GUIs continued

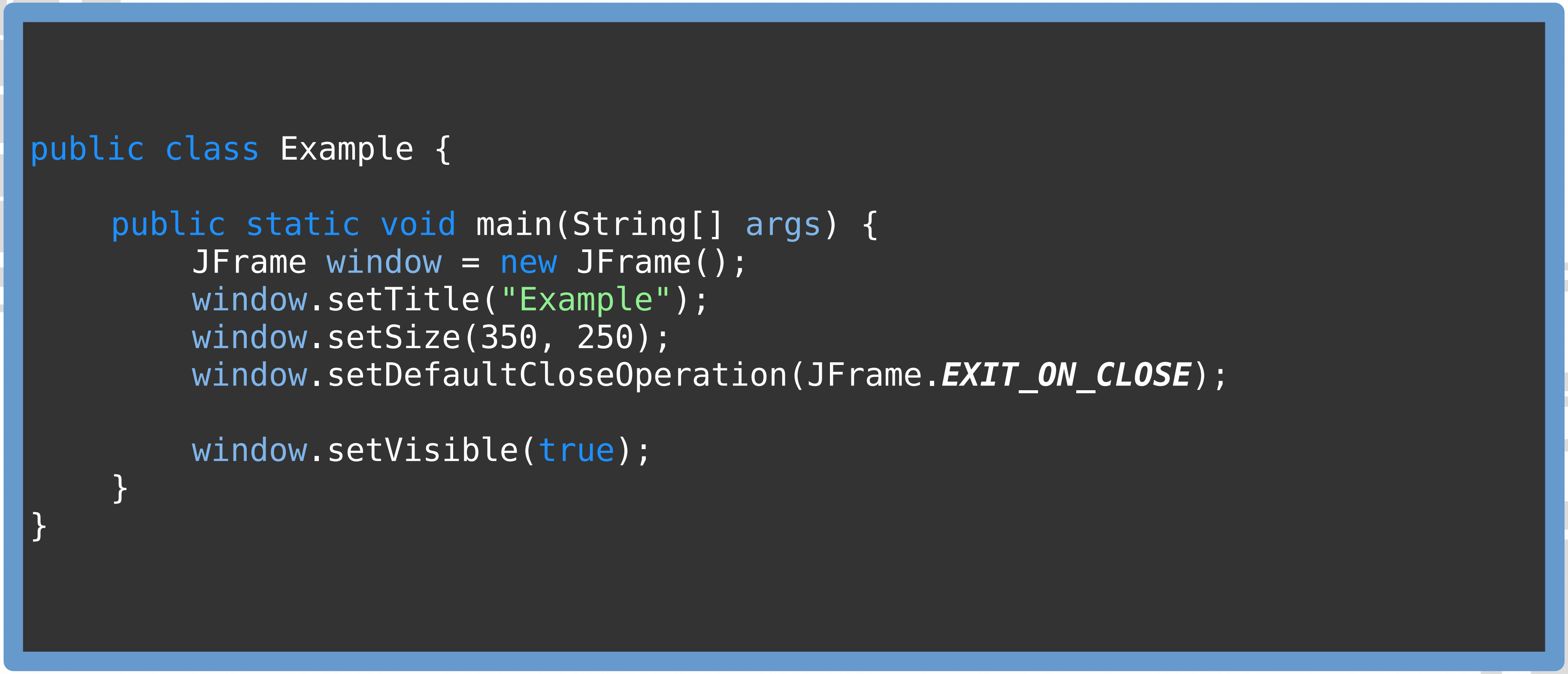
- Different ways of implementing action listeners and windows
- File Choosers
- Colour Choosers
- Menus
- Colours and borders
- Look and feel

Different ways of creating GUI applications

- There are several different ways of starting a GUI application in Java
  - Create a JFrame in the Main Method
  - Create a class that extends JFrame
  - Create a class that creates a JFrame in its constructor
- Each of these methods has specific advantages and disadvantages

Creating a JFrame in the main() method

- This is the method we've been using so far

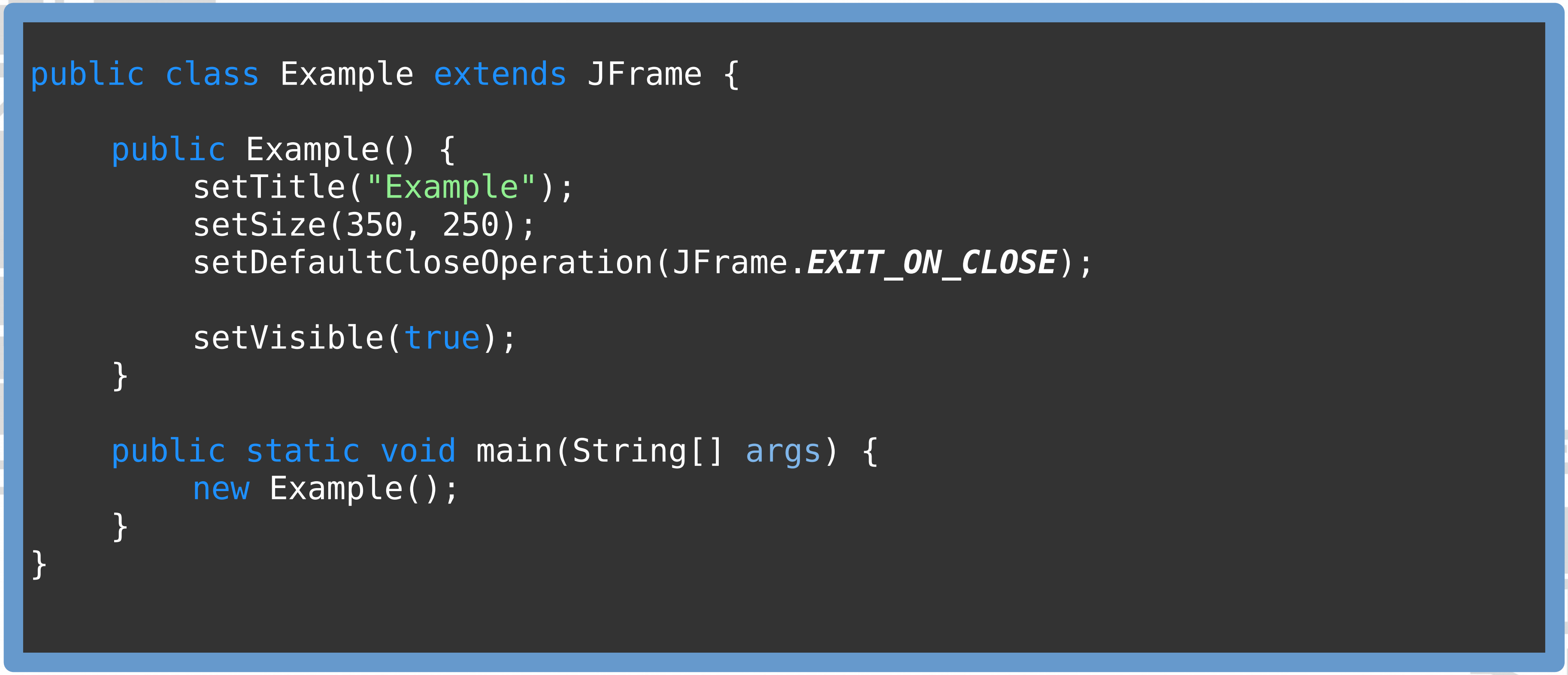


```java
public class Example {

    public static void main(String[] args) {
        JFrame window = new JFrame();
        window.setTitle("Example");
        window.setSize(350, 250);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        window.setVisible(true);
    }
}
```

Creating a JFrame in the main() method

- This is the simplest way of creating a GUI however, it has several downsides
    - Because it's in the main() method and the main method is static:
        - You cannot open the window more than once during the application
        - You cannot open two copies of the window with different data without running the application twice
        - Because it's static, you cannot access instance variables in event listeners
        - Because it's static, you cannot make the class its own action listener

Extending JFrame



```java
public class Example extends JFrame {

    public Example() {
        setTitle("Example");
        setSize(350, 250);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setVisible(true);
    }

    public static void main(String[] args) {
        new Example();
    }
}
```

Extending JFrame

- This is generally considered bad practice:
  - You have to be careful not to override any methods e.g. getX() and getY() and risk changing the behaviour of the window unexpectedly
  - Inheritance causes problems in general: Diamond Problem
  - You cannot easily move the components from a frame to a panel or elsewhere because you are defining a class
  - This is like extending an ArrayList just to add elements to it
  - Your application *has-a* window, your application is not a window

Creating a separate class for each window

```java
public class Example extends JFrame {
    public static void main(String[] args) {
        new MyWindow();
    }
}

public class MyWindow {
    public MyWindow() {
        JFrame window = new JFrame();
        window.setTitle("Example");
        window.setSize(350, 250);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        window.setVisible(true);
    }
}
```

Creating a separate class for each window

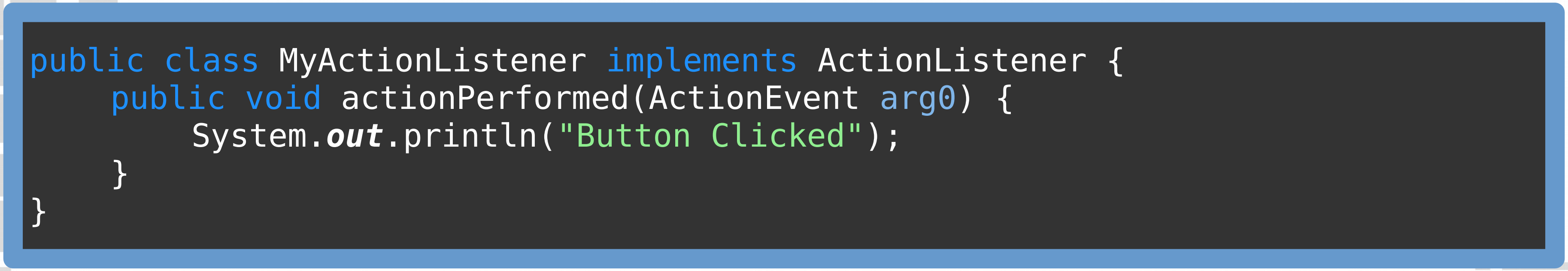
- The obvious disadvantage is that this requires more code
- The advantage of this is that it's more flexible. If you have multiple windows in your program you can easily change which one opens first
- This also avoids problems with creating objects inside static methods

Action Listeners

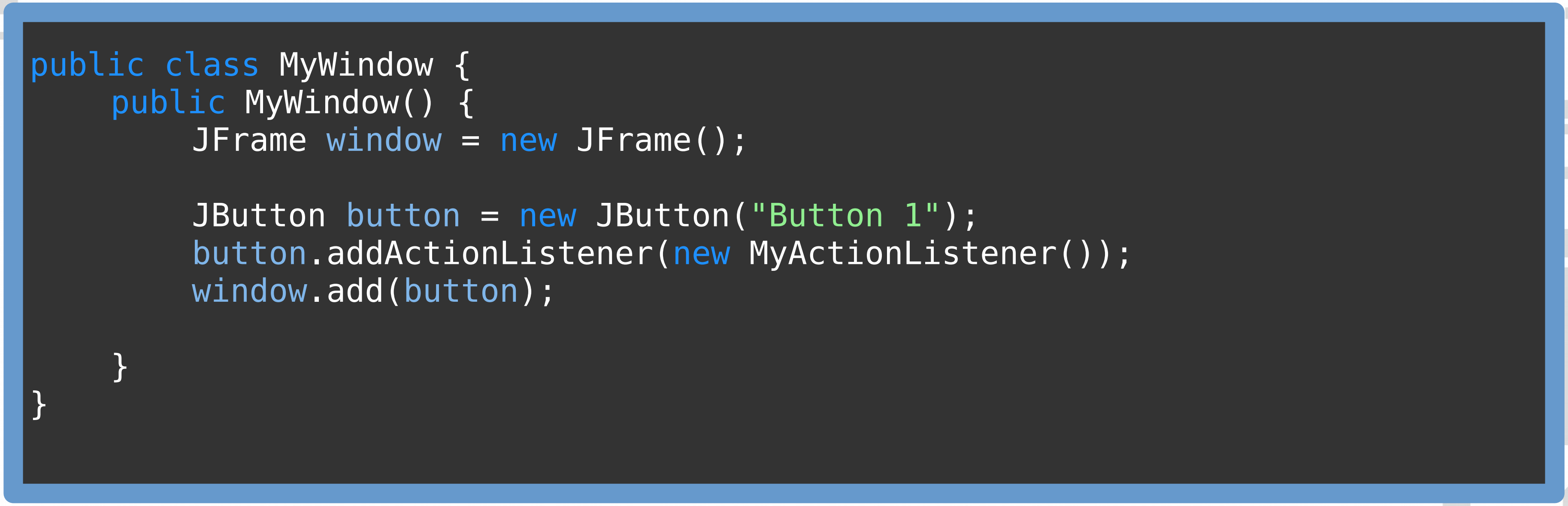
- There are also several ways of creating Action Listeners:
  - Concrete Classes
  - Inner Classes
  - Anonymous Inner Classes

Concrete Classes

- This involves putting the action listener in its own file/class



```
public class MyActionListener implements ActionListener {
    public void actionPerformed(ActionEvent arg0) {
        System.out.println("Button Clicked");
    }
}
```



```
public class MyWindow {
    public MyWindow() {
        JFrame window = new JFrame();

        JButton button = new JButton("Button 1");
        button.addActionListener(new MyActionListener());
        window.add(button);

    }
}
```

Concrete Classes

- The disadvantage of this approach is that it requires more code and editing code in two different files
- It also, like inner classes, requires passing arguments for required GUI components to the constructor:

Concrete Classes

```java
public class MyWindow {
    public MyWindow() {
        JFrame window = new JFrame();

        JButton button = new JButton("Button 1");
        button.addActionListener(new MyActionListener(button));
        window.add(button);

    }
}
```

```java
public class MyActionListener implements ActionListener {
    private JButton button;

    public MyActionListener(JButton button) {
        this.button = button;
    }

    public void actionPerformed(ActionEvent arg0) {
        this.button.setText("Button clicked");
    }
}
```

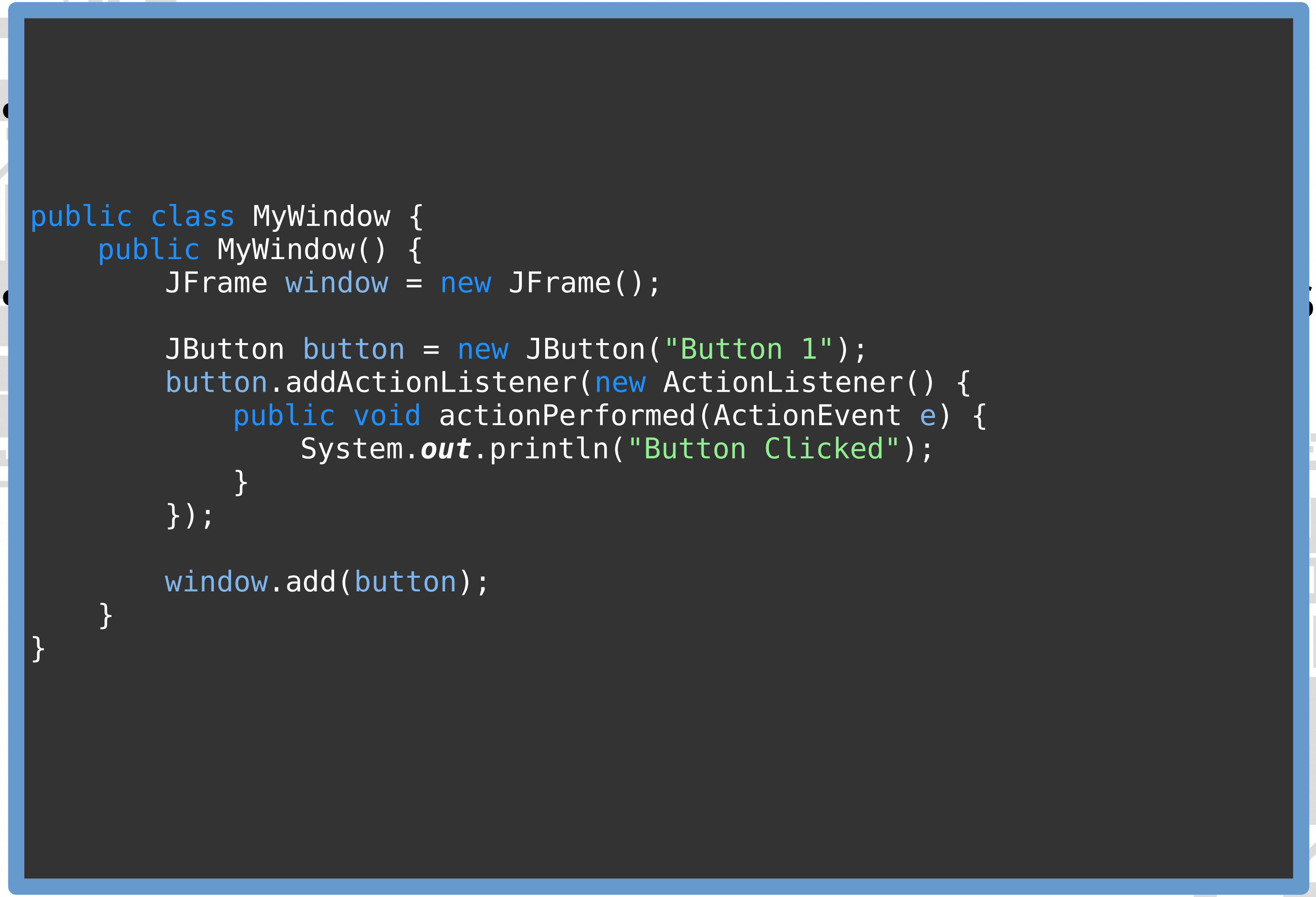
Inner Classes

- This is the method we have used in the last few weeks.
- It is the same as a Concrete class only the two classes are stored in the same file
- The disadvantages are the same, you still need to pass GUI components directly to the action listener

Inner Classes

```java
public class MyWindow {
    public class MyActionListener implements ActionListener {
        private JButton button;

        public MyActionListener(JButton button) {
            this.button = button;
        }

        public void actionPerformed(ActionEvent arg0) {
            this.button.setText("Button clicked");
        }
    }

    public MyWindow() {
        JFrame window = new JFrame();

        JButton button = new JButton("Button 1");
        button.addActionListener(new MyActionListener(button));
        window.add(button);

    }
}
```

Anonymous Inner Classes



```java
public class MyWindow {
    public MyWindow() {
        JFrame window = new JFrame();

        JButton button = new JButton("Button 1");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button Clicked");
            }
        });

        window.add(button);
    }
}
```

Anonymous Inner Classes

- These are less reusable than named classes as they can only be used where they are defined
- The are also arguably more difficult to read
- By mixing all the GUI building code with the action listeners you end up with very long methods that can become difficult to work with in larger applications

Anonymous Inner Classes

- Anonymous Inner classes cannot have constructors. This means you cannot pass in GUI components in the same way as inner classes.

Anonymous Inner Classes

- This is not possible:

```
public class MyWindow {
    public MyWindow() {
        JFrame window = new JFrame();

        JButton button = new JButton("Button 1");

        button.addActionListener(new ActionListener(button) {
            private JButton button;

            public ActionListener(JButton button) {
                this.button = button;
            }
            public void actionPerformed(ActionEvent e) {
                button.setText("Button clicked");
            }
        });

        window.add(button);
    }
}
```

Anonymous Inner Classes

- However, Anonymous Inner Classes can access *instance variables* for the class they are created in

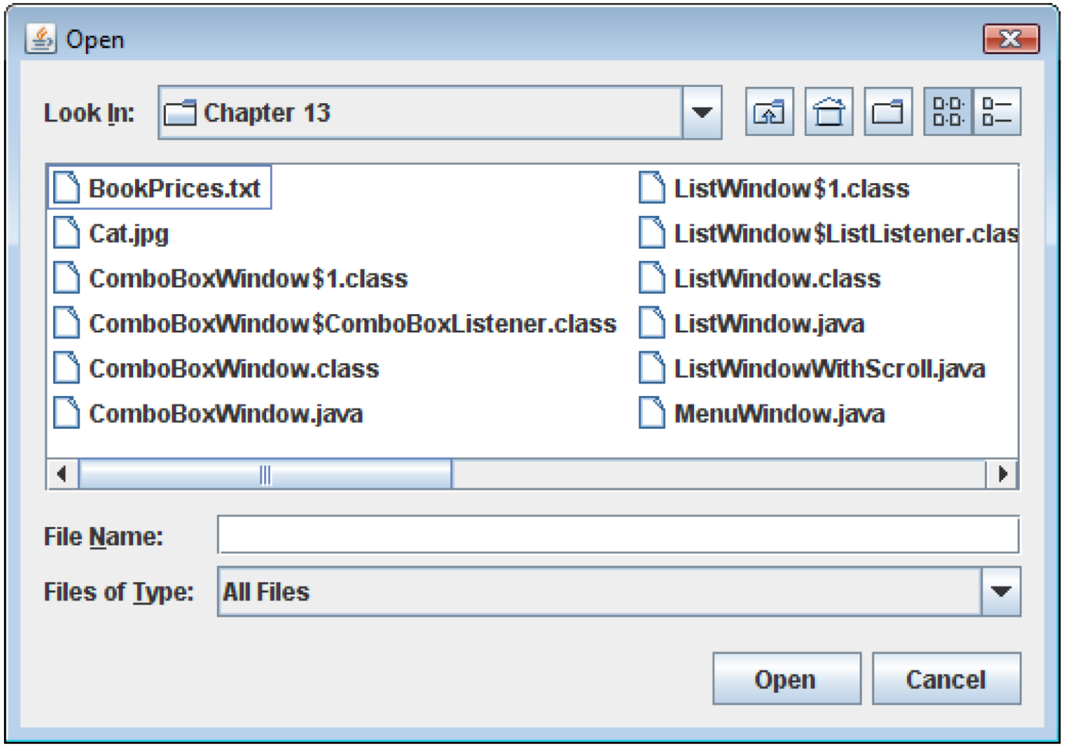
```
public class MyWindow {
    private JButton button;

    public MyWindow() {
        JFrame window = new JFrame();

        button = new JButton("Button 1");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                button.setText("Button clicked");
            }
        });

        window.add(button);
    }
}
```

Which to use?

- It's up to you which you prefer as each method has its advantages and disadvantages
- For the assignment you can use any method you like and it will not affect your grade

File Choosers

- A file chooser is a specialized dialog box that allows the user to browse for a file and select it.



Look In: Chapter 13

BookPrices.txt
Cat.jpg
ComboBoxWindow$1.class
ComboBoxWindow$ComboBoxListener.class
ComboBoxWindow.class
ComboBoxWindow.java
ListWindow$1.class
ListWindow$ListListener.clas
ListWindow.class
ListWindow.java
ListWindowWithScroll.java
MenuWindow.java

File Name:

Files of Type: All Files

File Choosers

- Create an instance of the ***JFileChooser*** class to display a file chooser dialog box.
- Two of the constructors have the form:

  `JfileChooser()` When no constructor arguments are passed to the file chooser, it will open at the default location. On Windows this is the *My Documents* folder. On Linux this is the home folder for the current user (~/)

- Alternatively you can provide a path that is opened

  `JFileChooser(String path)` The argument is the path that will be opened to start with

- If the constructor argument is not a valid path it will default to the default path as above.

File Choosers

- There are two types of File Chooser:
  - Open File Dialog box – lets the user browser for an existing file to open
  - Save File Dialog box – lets the user browse to a location to save
- The difference between them is that an open dialog forces the selection of an existing file
- When saving, a new file location can be specified

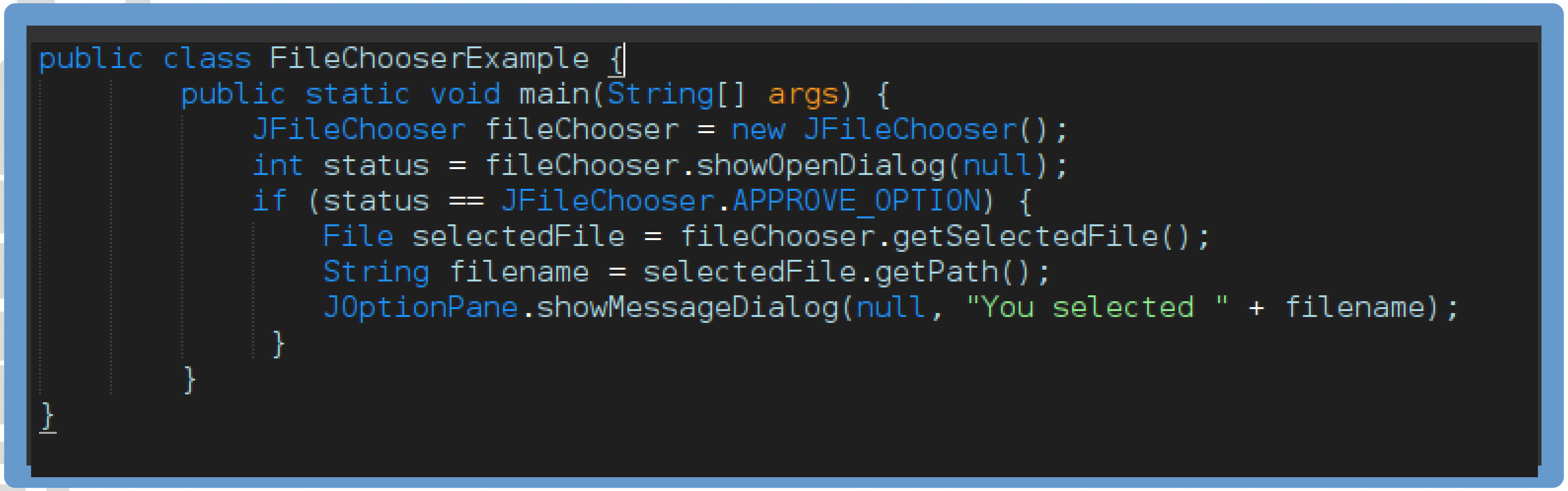
File Choosers

- To display a save file dialog use the method ***showSaveDialog(Component parent)*** method
- The argument is a component or reference to null. This is the parent window for the file chooser
- The method returns an integer that represents the action taken by the user.
  - If a file is selected the value represented by the constant ***JFileChooser.APPROVE_OPTION*** is returned from the method

File Choosers

- When the file launcher is opened, the rest of the program execution is paused until the user takes an action (Selecting a file or closing/cancelling the dialog)
- No more lines of code run until the user makes a choice.

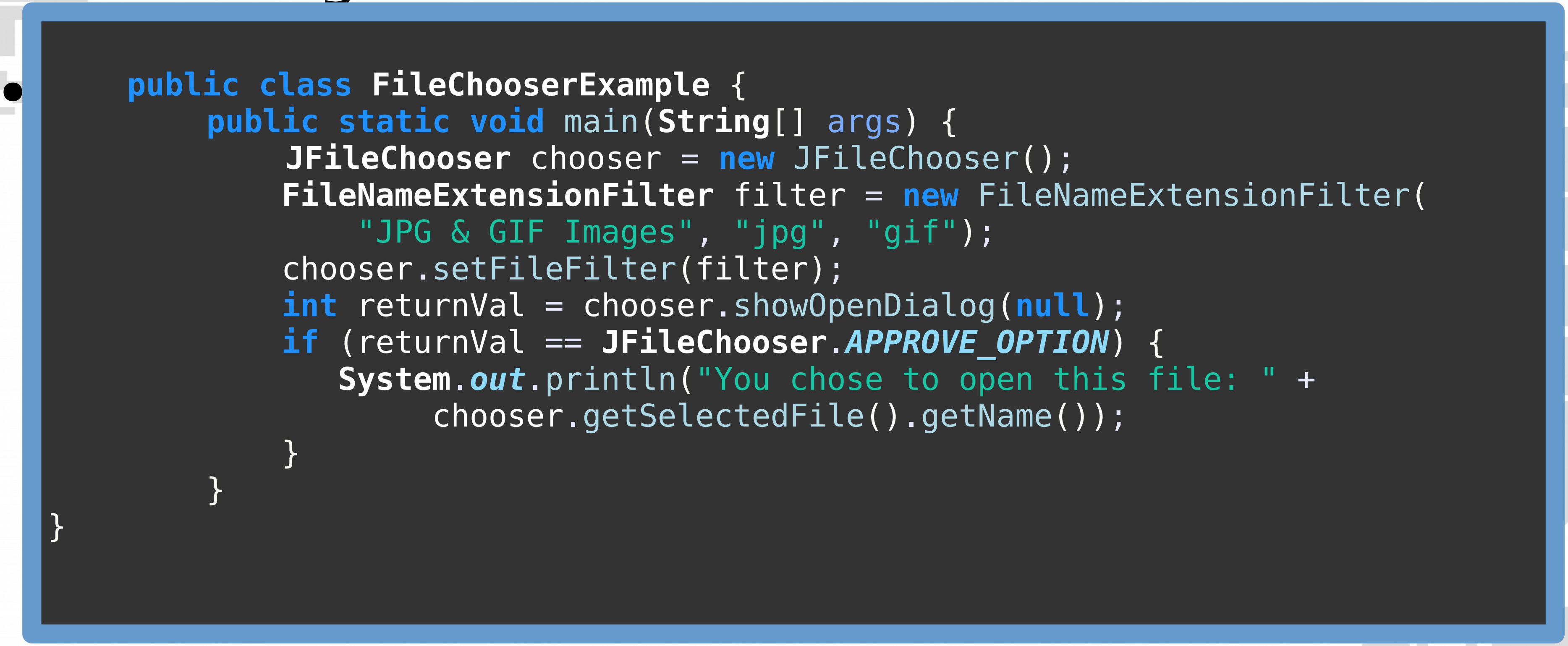
File Choosers



```java
public class FileChooserExample {
    public static void main(String[] args) {
        JFileChooser fileChooser = new JFileChooser();
        int status = fileChooser.showOpenDialog(null);
        if (status == JFileChooser.APPROVE_OPTION) {
            File selectedFile = fileChooser.getSelectedFile();
            String filename = selectedFile.getPath();
            JOptionPane.showMessageDialog(null, "You selected " + filename);
        }
    }
}
```

File Choosers

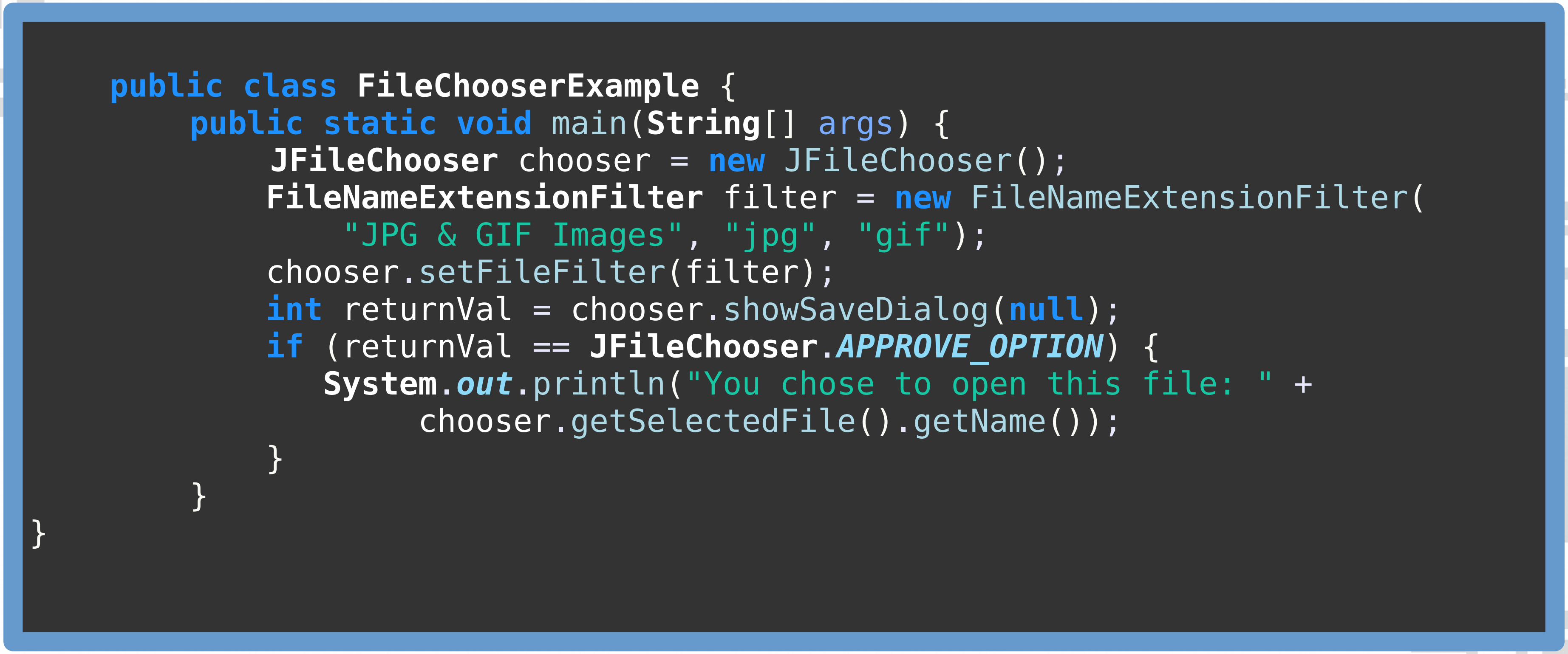
- To select a specific file type you can use the ***FileFilter*** class. E.g. to only allow JPEG and GIF images you can create a FileChooser using the following code:



```
    public class FileChooserExample {
        public static void main(String[] args) {
            JFileChooser chooser = new JFileChooser();
            FileNameExtensionFilter filter = new FileNameExtensionFilter(
                "JPG & GIF Images", "jpg", "gif");
            chooser.setFileFilter(filter);
            int returnVal = chooser.showOpenDialog(null);
            if (returnVal == JFileChooser.APPROVE_OPTION) {
               System.out.println("You chose to open this file: " +
                    chooser.getSelectedFile().getName());
            }
        }
}
```

Save Dialogs

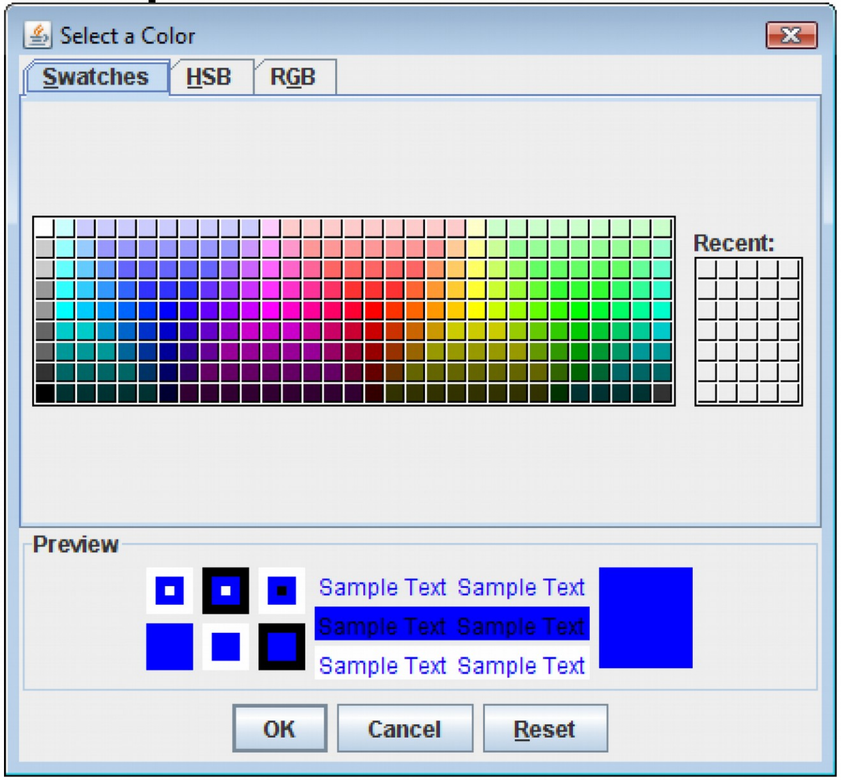
- Save dialogs are identical to open dialogs
- The only difference is they use the *showSaveDialog()* method



```
    public class FileChooserExample {
        public static void main(String[] args) {
            JFileChooser chooser = new JFileChooser();
            FileNameExtensionFilter filter = new FileNameExtensionFilter(
                "JPG & GIF Images", "jpg", "gif");
            chooser.setFileFilter(filter);
            int returnVal = chooser.showSaveDialog(null);
            if (returnVal == JFileChooser.APPROVE_OPTION) {
               System.out.println("You chose to open this file: " +
                    chooser.getSelectedFile().getName());
            }
        }
}
```

Colour Choosers

- A colour chooser is a specialised dialog box that allows the user to select a colour from a predefined palette of colours

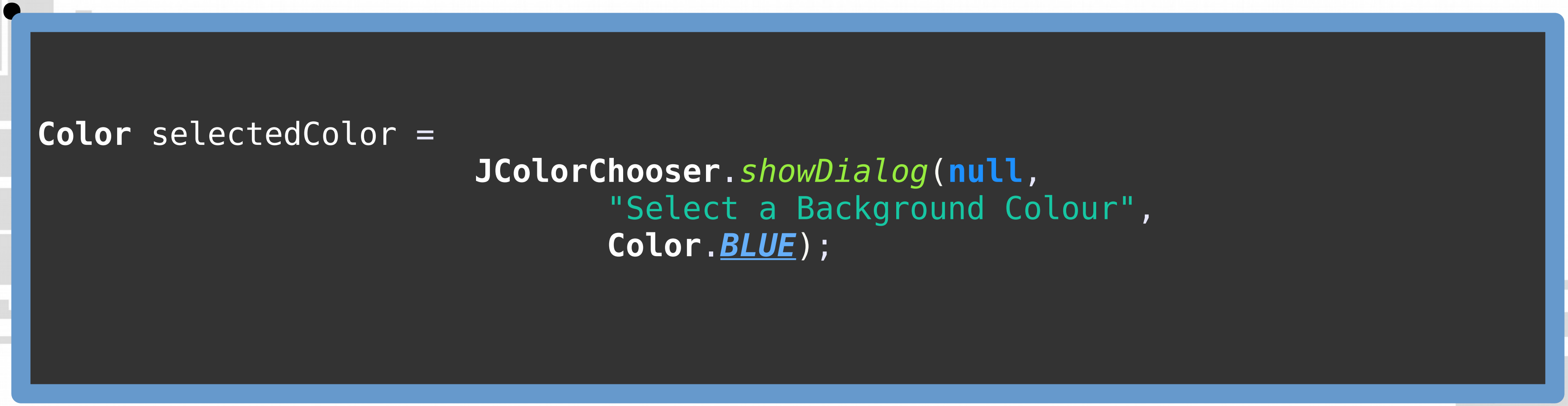


Colour Choosers

- The colour can be input in either HSB or RGB mode
- The colour is returned to your programming using the inbuilt java ***Color*** class
- *Note: Java uses the American spelling of 'color' throughout*
- The ***Color*** class is used throughout Swing, it's used for font colours, border colours and background colours.

Colour Choosers

- A colour chooser is created using the code:
-

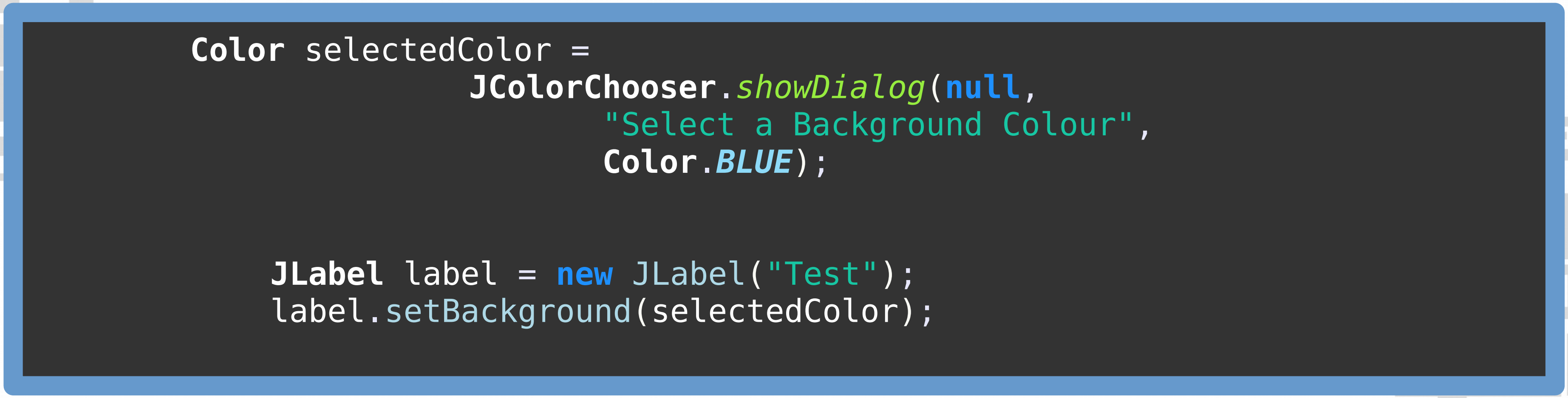


```
Color selectedColor =
                       JColorChooser.showDialog(null,
                              "Select a Background Colour",
                              Color.BLUE);
```

- The three parameters are, parent window, colour chooser window title and the default selected colour

Colour Chooses

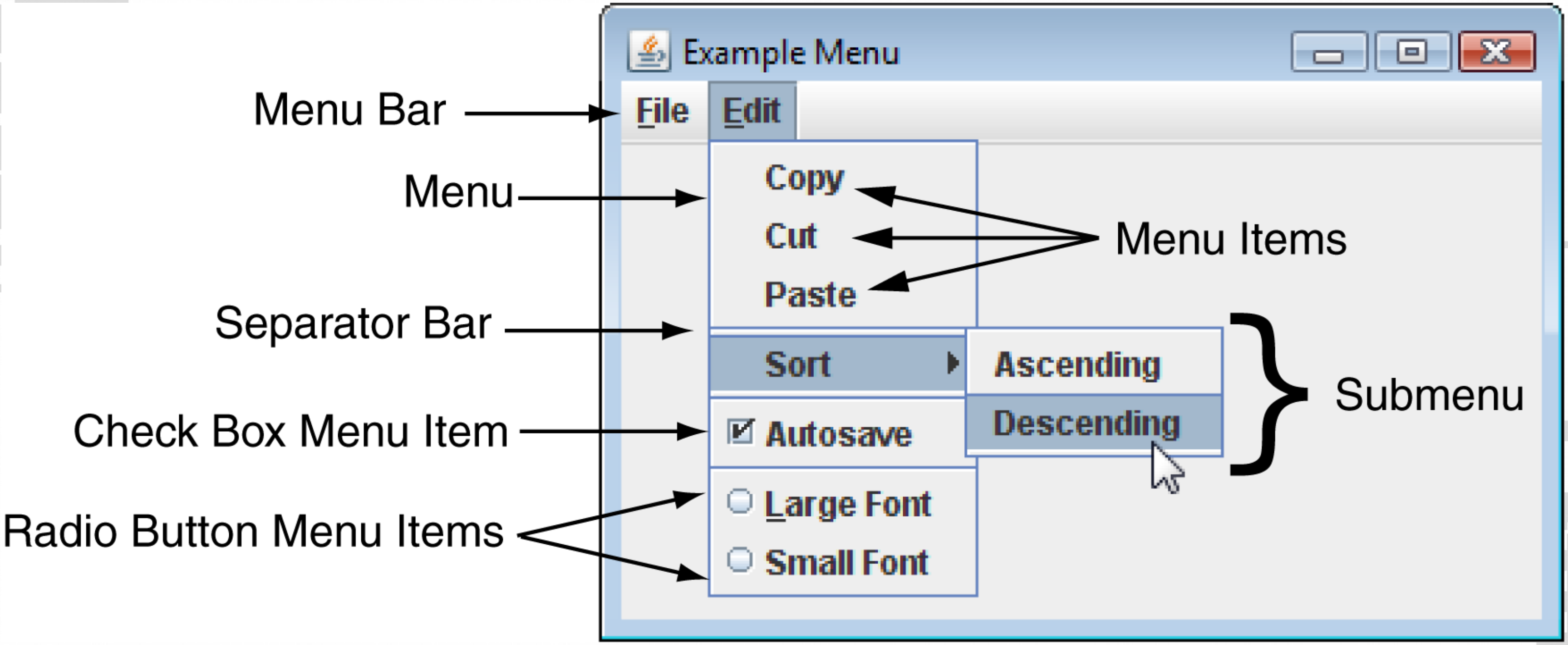
- Once you have a ***<u>Color</u>*** object you can use it in various methods on swing components.



```
Color selectedColor =
               JColorChooser.showDialog(null,
                      "Select a Background Colour",
                      Color.BLUE);


    JLabel label = new JLabel("Test");
    label.setBackground(selectedColor);
```

Menus

- A *menu system* is a collection of commands organised in one or more drop-down menus.



Menus

- Drop down menus can contain:
  - **Item** – a clickable entry which performs an action when selected
  - **Other menus** (Submenus) indicated with a > to the right hand side of the menu, expands the submenu when selected
  - **Checkboxes** – indicated as a checkbox to the left of the menu item
  - **Radio buttons** – indicated as a radio button to the right of the menu item
  - **Separator bar** – a graphical line between two menu entries

Menu Classes

- A menu system is constructed with the following classes:
- ***JMenuBar*** – *Used to create a menu bar.*
- A ***JMenuBar*** object can contain ***JMenu*** components.
  - JMenu – Used to create a menu. A JMenu component can contain:
- ***JMenuItem***, ***JCheckBoxMenuItem***, and ***JRadioButtonMenuItem*** components,
- as well as other ***JMenu*** components.
- A submenu is a ***JMenu*** component that is inside another ***JMenu*** component.
- ***JMenuItem*** – Used to create a regular menu item.
- A ***JMenuItem*** component generates an action event when selected.

Menu Classes

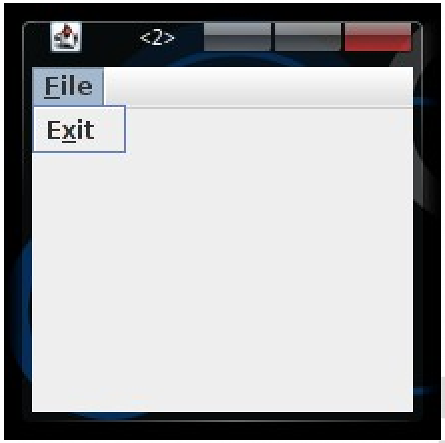
- ***JCheckBoxMenuItem* –** Used to create a check box menu item.
  - The class's ***isSelected*** method returns true if the item is selected, or false otherwise.
- A ***JCheckBoxMenuItem*** component generates an action event when selected.
- ***JRadioButtonMenuItem* –** Used to create a radio button menu item.
- ***JRadioButtonMenuItem*** components can be grouped together in a ***ButtonGroup*** object so that only one of them can be selected at a time.
- The class's ***isSelected*** method returns true if the item is selected, or false otherwise.
- A ***JRadioButtonMenuItem*** component generates an action event when selected.

Menus

- Menus are build as a hierarchy
- A JMenuBar is added to a JFrame
  - You may only have one menu per Jframe
- One or more JMenu components are added to the JMenuBar
- One or more JMenuItem components are added to each JMenu
- A JMenu can be added to another JMenu to act as a submenu

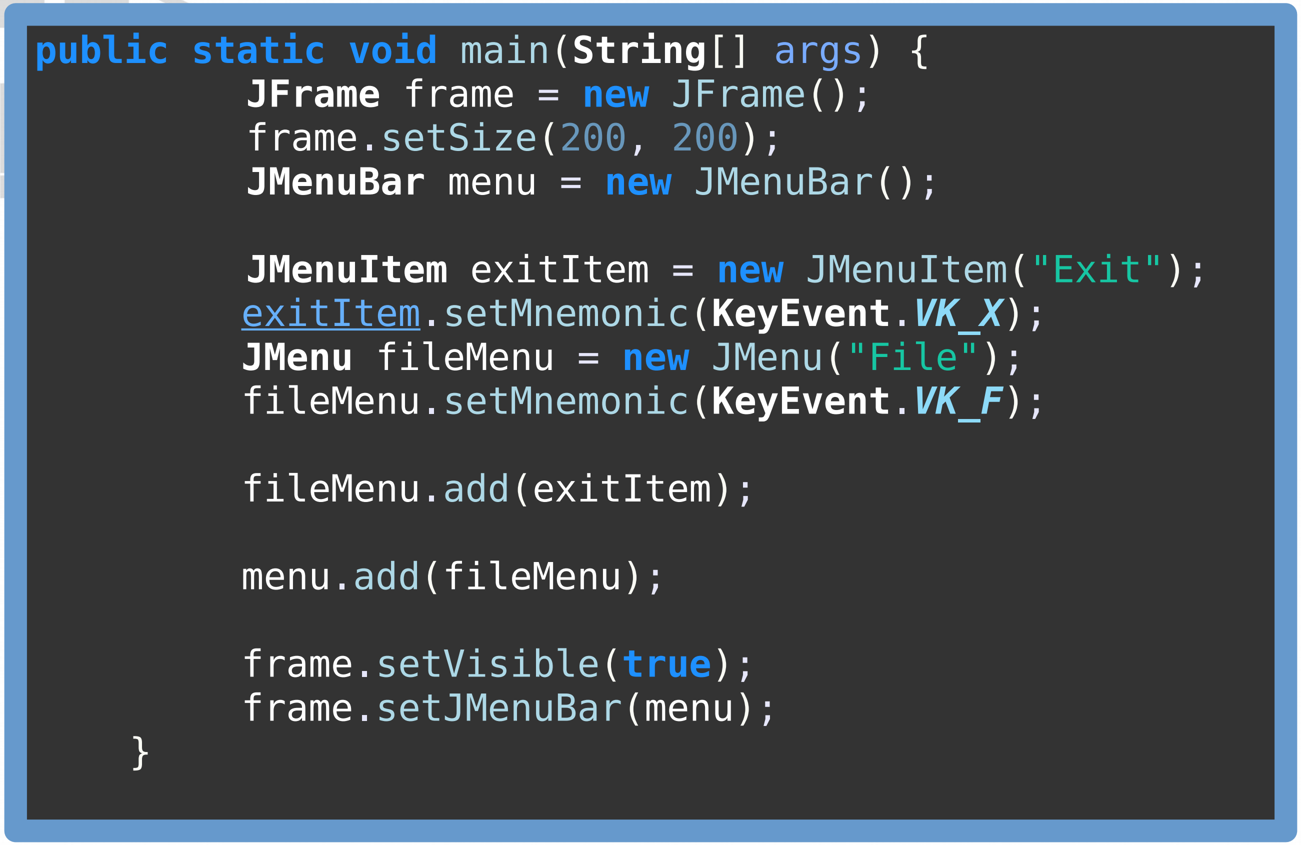
Menu Example

```
public static void main(String[] args) {
         JFrame frame = new JFrame();
         frame.setSize(200, 200);
         JMenuBar menu = new JMenuBar();

         JMenuItem exitItem = new JMenuItem("Exit");

         JMenu fileMenu = new JMenu("File");

         fileMenu.add(exitItem);

         menu.add(fileMenu);

         frame.setVisible(true);
         frame.setJMenuBar(menu);
    }
```

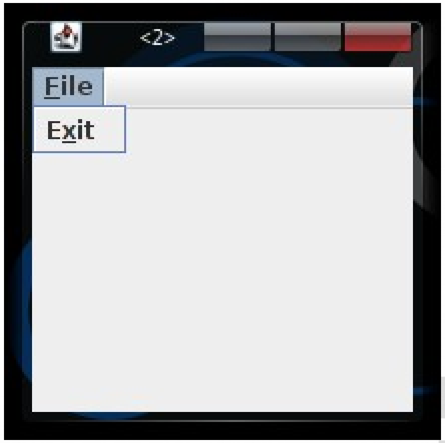


Menu Example

- You can set shortcut keys (mneumonics) on menus and individual items

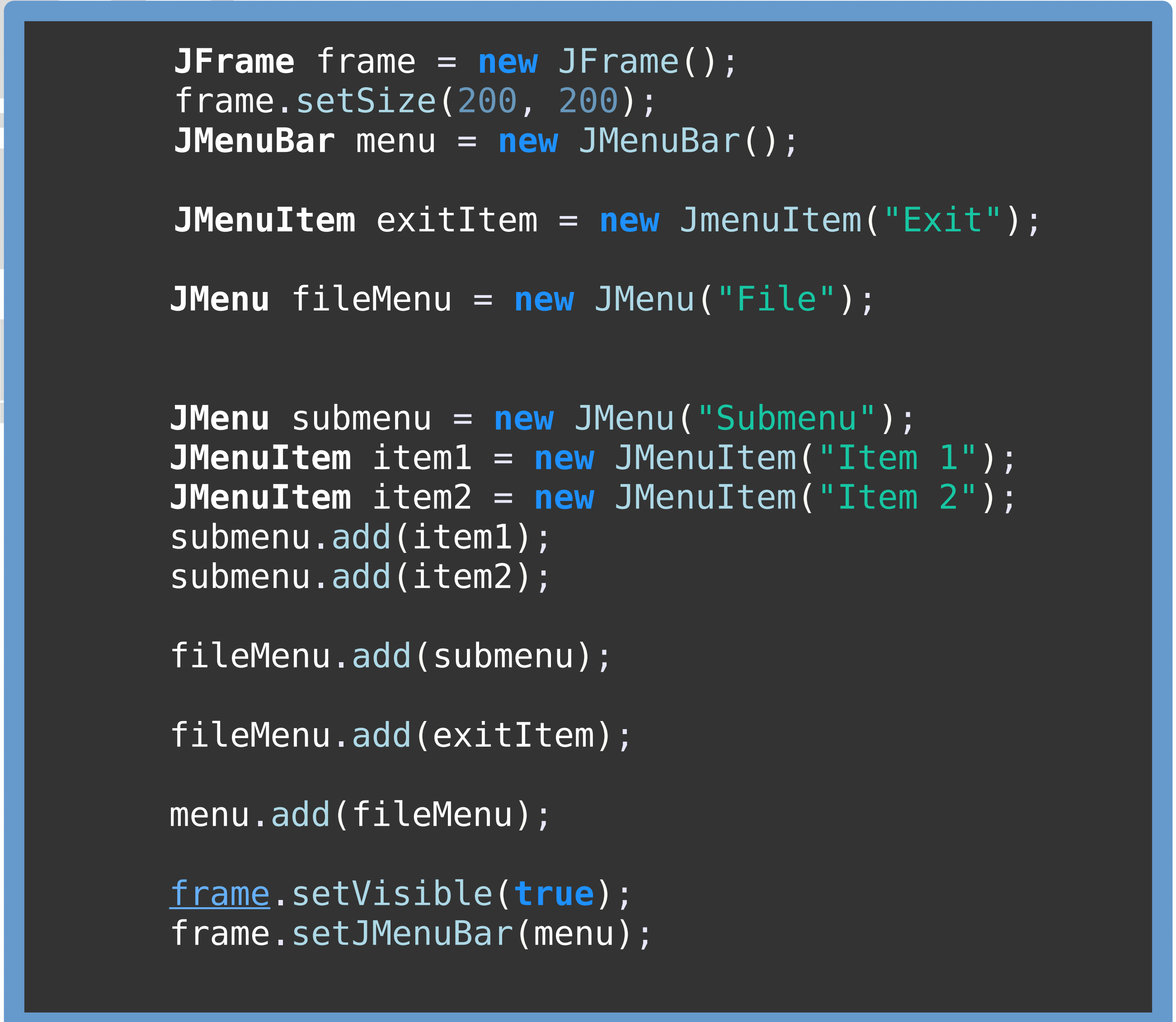


```
public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(200, 200);
        JMenuBar menu = new JMenuBar();

        JMenuItem exitItem = new JMenuItem("Exit");
        exitItem.setMnemonic(KeyEvent.VK_X);
        JMenu fileMenu = new JMenu("File");
        fileMenu.setMnemonic(KeyEvent.VK_F);

        fileMenu.add(exitItem);

        menu.add(fileMenu);

        frame.setVisible(true);
        frame.setJMenuBar(menu);
    }
```

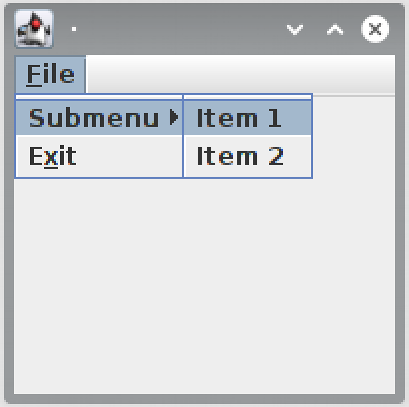


Submenus

- To add a menu add a menu to another menu

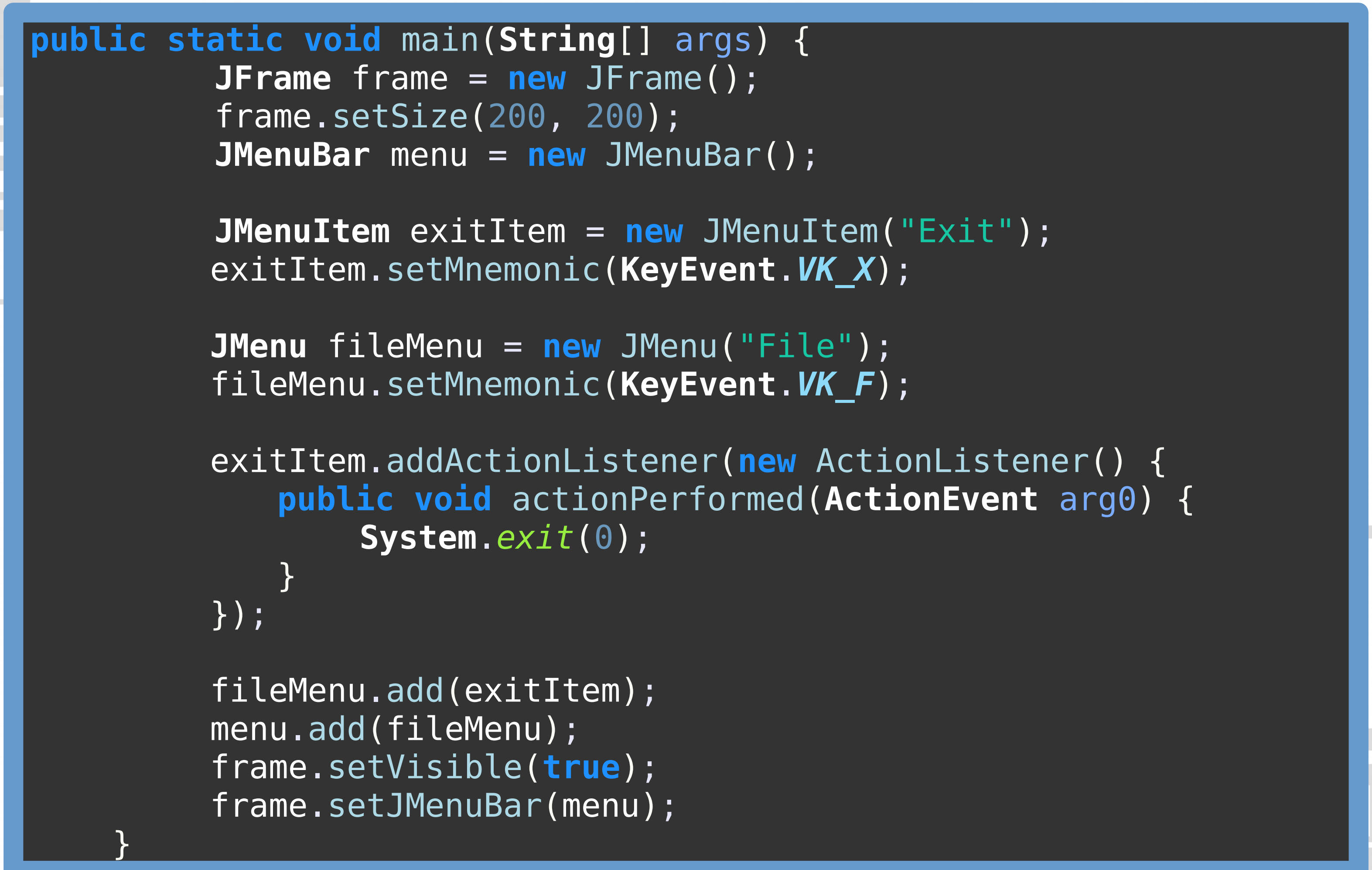


```
JFrame frame = new JFrame();
frame.setSize(200, 200);
JMenuBar menu = new JMenuBar();

JMenuItem exitItem = new JmenuItem("Exit");

JMenu fileMenu = new JMenu("File");


JMenu submenu = new JMenu("Submenu");
JMenuItem item1 = new JMenuItem("Item 1");
JMenuItem item2 = new JMenuItem("Item 2");
submenu.add(item1);
submenu.add(item2);

fileMenu.add(submenu);

fileMenu.add(exitItem);

menu.add(fileMenu);

frame.setVisible(true);
frame.setJMenuBar(menu);
```



Menu Action Listeners

- Menu items have action listeners which get triggered when the menu is selected
-



```
public static void main(String[] args) {
         JFrame frame = new JFrame();
         frame.setSize(200, 200);
         JMenuBar menu = new JMenuBar();

         JMenuItem exitItem = new JMenuItem("Exit");
         exitItem.setMnemonic(KeyEvent.VK_X);

         JMenu fileMenu = new JMenu("File");
         fileMenu.setMnemonic(KeyEvent.VK_F);

         exitItem.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent arg0) {
                 System.exit(0);
             }
         });

         fileMenu.add(exitItem);
         menu.add(fileMenu);
         frame.setVisible(true);
         frame.setJMenuBar(menu);
    }
```

Look and Feel

- By Default, Swing components do not look at all like the underlying operating system
- This can be good if you want it to look identical on every platform
- However, usually it's better to blend in with the operating system

Look and Feel

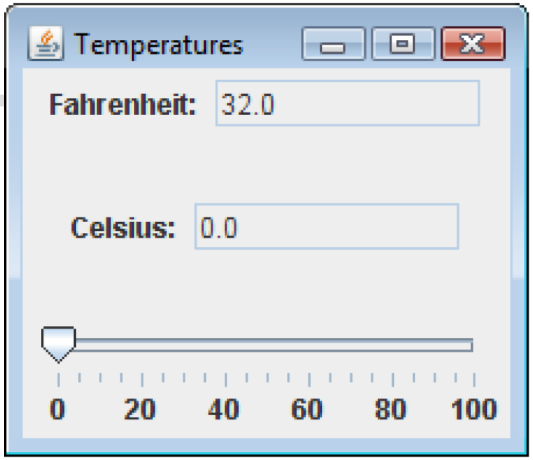
- In Swing, the various available visual styles are known as *look and feel*
- The default look and feel of swing is called *Metal*
- On modern operating systems in 2015 this looks very dates
-

Look and feel

- Java also contains some inbuilt look and feels which mimic various operating systems:
  - Windows is the look and feel of the Windows operating system
  - Motif is the look and feel of very old Unix based operating systems
  - GTK is the modern Linux look and feel

Look and Feel

Metal look and feel

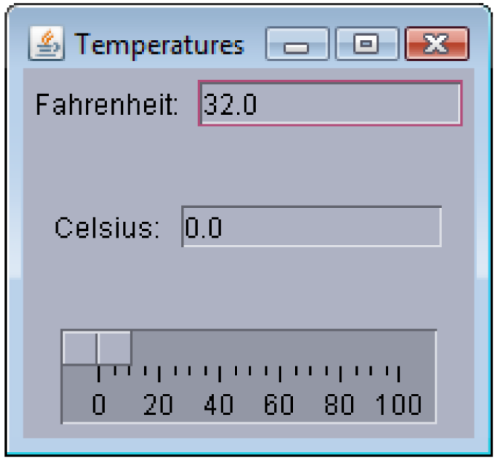


Temperatures

Fahrenheit: 32.0

Celsius: 0.0

Motif look and feel

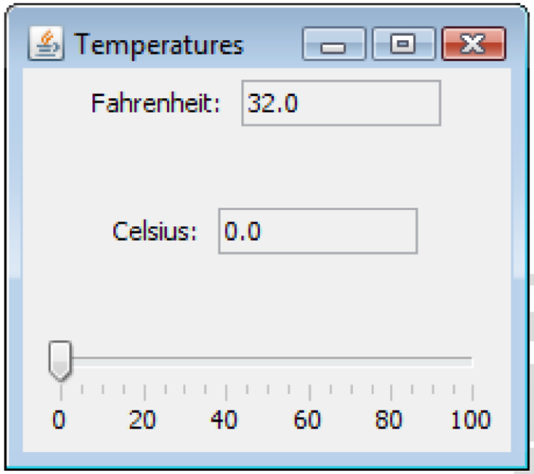


Temperatures

Fahrenheit: 32.0

Celsius: 0.0

Windows look and feel



Temperatures

Fahrenheit: 32.0

Celsius: 0.0

Look and Feel

- You can set the look and feel of the whole application using the code

```
UIManager.setLookAndFeel("name of look and feel");
```

Look and Feel

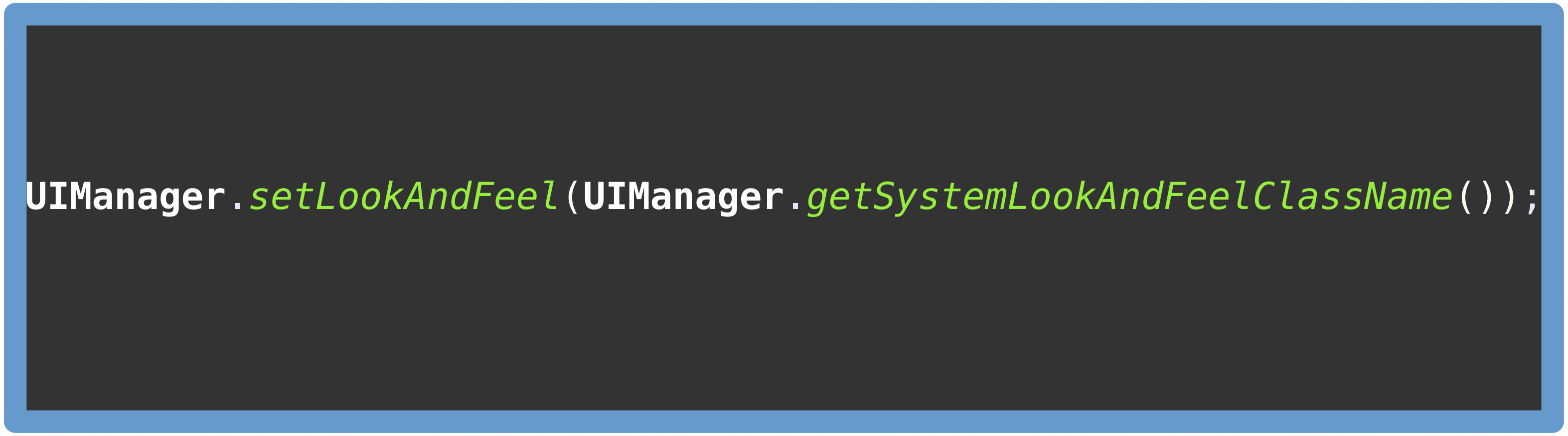
- The available look and feel names are:
  - Metal:

    "javax.swing.plaf.metal.MetalLookAndFeel"
  - Motif:

    "com.sun.java.swing.plaf.motif.MotifLookAndFeel"
  - Windows: "com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
  - GTK

    "com.sun.java.swing.plaf.gtk.GTKLookAndFeel

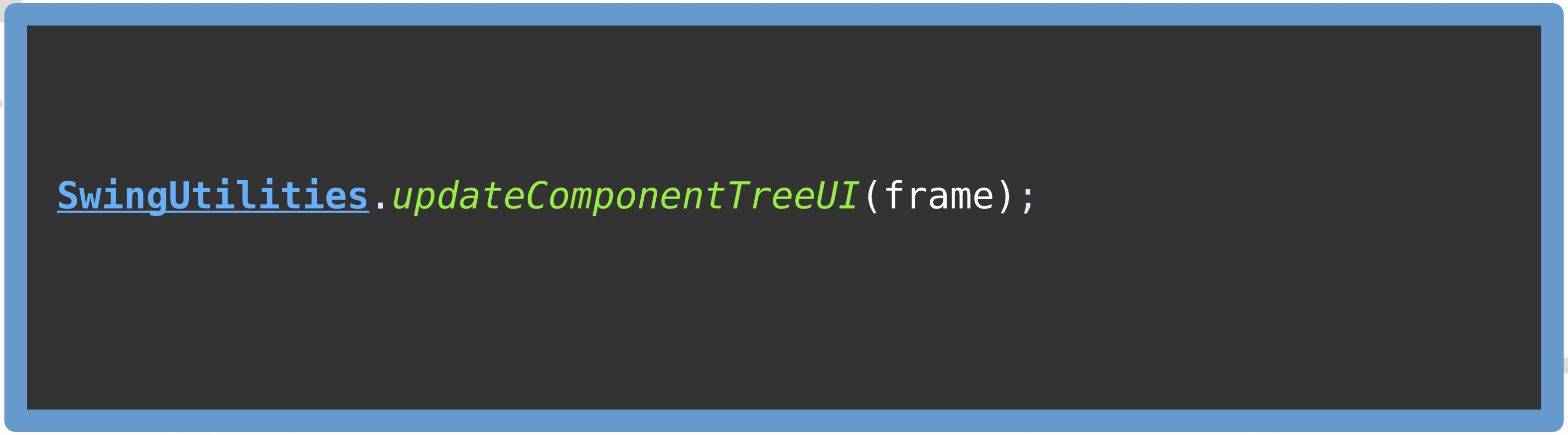
Look and Feel

- However, not all look and feels are available on all operating systems. Generally, you will want to use the look and feel based on the current operating system.
- Java contains a method which detects the operating system you are running the program on and finds the correct look and feel for you:
- 

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

Look and Feel

- If you set the look and feel after making the window visible, you will need to force an update to apply the look and feel. This is done using the line:
-



```
SwingUtilities.updateComponentTreeUI(frame);
```

Look And Feel

- Because the look and feel may not be applied successfully you must put the ***setLookAndFeel()*** method call in a try/catch block.
- The possible exceptions thrown by setLookAndFeel() are:
  - ***ClassNotFoundException***
  - ***InstantiationException***
  - ***IllegalAccessException***
  - ***UnsupportedLookAndFeelException***

Look and Feel

```
try {
      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
      SwingUtilities.updateComponentTreeUI(frame);
 }
 catch (Exception e) {
      System.out.println(e);
 }
```