username: bob

Branch nodes - Alignment

- `import javafx.geometry.*;`
  `box.setAlignment(position);`
- Valid positions:
  - `Pos.CENTER`
  - `Pos.CENTER_LEFT`
  - `Pos.CENTER_RIGHT`
  - `Pos.TOP_CENTER`
  - `Pos.BOTTOM_CENTER`
  - `Pos.TOP_LEFT`
  - `Pos.TOP_RIGHT`
  - `Pos.BOTTOM_LEFT`
  - `Pos.BOTTOM_RIGHT`

For more, see: https://docs.oracle.com/javase/8/javafx/api/javafx/geometry/Pos.html

Branch nodes - GridPane

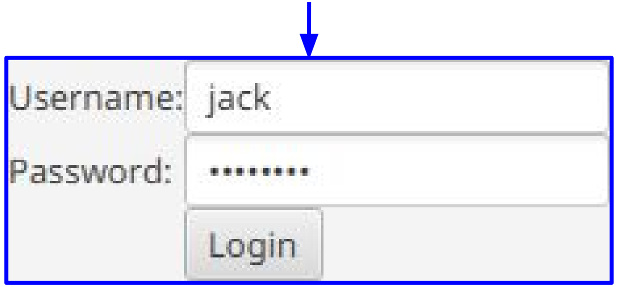
- A GridPane lays out its children in a grid of rows and columns.
- Create a GridPane:

```
GridPane grid = new GridPane();
```

- Add children to the grid:

column row

```
grid.add(usernameLbl, 0, 0);
grid.add(passwordLbl, 0, 1);
grid.add(usernameTf, 1, 0);
grid.add(passwordPf, 1, 1);
grid.add(loginBtn, 1, 2);
```



GridPane

Username: jack

Password: ••••••••

Application class

- The main class extends Application.
    - It defines a main method.
    - It overrides the start method.

```
public class BankApplication extends Application {
    public static void main(String[] args) { launch(args); }
    @Override
    public void start(Stage stage) throws Exception {
        ... code to set up and show the stage ...
    }
}
```

Setup code - 1. Create the leaves

```
public class BankApplication extends Application {
    private Label usernameLbl;
    private Label passwordLbl;
    private TextField usernameTf;
    private PasswordField passwordPf;
    private Button loginBtn;

    @Override public void start(Stage stage) throws Exception {
        usernameLbl = new Label("Username:");
        passwordLbl = new Label("Password:");
        usernameTf = new TextField();
        passwordPf = new PasswordField();
        loginBtn = new Button("Login");
        ...
```

Each leaf node is a field

Initialise each leaf in the start() method

Setup code - 2. Add the leaves to a branch

```
@Override public void start(Stage stage) throws Exception {
    ...
    GridPane gridPane = new GridPane();
    gridPane.add(usernameLbl, 0, 0);
    gridPane.add(passwordLbl, 0, 1);
    gridPane.add(usernameTf, 1, 0);
    gridPane.add(passwordPf, 1, 1);
    gridPane.add(loginBtn, 1, 2);
    ...
```

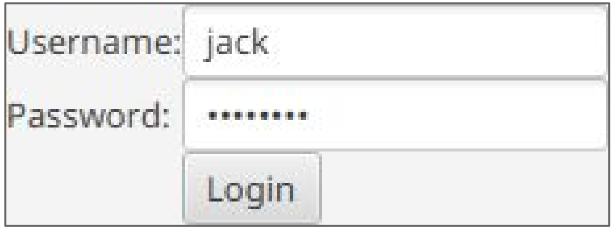
Setup code - 3. Set the scene, show the stage

```
@Override public void start(Stage stage) throws Exception {
    ...
    stage.setScene(new Scene(gridPane));
    stage.setTitle(“Login”);
    stage.show();
}
```

The root node of the scene

The window title

Show the window



Username: jack

Password: •••••••••

New Patterns and Syntax

Required new patterns and syntax:

1. The Observer Pattern
2. Inner Classes
3. Anonymous Inner Classes
4. Lambda Expressions

1. The Observer Pattern

1. The Observer Pattern

- **Goal**: Observers are notified whenever a subject changes.

Examples:

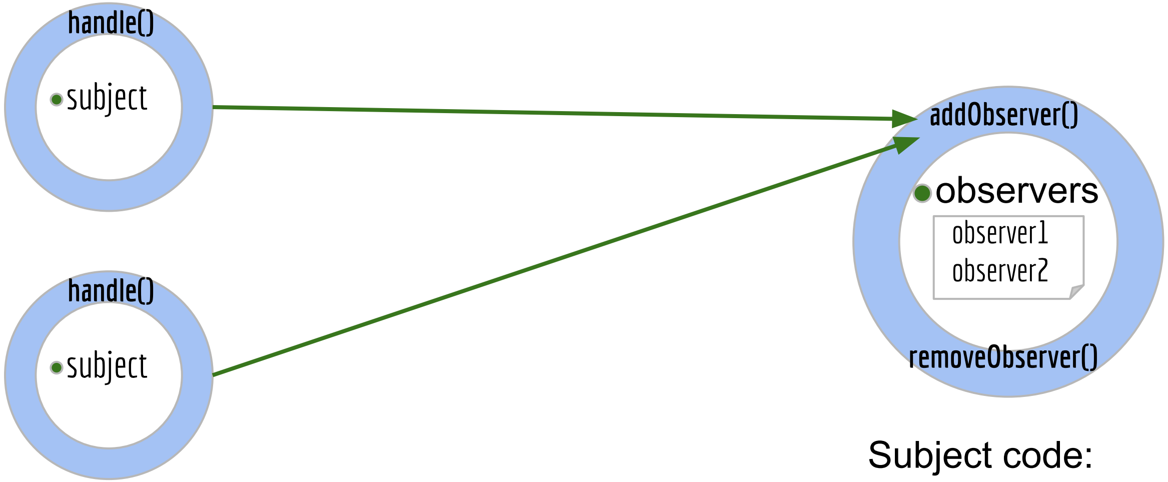
- A Button notifies you when it is clicked.
- A File notifies you when it is modified.
- A Product notifies you when it is sold.

Solution. The solution has two phases:

Phase 1. Observers register with the subject.
Phase 2. When something happens to the subject, it notifies the observers.

The Observer Pattern

- **Phase 1 (registration)**: Each observer registers to be notified.



Observer code:

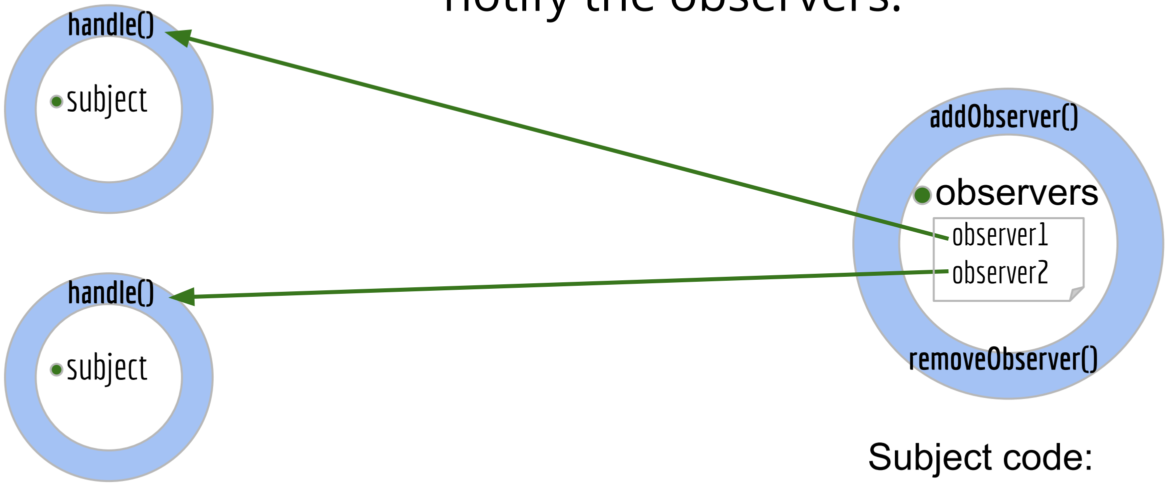
```
subject.addObserver(this);
```

Subject code:

```
public void addObserver(Observer o) {
     observers.add(o);
}
```

The Observer Pattern

- **Phase 2 (notification)**: When something happens to the subject, notify the observers.



Observer code:

```
public void handle() {
     do something in response
}
```

Subject code:

```
for (Observer o : observers)
     o.handle();
```

The Observer Interface

- An observer is any object that can handle the notification.
- Define an interface:

```
public interface Observer {
    void handle();
}
```

- An observer is any object that implements this interface.
- Each observer implements the `handle()` method to achieve its own goal.

Example

Notify observers when a product is sold

The Observer

- Observers want to be notified when a product is sold. Define an interface:

```java
public interface ProductObserver {
    void handleSale(double money);
}
```

- The CashRegister is an observer:

```java
public class CashRegister implements ProductObserver {
    private double cash;
    @Override public void handleSale(double money) {
        cash += money;
    }
}
```

Phase 1: Registration

```
public class Store {
  private Product product;
  private CashRegister cashRegister;

  public Store() {
    product = new Product();
    cashRegister = new CashRegister();
    product.addObserver(cashRegister);
  }
}
```

```
public class Product {
  private LinkedList<ProductObserver>
      observers = new LinkedList<ProductObserver>();

  public void addObserver(ProductObserver o) {
    observers.add(o);
  }

  public void removeObserver(ProductObserver o) {
    observers.remove(o);
  }
}
```

Phase 2: Notification

Rule: Whenever the Product changes
The Product notifies the observers.

```
public class Product {
    ...
    public void sell(int n) {
        sold += n;
        double money = n * price;
        for (ProductObserver observer : observers)
            observer.handleSale(money);
    }
}
```

A field in the Product changed

So notify the observers

```
public class CashRegister implements ProductObserver {
    private double cash;
    @Override public void handleSale(double money) {
        cash += money;
    }
}
```

2. Inner Classes

2. Inner Classes

- An **inner class** is a class defined inside another class.
- An inner class can access all members of the outer class.
- An inner class offers **better encapsulation**:
    - x and foo can be hidden from the outside but shared with the inner class.
    - The inner class can also be hidden from the outside.

```
public class OuterClass {
    private int x;
    private void foo() { x++; };
    private class InnerClass {
        public void bar() {
            foo();
            System.out.println(x);
        }
    }
}
```

Example

```
public class Store {
    private Product product;
    private CashRegister cashRegister;
    public Store() {
        product = new Product();
        cashRegister = new CashRegister();
        product.addObserver(cashRegister);
        product.addObserver(new SalePrinter());
    }
    private class SalePrinter implements ProductObserver {
        @Override public void handleSale(double money) {
            System.out.println(“You paid $” + money);
        }
    }
}
```

3. Anonymous Inner Classes

3. Anonymous inner classes

- An interface cannot be instantiated since it has no implementation:

✗

```
new ProductObserver()
```

- However, you can provide the implementation while instantiating it:

✓

```
new ProductObserver() {
    @Override public void handleSale(double money) {
        System.out.println(“You paid $” + money);
    }
}
```

- Same as defining a class that implements the interface, then creating a new instance of that class.
  Except the class has no name. Hence, it is “anonymous”.

Example

```
public class Store {
    private Product product;
    private CashRegister cashRegister;
    public Store() {
        product = new Product();
        cashRegister = new CashRegister();
        product.addObserver(cashRegister);
        product.addObserver(new ProductObserver() {
            @Override public void handleSale(double money) {
                System.out.println(“You paid $” + money);
            }
        });
    }
}
```

4. Lambda Expressions

Lambda Expressions (Java 8)

- Anonymous inner classes with one method are very common.

```
new ProductObserver() {
     @Override public void handleSale(double money) {
          System.out.println("You paid $" + money);
     }
}
```

- This is a LOT of syntax for just one method!

- A lambda expression is a shorter way to write such a method:

```
money -> System.out.println("You paid $" + money)
```

Method parameter

Method body

Lambda Expressions (Java 8)

- A body with one statement has no braces or semicolon:

```
money -> System.out.println(“Sale: $” + money)
```

- Curly braces enclose a block of code. Each statement has a semicolon:

```
money -> {
    String moneyStr = formatted(money);
    System.out.println(”Sale: $” + moneyStr);
}
```

- Multiple parameters are enclosed in parentheses:

```
(param1, param2, param3) -> body
```

Example

```
public class Store {
    private Product product;
    private CashRegister cashRegister;
    public Store() {
        product = new Product();
        cashRegister = new CashRegister();
        product.addObserver(cashRegister);
        product.addObserver(
                     money -> System.out.println(“You paid $” + money)
                 );
    }
}
```

Which one should I use?

- Use a lambda expression if the class has one method and is used once.
- Use an anonymous inner class if the class has multiple fields/methods.
- Use an inner class if you also need to create more than one instance.
- Use a normal class if you also need to access it from other classes
  (or if you anticipate needing to)

Event-driven programming

Event-driven programming

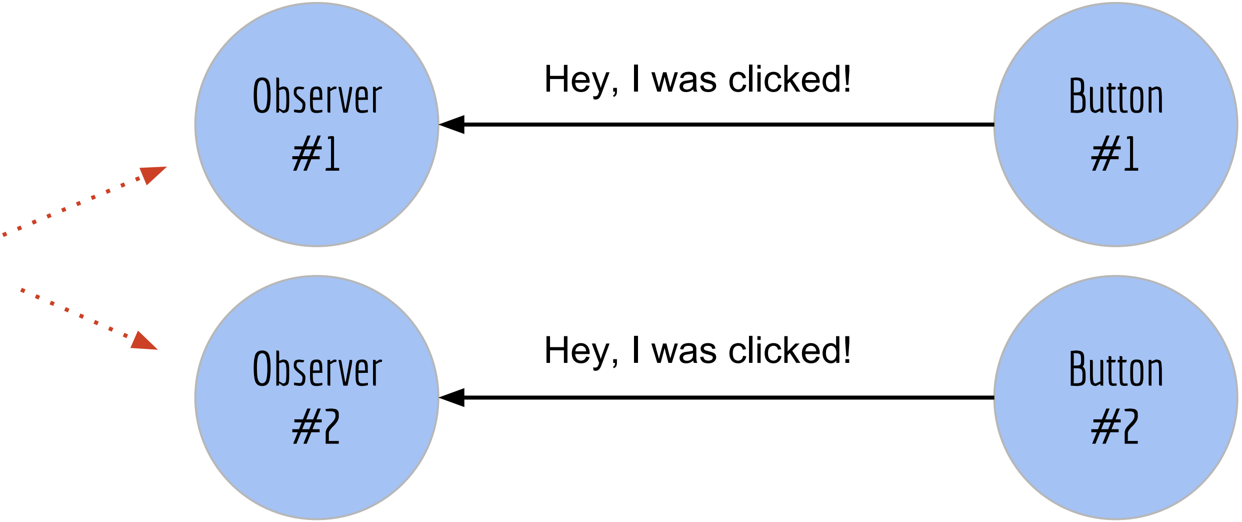
- An “event” is something that “happens” in a GUI application.
    - A button is clicked
    - The mouse is dragged
    - A menu item is selected

- GUI programs are entirely driven by events using the observer pattern.
    - Notify me when a button is clicked
    - Notify me when the mouse is dragged
    - Notify me when this menu item is selected

- The observers respond to events to achieve the program’s goals.

Handling a button click

- Define an observer for each button.
- When a button is clicked, that button notifies your observer.



Program these objects to do something useful!

i.e. what do you want to happen when the button is clicked?

Registering an observer

- Package:

```
import javafx.event.*;
```

- Observer interface:

```
public interface EventHandler< X> {
    void handle(X event);
}
```

- `X` is the event type. e.g.:
    - `ActionEvent` - when a button is clicked or a menu item is selected
    - `KeyEvent` - when a key is pressed, released or typed

- Registering an observer:

```
loginBtn.setOnAction(observer);
usernameTf.setOnKeyTyped(observer);
```

For more, see: https://docs.oracle.com/javafx/2/events/convenience_methods.htm

Registering an observer as an inner class

```
public class MyApplication extends Application {
  private TextField usernameTf;
  private PasswordField passwordTf;
  @Override public void start(Stage stage) {
    Button loginBtn = new Button(“Login”);
    loginBtn.setOnAction(new LoginButtonHandler());
    ...
  }
  private class LoginButtonHandler implements EventHandler<ActionEvent> {
    @Override public void handle(ActionEvent event) {
      if (checkPassword(usernameTf.getText(), passwordPf.getText())
        ...
    }
  }
}
```

Registering as an anonymous inner class

```
public class MyApplication extends Application {
  private TextField usernameTf;
  private PasswordField passwordTf;
  @Override public void start(Stage stage) {
    Button loginBtn = new Button(“Login”);
    loginBtn.setOnAction(new EventHandler<ActionEvent>() {
      @Override public void handle(ActionEvent event) {
        if (checkPassword(usernameTf.getText(), passwordPf.getText())
          ...
      }
    });
    ...
  }
}
```

Registering as a lambda expression

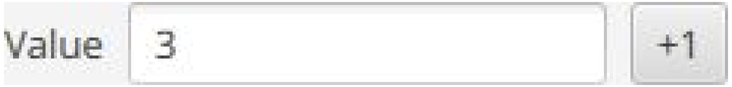
```
public class MyApplication extends Application {
  private TextField usernameTf;
  private PasswordField passwordTf;

  @Override public void start(Stage stage) {
    Button loginBtn = new Button(“Login”);
    loginBtn.setOnAction(event -> {
        if (checkPassword(usernameTf.getText(), passwordPf.getText())
          ...
    });
    ...
  }
}
```

Example

Specification

Build a GUI to add 1 to a value when you click a button.

The GUI looks like this:

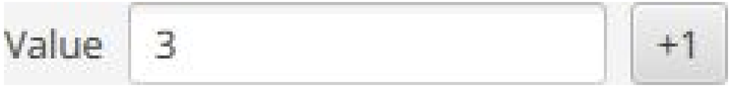


The pieces:

- `Label`
- `TextField`
- `Button`
- `HBox`
- `EventHandler<X>`
- `ActionEvent`
- `Scene`
- `Stage`

The layout

```
public class IncrementorApplication extends Application {
    public static void main(String[] args) { launch(args); }
    private Label valueLbl;
    private TextField valueTf;
    private Button incrementBtn;
    @Override
    public void start(Stage stage) {
        valueLbl = new Label(“Value”);
        valueTf = new TextField();
        incrementBtn = new Button(“+1”);
        HBox hBox = new HBox(10, valueLbl, valueTf, incBtn);
        stage.setScene(new Scene(hBox));
        stage.setTitle(“Incrementor”);
        stage.show();
    }
}
```



TextField getter/setter pattern

- A `TextField` has a getter that converts from a String.
  - Use `Integer.parseInt(s)` to convert the `String s` to an `int`.
  - Use `Double.parseDouble(s)` to convert the `String s` to a `double`.
- A `TextField` has a setter that converts to a String.

```
public class IncrementorApplication extends Application {
    private TextField valueTf;
    private int getValue() {
        return Integer.parseInt(valueTf.getText() );
    }
    private void setValue(int value) {
        valueTf.setText(“” + value);
    }
}
```

Set the event handler (observer)

```
public class IncrementorApplication extends Application {
    private TextField valueTf;
    private int getValue() { return Integer.parseInt(valueTf.getText()); }
    private void setValue(int value) { valueTf.setText(“” + value); }

    @Override
    public void start(Stage stage) {
        ...
        incrementBtn = new Button(“+1”);
        incrementBtn.setOnAction(event -> setValue(getValue() + 1));
    }
}
```

- The event handler can access getValue/setValue from the outer class.