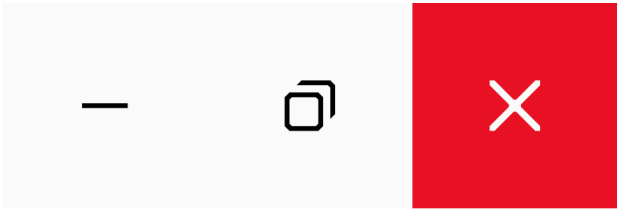


JavaFX

The GUI Stack

JavaFX

May 17

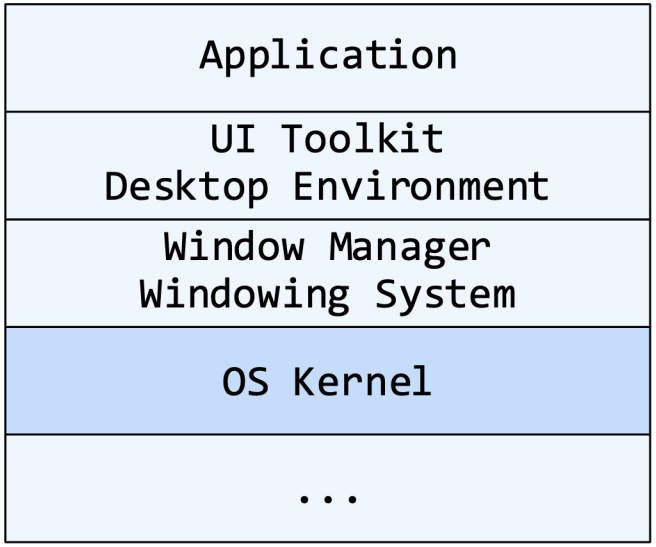
The GUI Stack

GUI Stack Components

OS Kernel: hardware access, device management, see CS350

Layers “above” the OS Kernel are responsible for handling

- Window management
- User-interaction (input/output)
- Executing applications

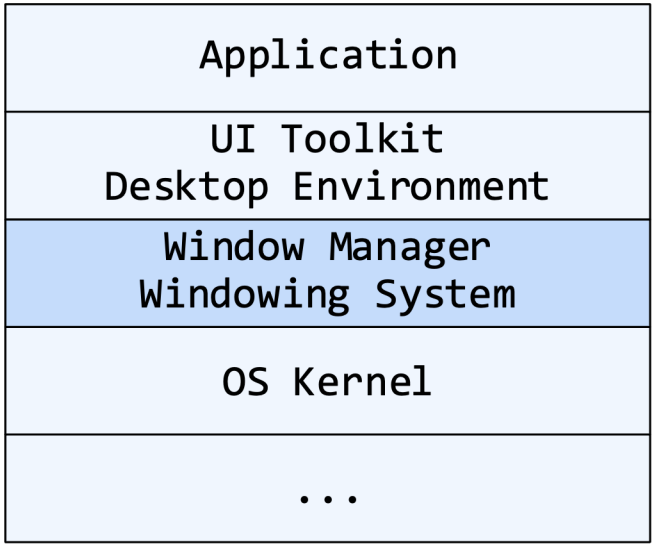


Window Manager (WM)

A WM provides the following functionality:

- Communication with the OS for creating, destroying, and managing application windows. This includes tiling windows, overlapping windows, etc.
- Routing of (user and system) input to the correct window. Typically, the window that “has focus” receives input.

A WM shields the application from the frame buffer and graphics drivers, its own location and visibility, and any other application window.



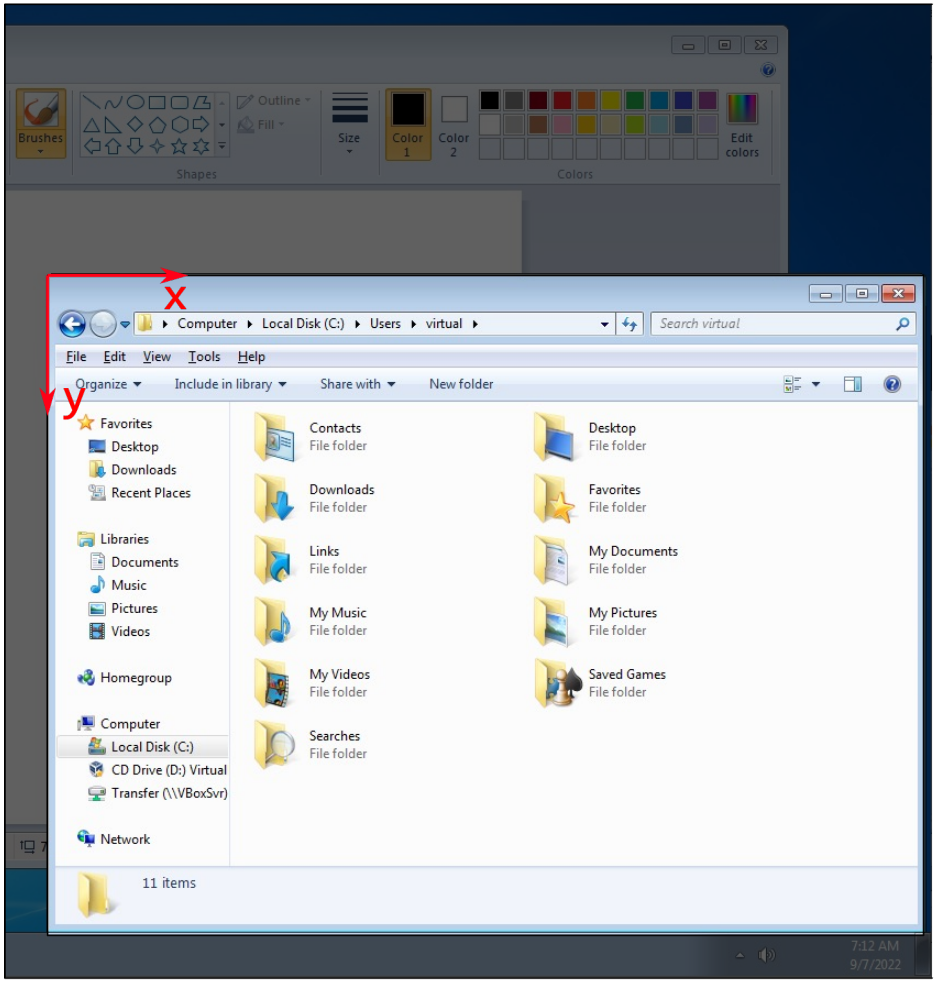
Window Manager – Canvas Abstraction

Each window contains a “canvas” or drawing area for the application

Each window is independent and has no knowledge of other windows.

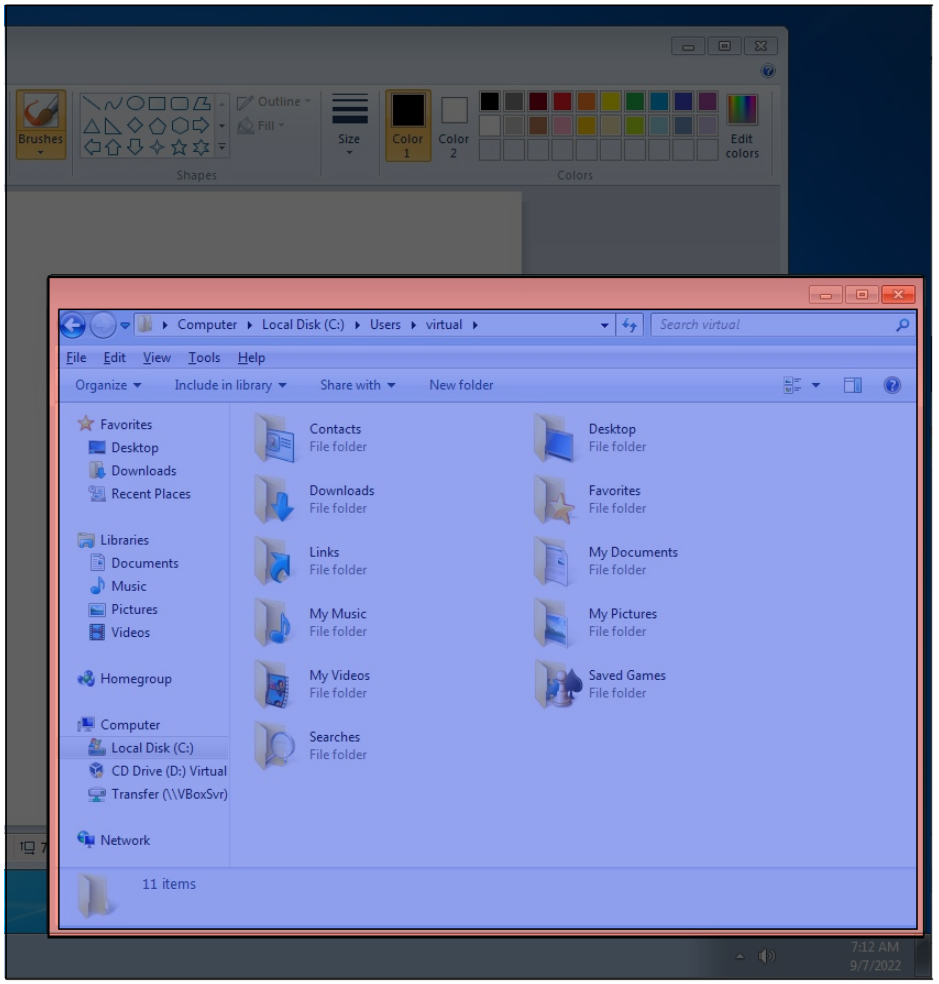
Each window has its own coordinate system:

- The WM transforms between global (screen) and local (window) coordinates
- An application does not worry where it is on screen; it assumes its top-left coordinate is `(0,0)`



Window Manager – Window components

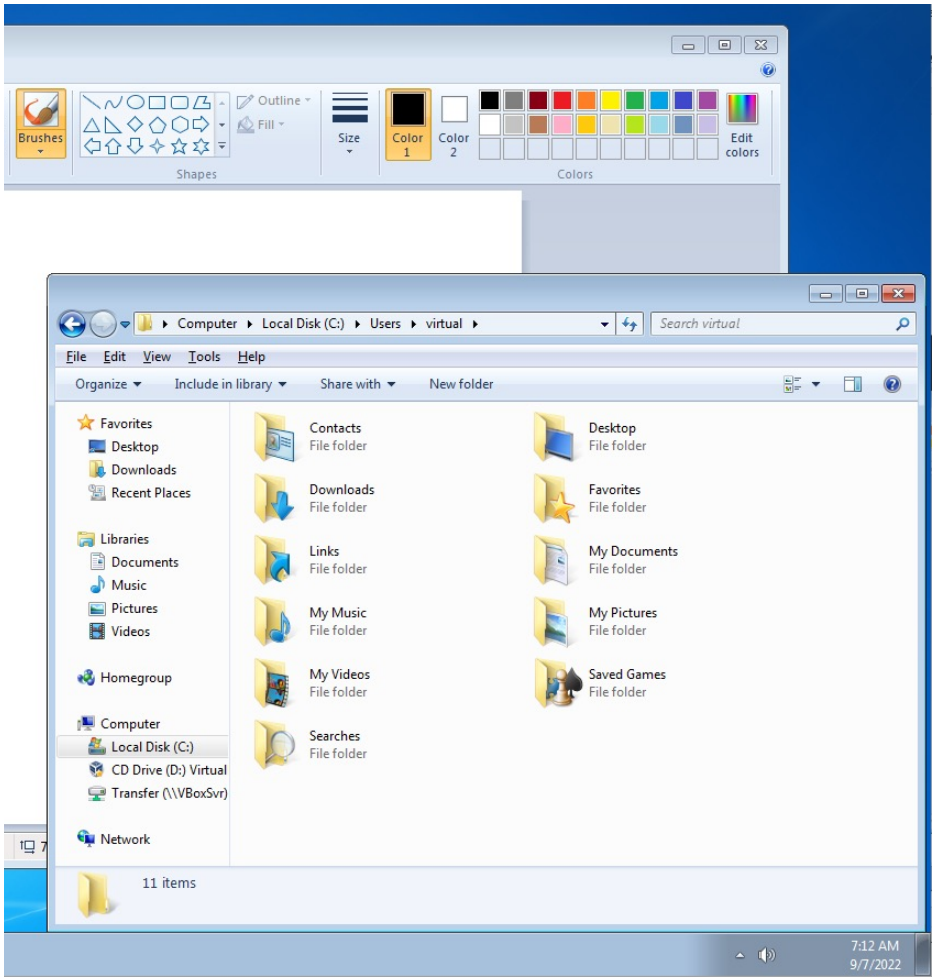
While the windows manager "owns" the application window, the application "owns" the content of the application window.



Window Manager – Additional Functionality

A window manager also provides:

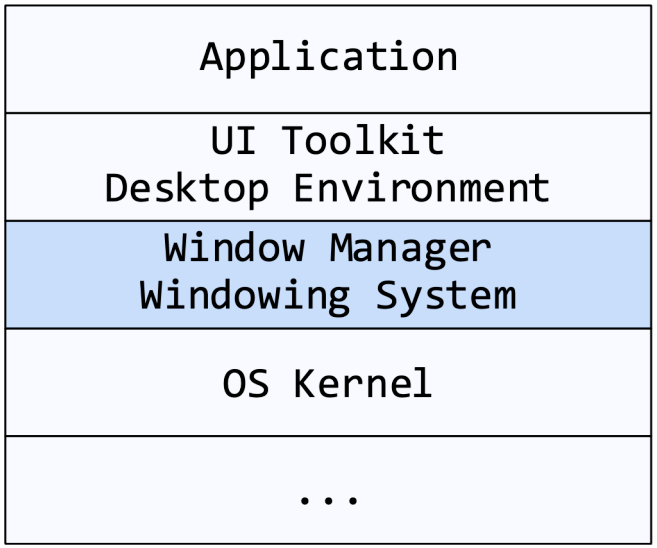
- Facilities to modify size and location of each window (resize handle, move handle, etc.)
- Window-related interactive components (close button, minimize button, etc.)

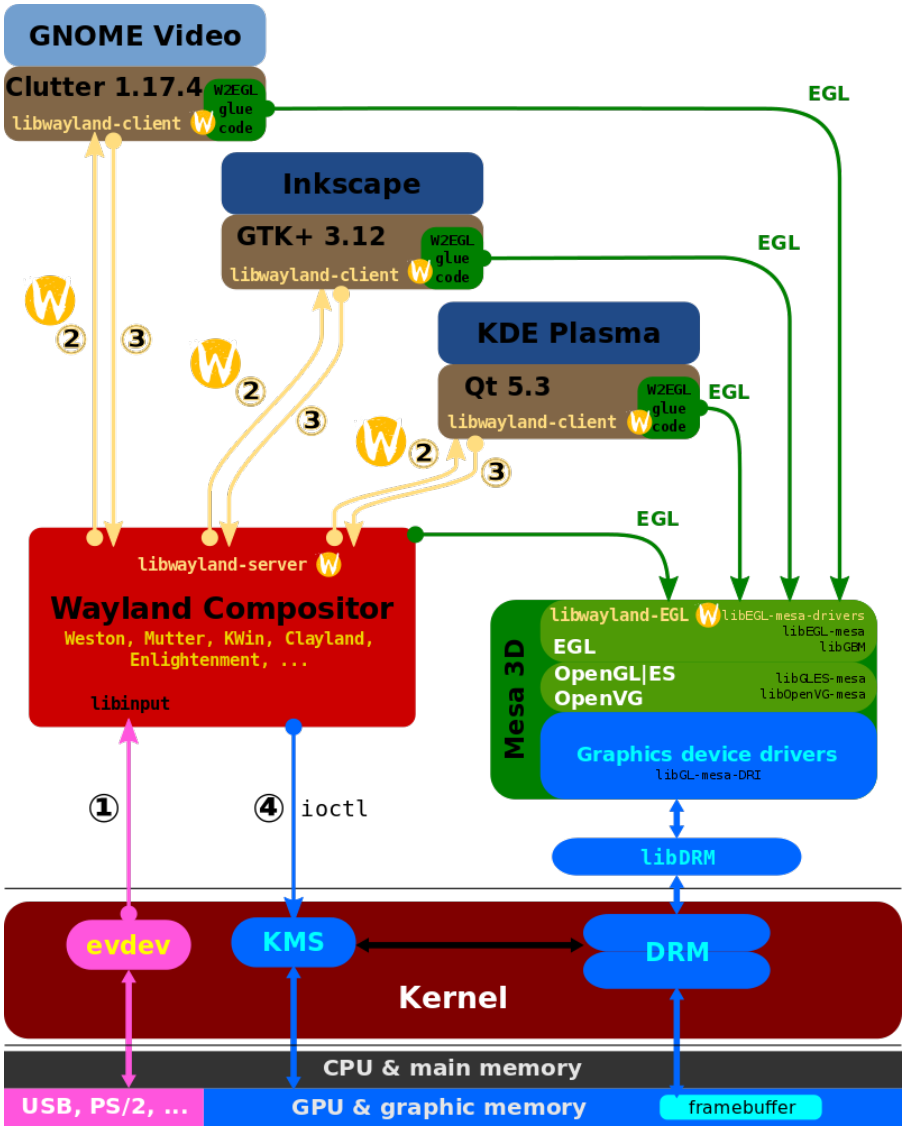


Window Manager – Architectures

Examples of Window Manager architectures:

- X11, Wayland (Linux)
- Quartz (macOS)
- Desktop Window Manager (Windows)

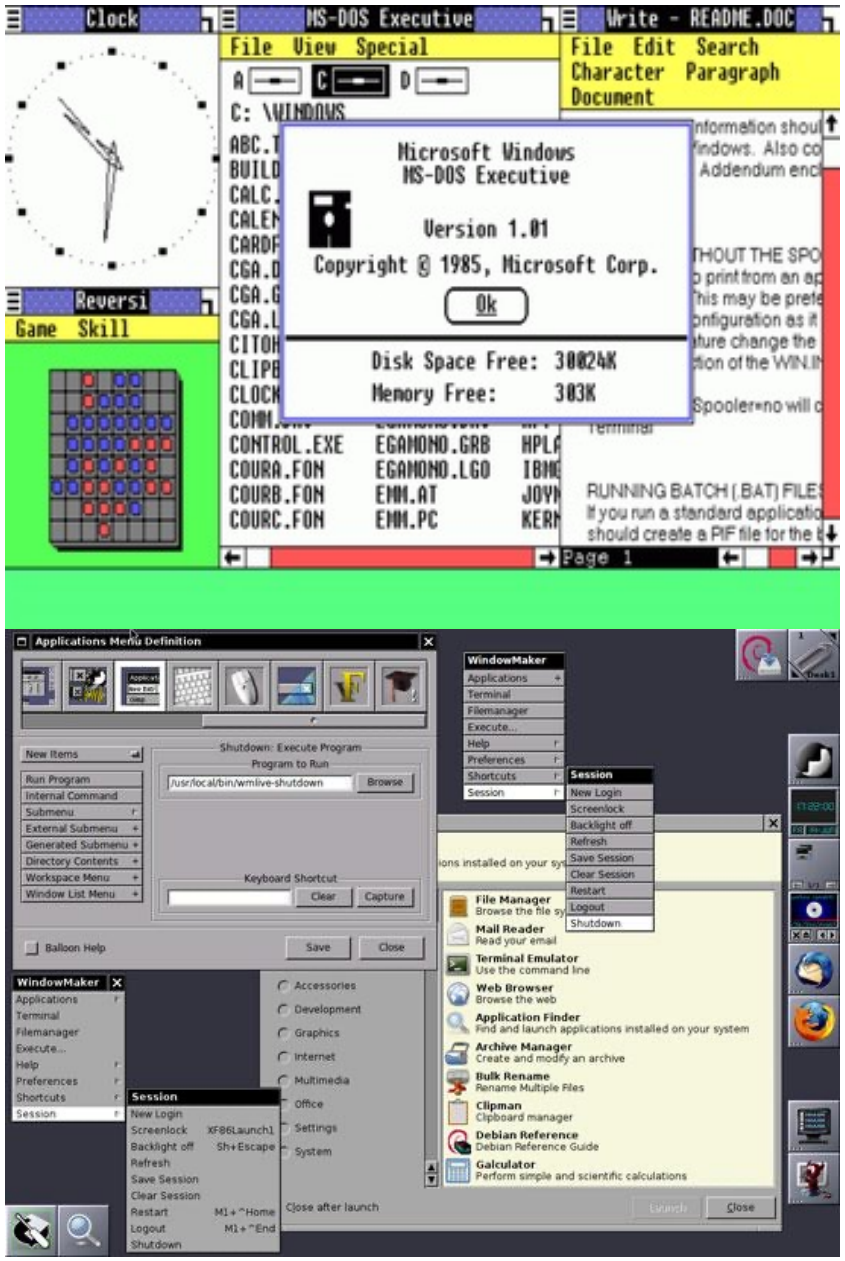




Window Manager – Examples

Examples of Window Managers:

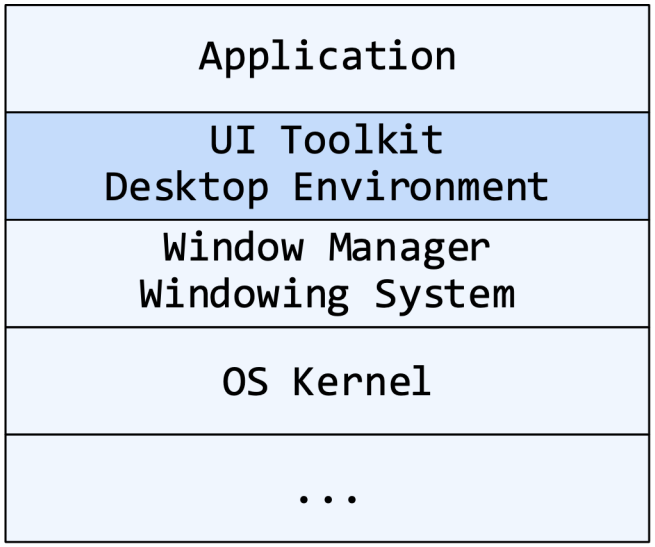
- Windows 1.0
  - 1985
  - Tiling
  - Integrated
- Window Maker
  - 1997
  - Stacking
  - X11 as *Windowing System;* e.g., GNUstep as *Desktop Environment*
- Mutter
  - 2011
  - Compositing
  - Wayland as *Windowing System;* e.g., GNOME as *Desktop Environment*



UI Toolkits

Window Managers include only **basic** capabilities for input, output, and window management.

For implementing the actual content of a UI, we need a **UI Toolkit – a set of classes for building User Interfaces**.



Low-level (or native or heavyweight) toolkits:

- Built into or tightly integrated with the underlying OS.
- Examples: Win32 on Windows, Xlib on Unix, Cocoa on Mac
- Often provided by OS vendor.

High-level (or lightweight) toolkits:

- Sit “above” the operating system, with no tight integration.
- Examples: Qt, Gtk+, wxWidgets, Swing, and JavaFX
- Often provided by a third-party.

Toolkit Features: I/O

Toolkits provide class abstractions for IO devices.

Input

- Keyboard
- Mouse (or pointing device)
- Cameras, sensors, *etc*.

Output

- User interface widgets
- Graphics primitives, e.g., shapes and images
- Animation
- Media

Toolkit Features: Desktop Functionality

Other “standard” desktop features are provided by toolkits.

Standard menu bars

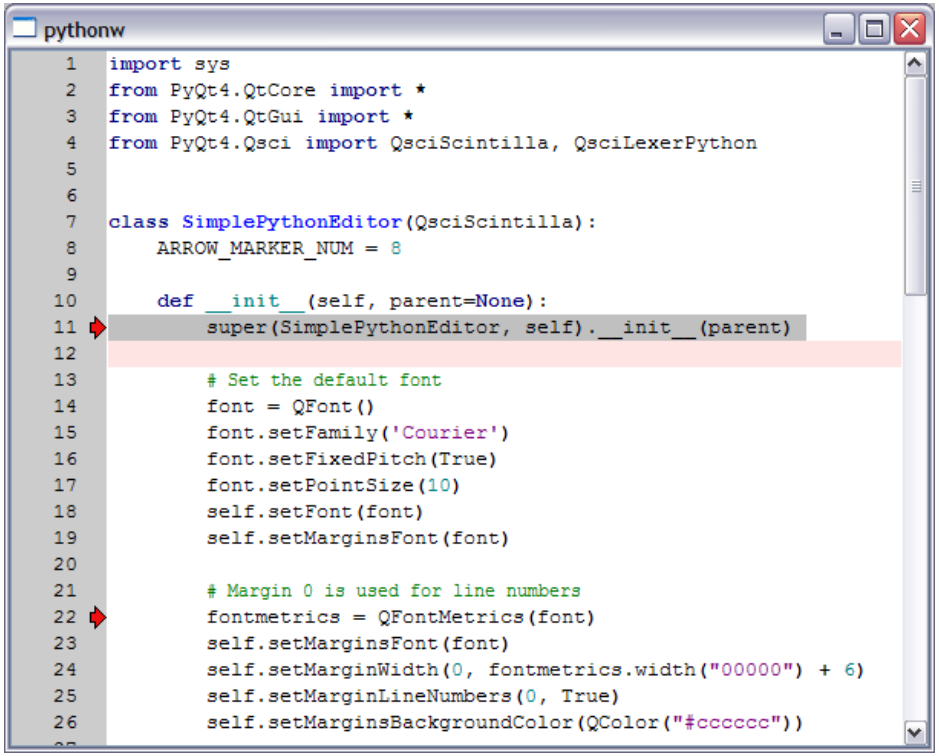
- File: New, Open, Close, Print, Quit.
- Edit: Cut, Copy, Paste.
- Window: Minimize, Maximize.
- Help: About.

- **Keyboard shortcuts**
- Ctrl-N for File-New, Ctrl-O for File-Open, Ctrl-Q for Quit.
- Ctrl-X for Cut, Ctrl-C for Copy, Ctrl-V for Paste.
- F1 for Help.

Toolkit Style 1: Imperative

Code is used to manually construct the view. Instantiate classes and set fields/properties.

Virtually every programming environment offers some ability to do this (e.g. Java/Swing, C++/Qt, Python, Javascript/HTML).



Python w. Qt toolkit

Benefits

- You have complete control over how objects are created and managed.

Drawbacks

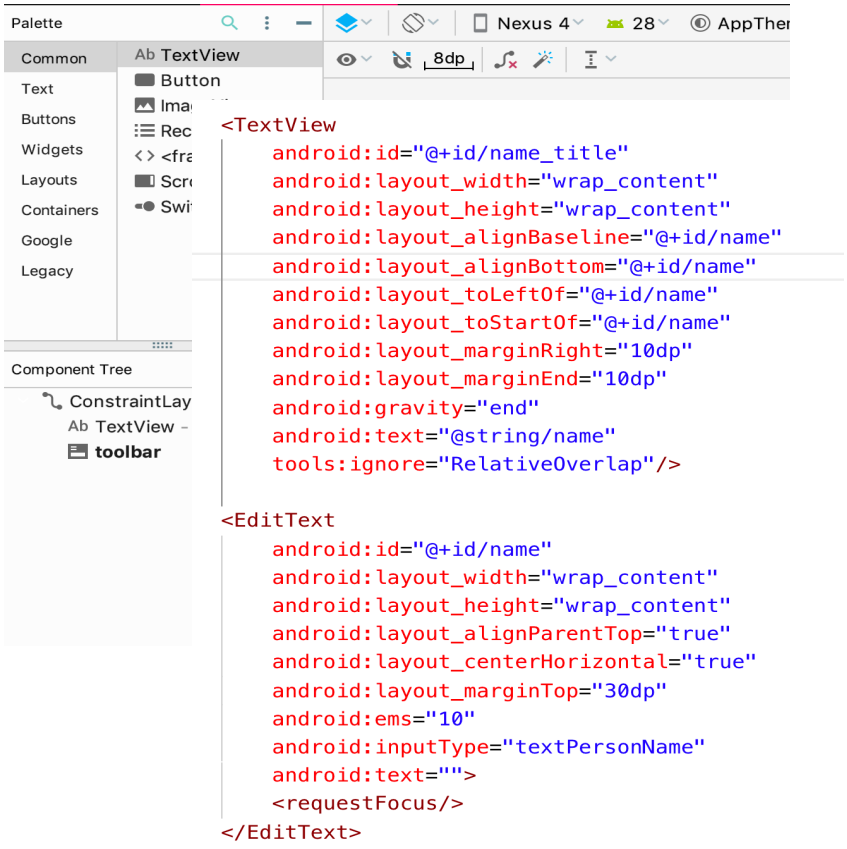
- Requires programming knowledge to create or change.
- It's can be tedious to build a complex UI in this fashion!

Toolkit Style 2: Declarative

The layout is described in some other format. Graphical elements are associated with code (somehow).

Format may be **binary** or **human-readable** (XML, JSON).

Android is an example of this: you describe a UI in XML, which is then loaded dynamically at runtime. Code is written in Java or Kotlin.



Android GUI builder and Layout

Benefits

- Non-programmers can build the UI.

Drawbacks

- May require proprietary tools to generate or modify to the UI.
- Binary formats cannot be ‘diff’d.

JavaFX

History of Java FX

- Java 1.0 (1996)
- Cross-platform
- Java wrappers for native widgets
- In practice, underlying platform differences meant that they looked and behaved differently across platforms
- Support imperative programming
- “heavyweight” toolkit

- Java 1.1 (1998)
- Cross-platform
- Java implementations of core widgets
- Often lower than native widgets, and missing modern features like animations, shading and so on.
- Support imperative programming
- “lightweight” toolkit

- Java 6 (2007)
- Cross-platform
- Java implementation of full framework + widgets
- Competitor w. Adobe Flash; designed for “rich multimedia apps”
- A “better Swing” with 3D, graphs, more controls.
- Imperative + declarative with GUI builder
- “Lightweight” toolkit

Create a JavaFX Project

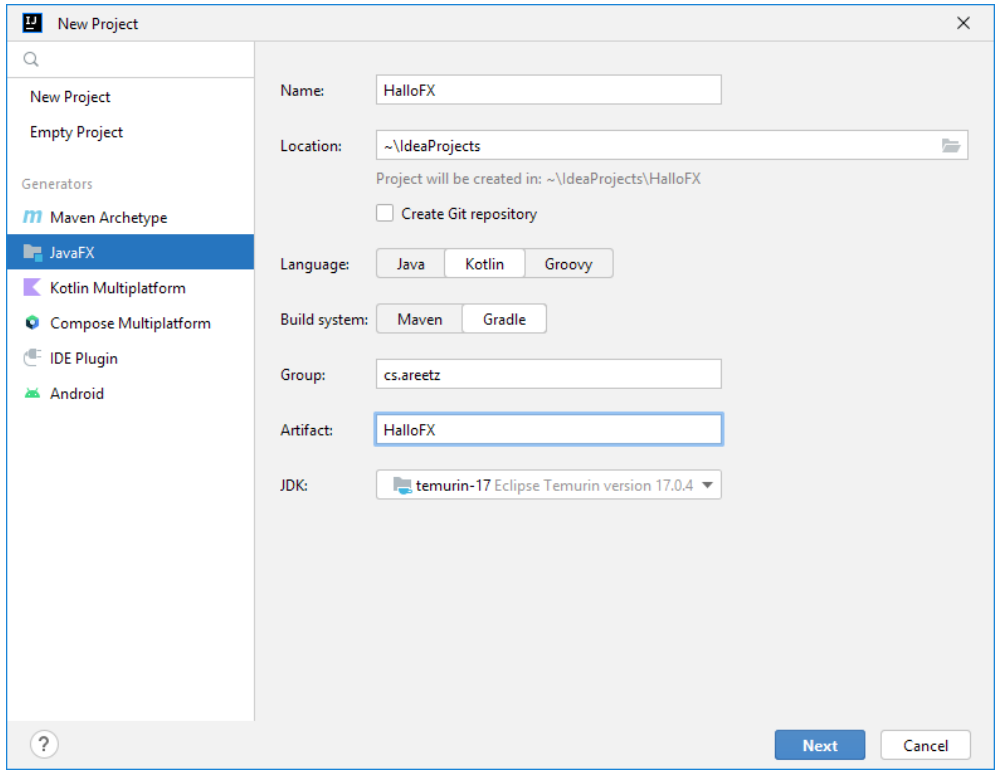
When creating a project:

Option 1: Empty Project

- Blank project.
- You can always add JavaFX dependencies by-hand.

Option 2: JavaFX

- Will create a populated project for you (declarative).
- May need to remove unused classes and change structure.



Option 1: Add JavaFX to an Existing/Empty Project

You will need to add the javafx dependencies to the project's `build.gradle` file, Gradle will download and import the libraries automatically.

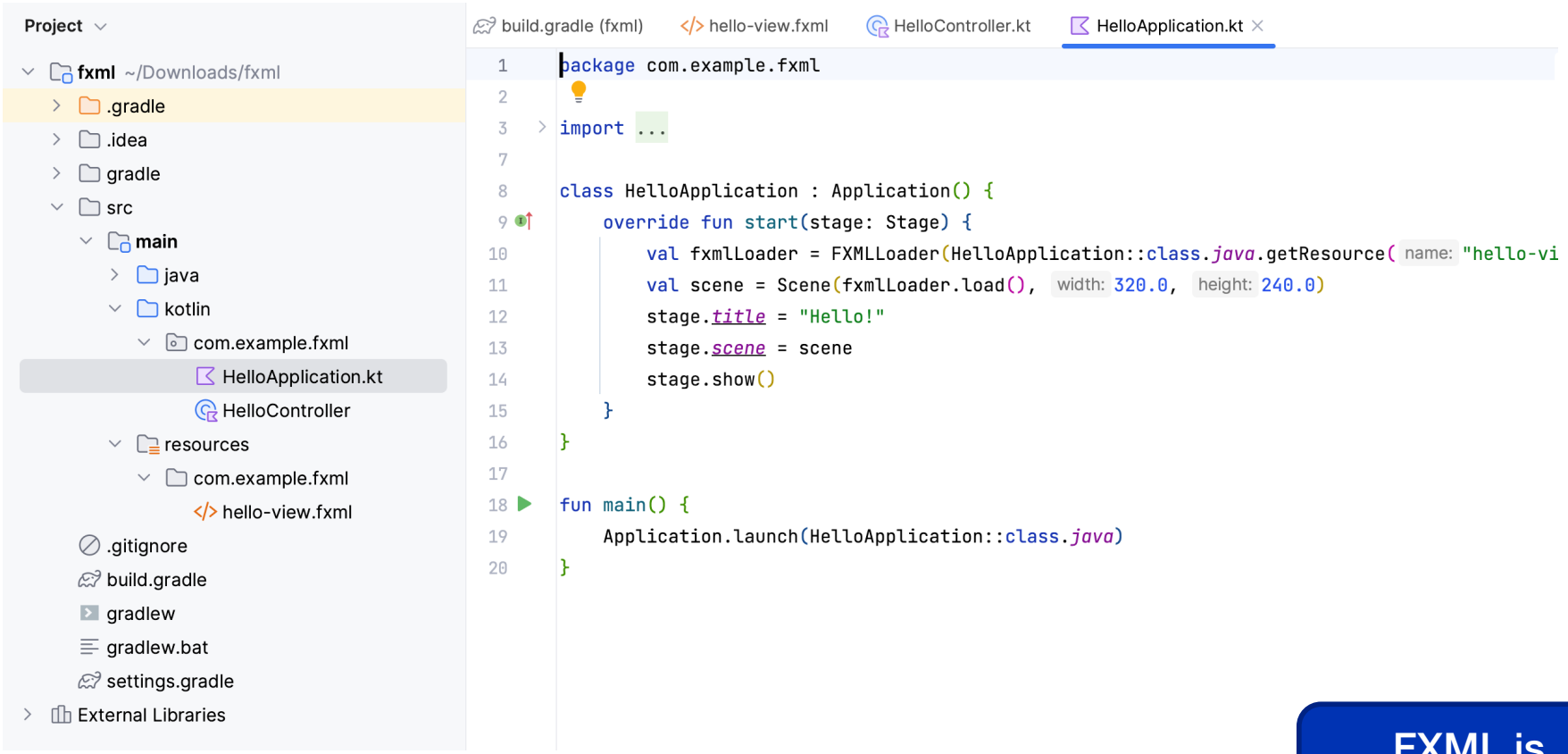
```
plugins {
    application
    kotlin("jvm") version "1.8.20"
    id("org.openjfx.javafxplugin") version "0.0.14"
}


application {
    mainClass.set("Main")
}


javafx {
    version = "18.0.2"
    modules("javafx.controls", "javafx.graphics")
}
```

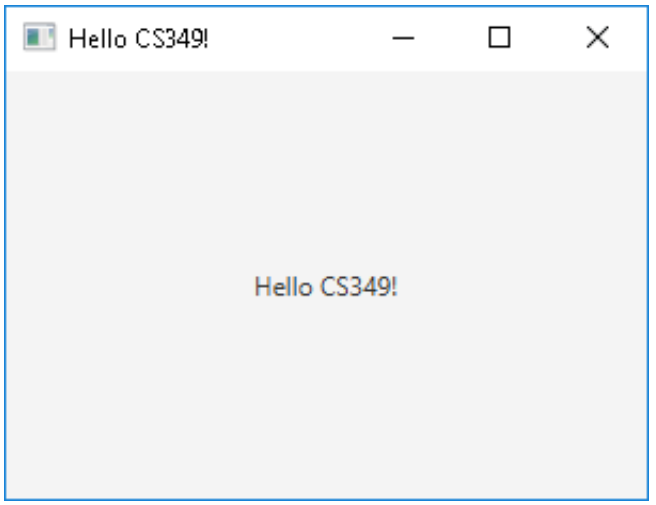
Option 2: JavaFX Wizard

This will generate some starter code and resources, which you may need to modify, *but* the build configuration doesn't need changes.



FXML is declarative…

Hello JavaFX



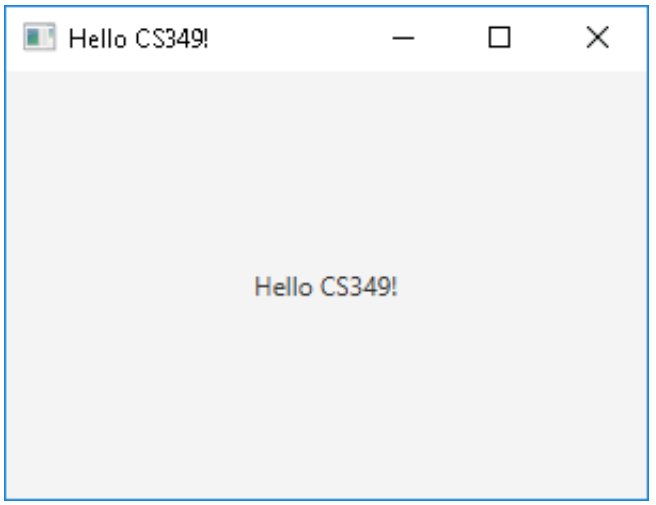

```
package ui.lectures.hellofx

import ...

class HelloApplication : Application() {
    override fun start(stage: Stage) {
        val root = Pane()
        val scene = Scene(root, 320.00, 240.00)

        stage.scene = scene
        stage.title = "Hello CS349!"
        stage.isResizable = false
        stage.show()
    }
```

Hello JavaFX




```
package ui.lectures.hellofx

import ...

class HelloApplication : Application() {
    override fun start(stage: Stage) {
        stage.apply {
            title = "Hello CS349!"
            scene = Scene(StackPane(Label("Hello CS349!")),
                                    300.0, 240.0)
        }.show()
    }
}
```

This implementation has a different style:

- The `StackPane` and the `Label` remain anonymous.
- We use `apply` to setup the stage in a single block.

Application Lifecycle

JavaFX applications extend the `Application` class, which is the core class in JavaFX.

The JavaFX runtime does the following when an application is launched:

- Creates an instance of the specified `Application` class
- Calls the instance's `init()` method
- Calls its `start()` method
- Waits for the application to finish, when either
  - the application calls `Platform.exit()`
  - the last application window has been closed.
- Calls its `stop()` method.

Most time is spent here, waiting for things to happen

The `start()` method is abstract and must be overridden.

The `init()` and `stop()` methods are optional but may be overridden.

Application Lifecycle

```
import javafx.application.Application
import javafx.stage.Stage

class Stages : Application() {
    override fun init() {
        super.init()
        println("init")
    }

    override fun start(stage: Stage) {
        println("start")
    }

    override fun stop() {
        super.stop()
        println("stop")
    }
}
```

Methods are invoked in this order.

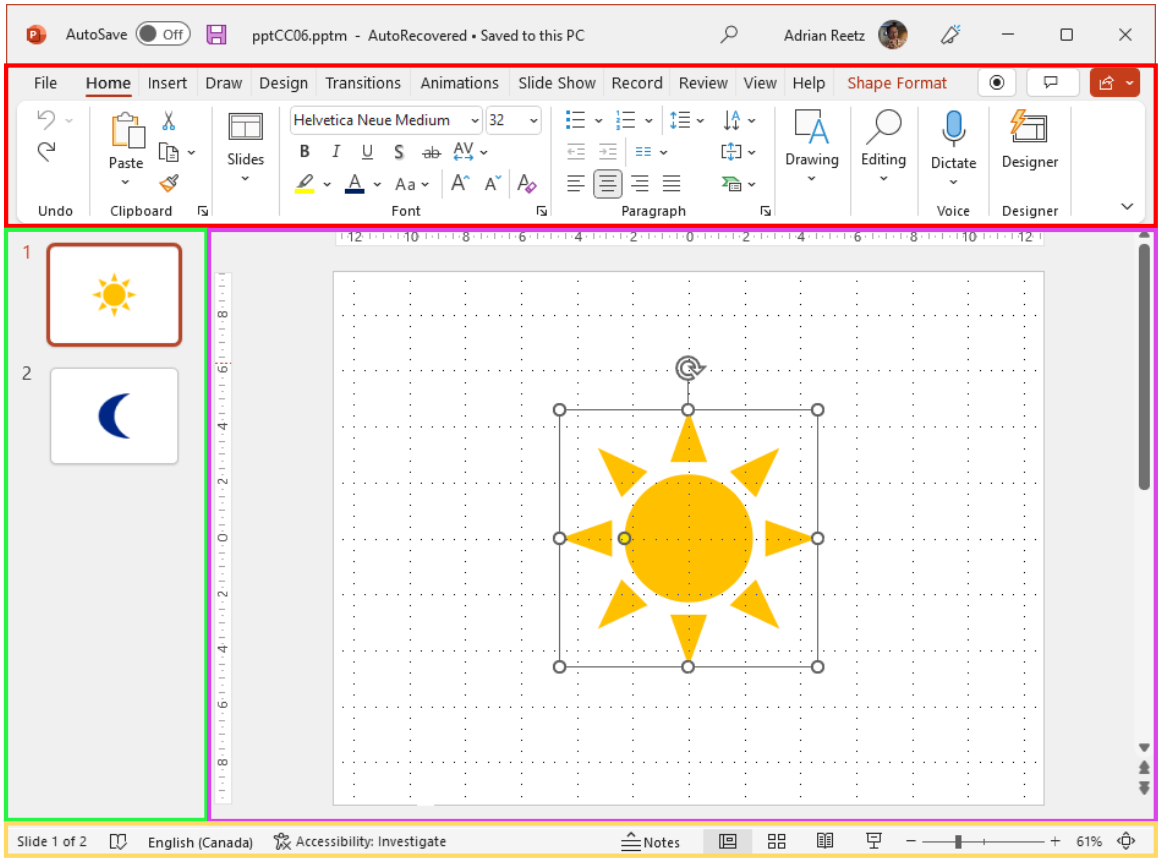
1. main()
2. init()
3. start()
4. stop()

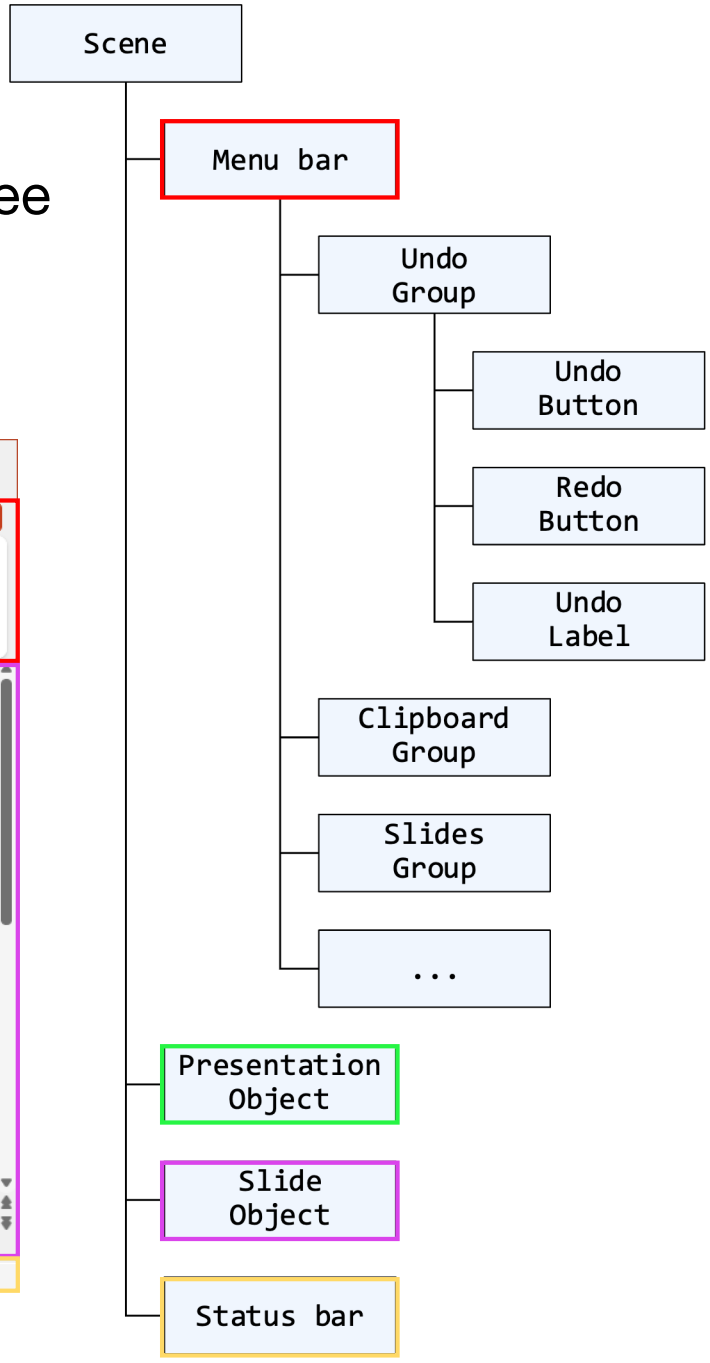
Note that all of these are abstract base class methods and have default implementations.

Start() is the only required method

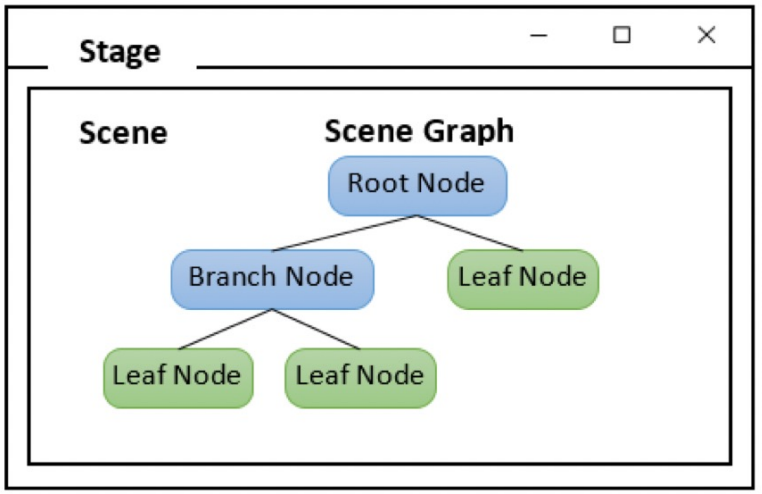
Scene Graph

In computer graphics, a **scene graph** is a tree structure that arranges all the elements of a screen into a hierarchy.





Scene Graph



In computer graphics, a **scene graph** is a tree structure that arranges all the elements of a screen into a hierarchy:

- Manages dependencies between objects on the screen
- Makes drawing, event dispatch, and other operations more efficient

JavaFX stores an interface as a scene graph.

- **Stage** is the main window
- **Scene** is the content of the application, which includes the scene-graph containing the UI
- Everything in a scene is a **Node**, ordered in a tree-like hierarchy

Stage – `javafx.stage.Stage`

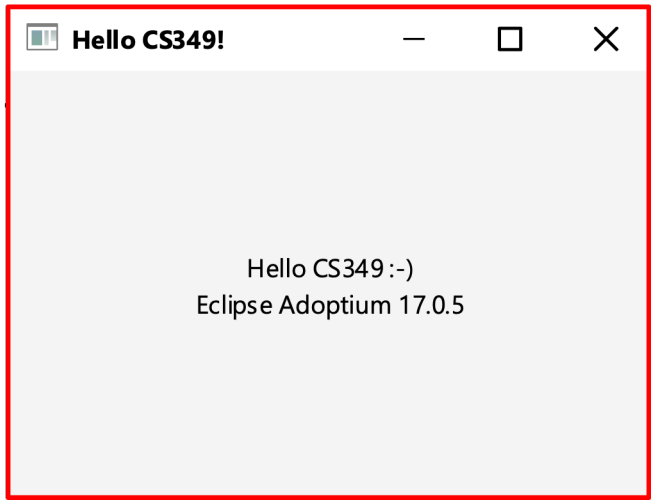
Stage is the top-level container, representing the entire application window. It is automatically created by the platform. Use properties to set or change behavior of the window.

```
override fun start(stage: Stage) {
    val greeting = Label("Hello CS349 :-)")

    val vendor = Label(System.getProperty("java.vendor"))
    val version = Label(System.getProperty("java.version"))
    val javaInfo = HBox(vendor, version).apply {
        alignment = Pos.CENTER
    }

    val root = VBox(greeting, javaInfo).apply {
        alignment = Pos.CENTER
    }

    stage.apply {
        scene = Scene(root, 300.0, 200.0)
        title = "Hello CS349!"
    }.show()
}
```



Scene – `javafx.scene.Scene`

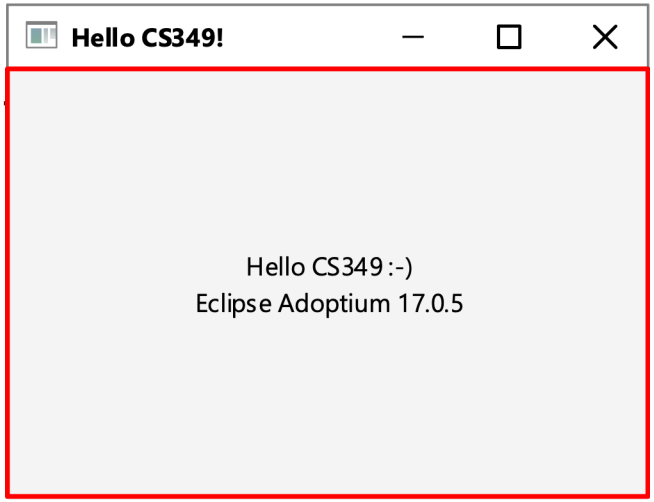
Scene is the container for the content. It must specify the root node for the scene graph.

```
override fun start(stage: Stage) {
    val greeting = Label("Hello CS349 :-)")

    val vendor = Label(System.getProperty("java.vendor"))
    val version = Label(System.getProperty("java.version"))
    val javaInfo = HBox(vendor, version).apply {
        alignment = Pos.CENTER
    }

    val root = VBox(greeting, javaInfo).apply {
        alignment = Pos.CENTER
    }

    stage.apply {
        scene = Scene(root, 300.0, 200.0)
        title = "Hello CS349!"
    }.show()
}
```



Nodes – `javafx.scene.Node`

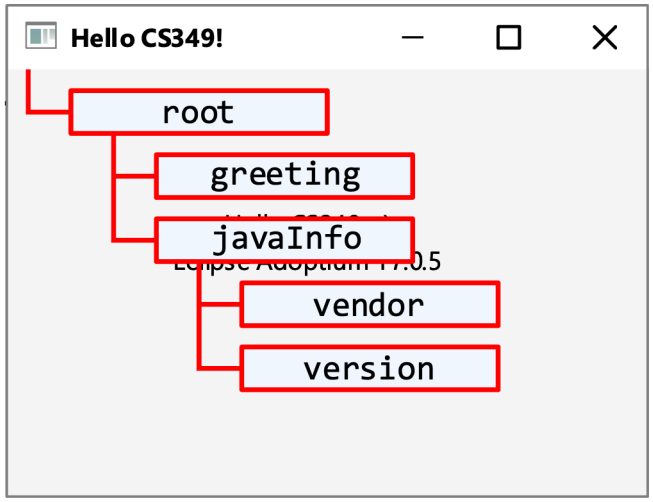
Nodes are either the displayable objects or layouts for structuring displayable objects.

```
override fun start(stage: Stage) {
    val greeting = Label("Hello CS349 :-)")

    val vendor = Label(System.getProperty("java.vendor"))
    val version = Label(System.getProperty("java.version"))
    val javaInfo = HBox(vendor, version).apply {
        alignment = Pos.CENTER
    }

    val root = VBox(greeting, javaInfo).apply
        alignment = Pos.CENTER
    }

    stage.apply {
        scene = Scene(root, 300.0, 200.0)
        title = "Hello CS349!"
    }.show()
}
```



Nodes – `javafx.scene.Node`

Root Node

- If a `Group` is used as the root, the contents of the scene graph will be clipped by the scene's width and height.
- If a resizable node (layout `Region` or `Control` is set as the root, then the root's size will track the scene's size, causing the contents to be resized as necessary.

Internal Nodes

- Layouts, such as: `Group`; (`Region`); `Pane`: `GridPane` , `StackPane`, `VBox`, etc.

Leaf Nodes

- Controls (“Widgets”), such as: `Button`, `Choicebox`, `Label`, `Slider`, `Spinner`, etc.
- Shapes, such as: `Circle`, `Line`, `Polygon`, `Rectangle`, `Text`, etc.

What can we draw on a Scene?

In this course, we will focus on the following:

Layouts (`javafx.scene.layout` subclasses)

- `HBox`, `VBox`, `Pane`, `FlowPane`, `GridPane`, `StackPane`, `TilePane`, etc.

Controls ("Widgets") (`javafx.scene.control` subclasses)

- Accordion, ButtonBar, ChoiceBox, ComboBoxBase, HTMLEditor, Labeled, ListView, MenuBar, Pagination, ProgressIndicator, ScrollBar, ScrollPane, Separator, Slider, Spinner, SplitPane, TableView, TabPane, TextInputControl, ToolBar, TreeTableView, TreeView

Graphics Primitives (`javafx.scene.shape` subclasses)

- Arc, Circle, CubicCurve, Ellipse, Line, Path, Polygon, Polyline, QuadCurve, Rectangle, SVGPath, and Text

In upcoming lectures, we will talk about each of these in greater detail.

End of the Chapter

- The elements of the UI stack.
- Scene Graph, Scene Graph, **Scene Graph**!

Any further questions?