

-



X

# Introduction

Motivation

Course Content

U

CS 349

January 09

# Instructor

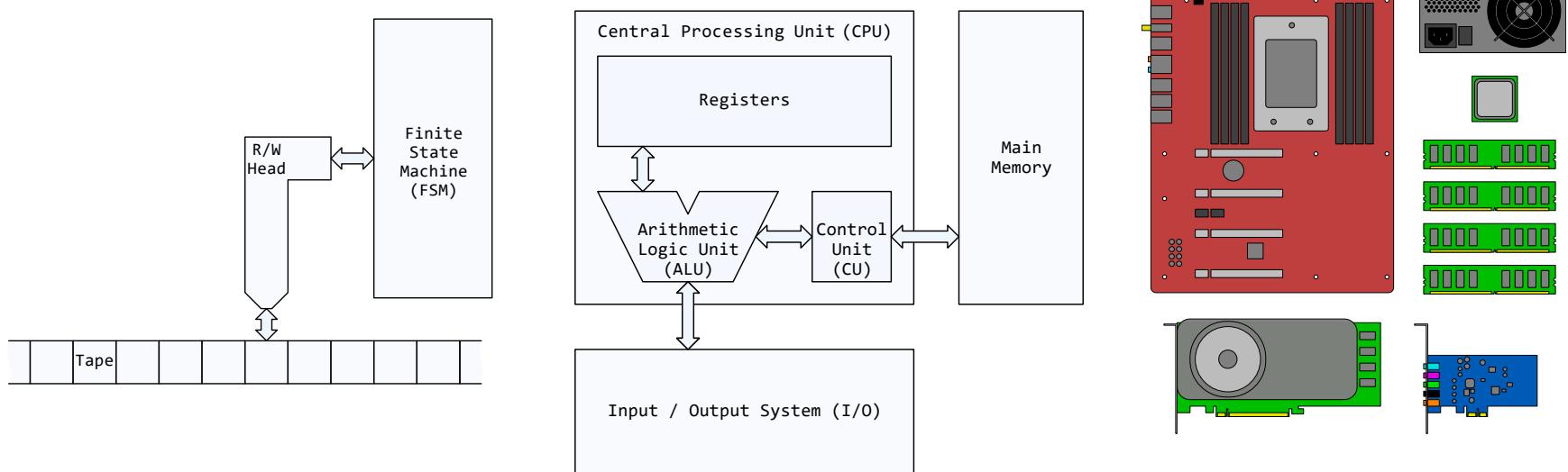


- Name: Adrian Reetz
- Email: [adrian.reetz@uwaterloo.ca](mailto:adrian.reetz@uwaterloo.ca)
- Office: DC 3121

U  
CS 349

# Motivation

# How Expert Users Describe Computers



How would an average user describe a computer?

# How Average Users Describe Computers



write essays,  
edit videos,  
respond to emails



get directions,  
take photos,  
connect to friends



get from A to B



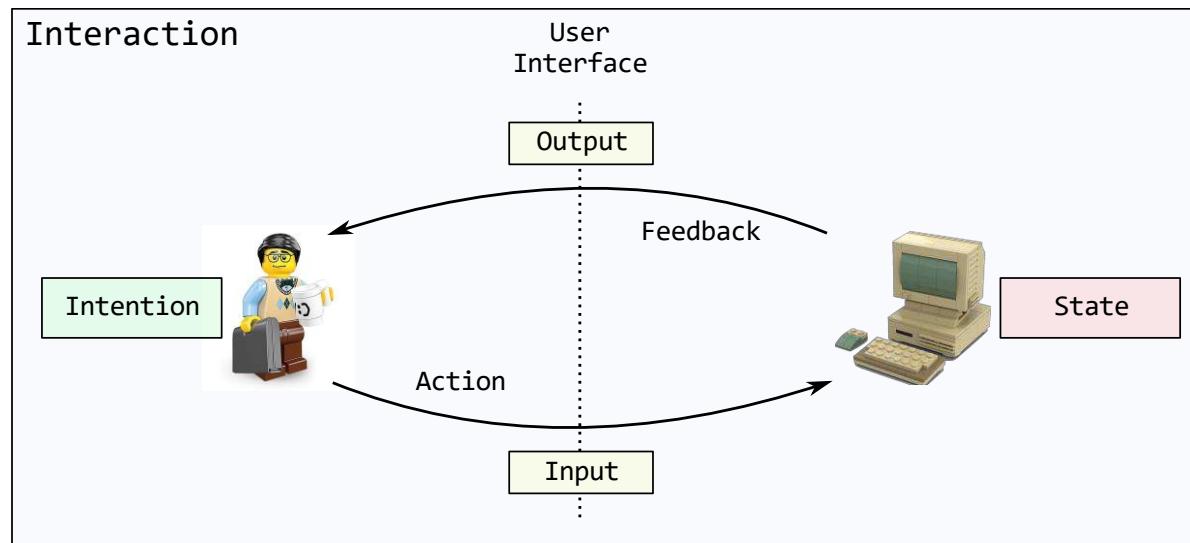
dig hole

For average users, a computer is just a tool they use to perform different tasks. Users interact with this tool through a so-called *user interface* (or: *UI*). Users just want an UI that helps them accomplish their tasks **effectively**.

In this course, we focus on building effective user interfaces.

# Human-Computer Interaction

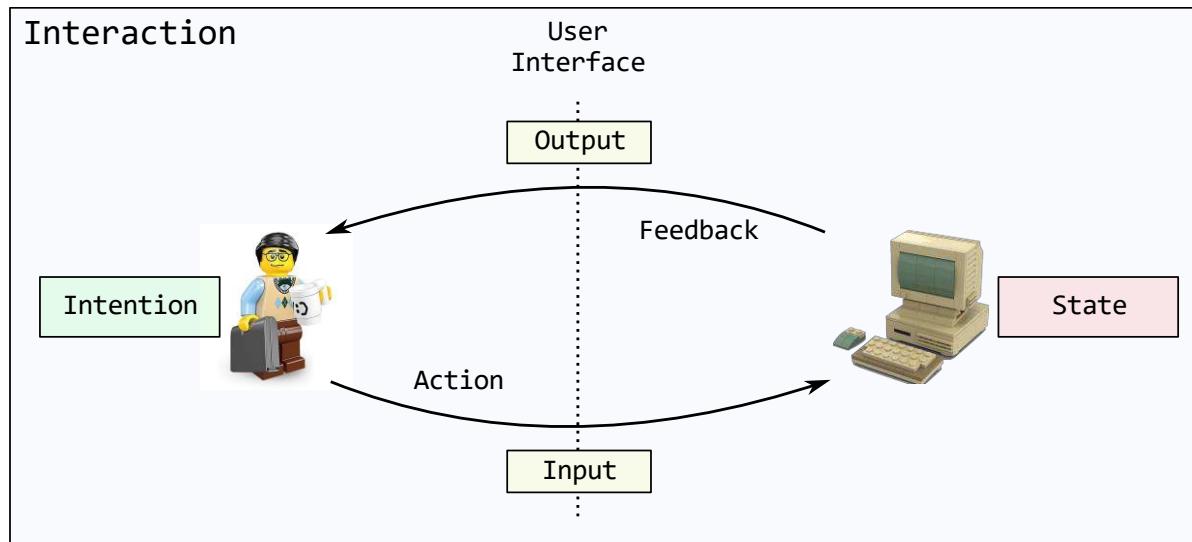
Interaction is the process where a user expresses their intention to a system (“input”), and the system presents feedback to the user (“output”).



# Human-Computer Interaction and User Interfaces

Interaction refers to actions by user and feedback by the system over time:

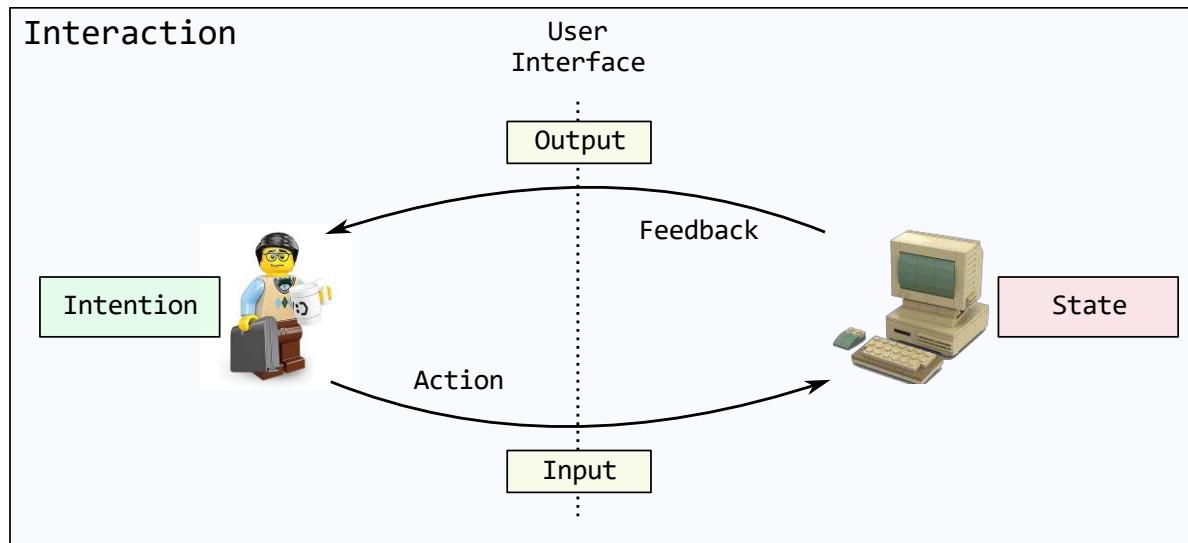
- Interaction is a dialog between the user and system
- Alternates between the user manipulating controls through input and the system responding with feedback via output



# Human-Computer Interaction and User Interfaces

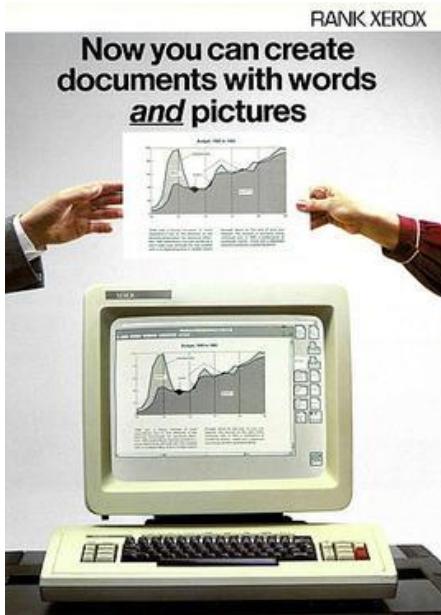
A User Interface is the external presentation of the system to the user:

- Action: how users communicate their intention to the system
- Feedback: what the system uses to communicate its (new) state or response to an action



# User Interfaces

In this course, we primarily focus on graphical user interfaces



# User Interfaces

Good UIs empower people to do things they could not otherwise do.

- e.g., music production, e-commerce, assistive technologies, ...

Good UIs create digital tools than can change the world.

- e.g., social media, video streaming, photography, ...



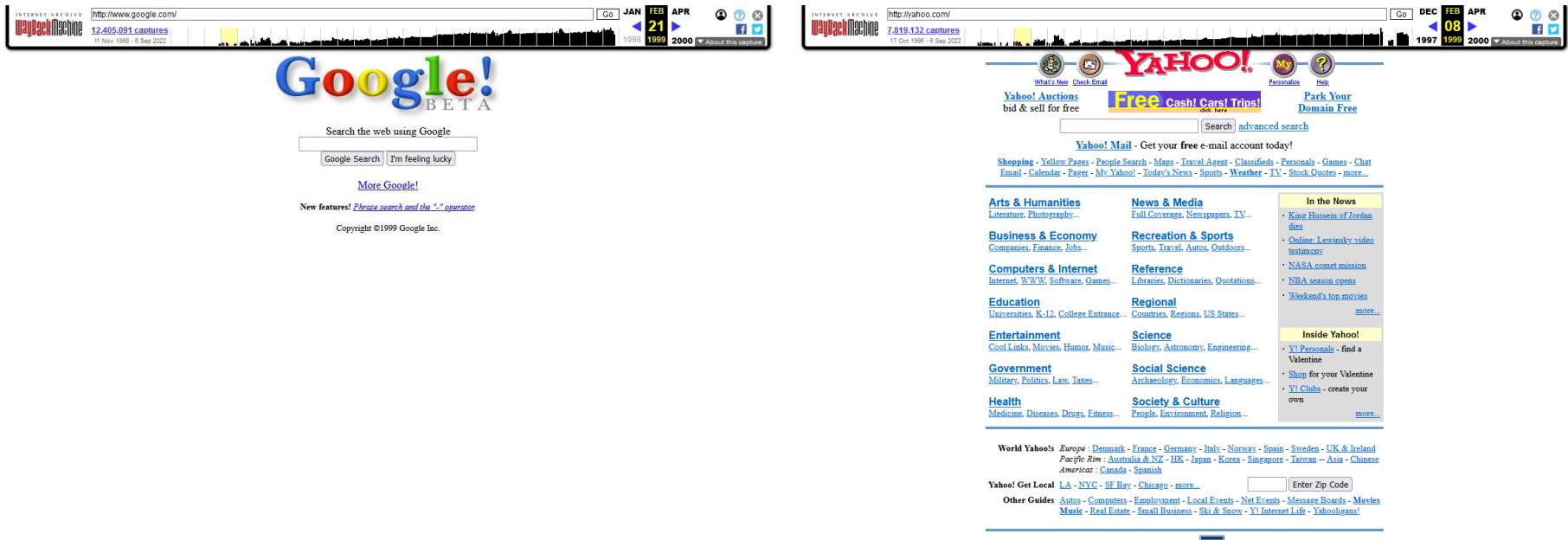
# User Interfaces

A well-designed and well-implemented user interface is a **critical part** of successful user-focused software and digital tools.



# User Interfaces

A well-designed and well-implemented user interface is a **critical part** of successful user-focused software and digital tools.



# Course Content

U

CS 349

# CS349 – User Interfaces

This course focuses on

- creating user interfaces (UIs), including underlying UI architecture and algorithms,
- practice implementing UIs using existing frameworks, and
- theories and methods relevant to interface design.

<https://student.cs.uwaterloo.ca/~cs349>

# CS349 – User Interfaces

## Goals

- The focus of this course is on building user-interfaces.
- Our overall objective is to teach you to build compelling and useful user-interfaces, across a variety of platforms and devices. (Kotlin / JavaFX & Kotlin / Android)

# CS349 – User Interfaces

## Learning Objectives

- Understand the architecture, algorithms, and design principles underlying common user-interfaces (and UI frameworks)
- Develop and demonstrate the ability to implement a compelling and useful UI on both desktop and mobile platforms.
- Articulate and use basic theories and methods for UI design.
- Leverage HCI research directly related to building user-interfaces.

# Course Structure

See <https://student.cs.uwaterloo.ca/~cs349/1231/schedule/>

Week	Date	Topics	Due
1	Jan 9 - 13	Introduction, History, Kotlin	
2	Jan 16 - 20	JavaFX, Widgets	
3	Jan 23 - 27	Layout, Events	Q1
4	Jan 30 - Feb 3	Model-View-Controller, GUI Interaction	
5	Feb 6 - 10	GUI Interaction, Drawing	A1
6	Feb 13 - 17	Graphics, Transformations, Hit-testing	Q2
	Feb 20 - 24	Reading Week	
7	Feb 27 - Mar 3	Animation, Input	A2
8	Mar 6 - 10	Mobile UI, Android	Q3
9	Mar 13 - 17	Android	A3
10	Mar 20 - 24	Responsiveness, Undo-Redo	Q4
11	Mar 27 - 31	Input Performance	
12	Apr 3 - 7	Accessibility, The Future of Interaction	A4
	Apr 10		Q5

# Assessment

- Assignments: 4x, 15% each 60%
- Quizzes: 5x, 4% each 20%
- Final exam: 1x 20%

Assignments and the final exam are manually graded. Quizzes are auto-graded by Learn.

All grades are posted to Learn.

# Quizzes

- Available on Learn: only during a 24-hour period, 30 minutes to complete after starting
- No late submissions accepted
- Review of the lecture material from the previous two to three weeks (NOT cumulative)
- Topics will be finalized closer to the quiz dates
- Consist of multiple-choice, true / false, and short-answer questions
- See <https://student.cs.uwaterloo.ca/~cs349/1231/quizzes/>

#	Topics	Writing Time Frame
Q1	Introduction, History, Kotlin, JavaFX, Widgets, Layout, Events	Fri, Jan 27, 12:01 am to 11:59 pm
Q2	MVC, GUI Interaction, Drawing, Graphics, Transformations	Fri, Feb 17, 12:01 am to 11:59 pm
Q3	Hit-Testing, Animation, Input, Mobile UI	Fri, Mar 10, 12:01 am to 11:59 pm
Q4	Android, Undo-Redo, Responsiveness	Fri, Mar 24, 12:01 am to 11:59 pm
Q5	Input Performance, Accessibility	Mon, Apr 10, 12:01 am to 11:59 pm

# Assignments

- Develop on your machine (talk to us if you run into any problems)
- Individual work, not group work; please review academic integrity
- Submitted with git to a personal CS349 Waterloo GitLab repository we generated for you
- Late policy is 5% per 24 hours, up to 48 hours
- See <https://student.cs.uwaterloo.ca/~cs349/1231/assignments/>

#	Title	Due Date
A1	TBD	Fri, Feb 10, 6 pm
A2	TBD	Fri, Mar 3, 6 pm
A3	TBD	Fri, Mar 17, 6 pm
A4	TBD	Thu, Apr 6, 6 pm

# Getting Help

## Piazza

- Class announcements and news
- Class forum to discuss lecture topics, clarify assignments, technical troubleshooting, etc.
- Please sign up with your real name!

## Microsoft Teams for Office Hours

- At least 1 hour per day, Mon – Fri
- During scheduled times, post to “Office Hours” channel in our course “Team”
- Schedule will be posted at the start of week 2.

See <https://student.cs.uwaterloo.ca/~cs349/1231/about/help/>

# Getting Started

Explore the course website: <https://student.cs.uwaterloo.ca/~cs349/>

- Review policies (specifically, academic integrity, due dates).
- Review the slides.

Setup the Gradle / IntelliJ toolchain:

<https://student.cs.uwaterloo.ca/~cs349/1231/getting-started/>

# End of the Chapter



Please make sure to

- understand how your mark is calculated,
- review all deadlines carefully, and
- familiarize yourself with the CS349 website.



Any further questions?

-



# The History of Interaction

A Brief History of Computers

A History of Early User Interfaces

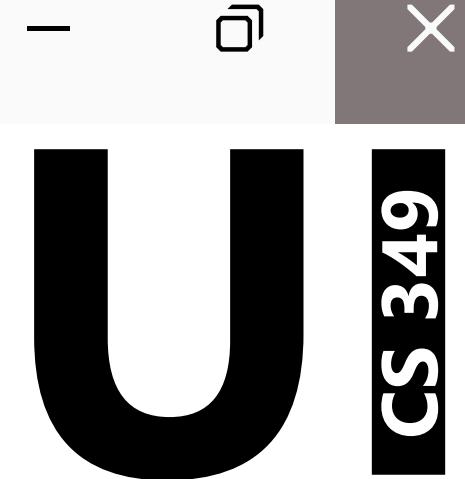
Command-line Interfaces

Graphical User Interfaces and the WIMP paradigm

U

CS 349

## January 09



# A Brief History of Computers

# What is a computer?

[English] Compute: calculate

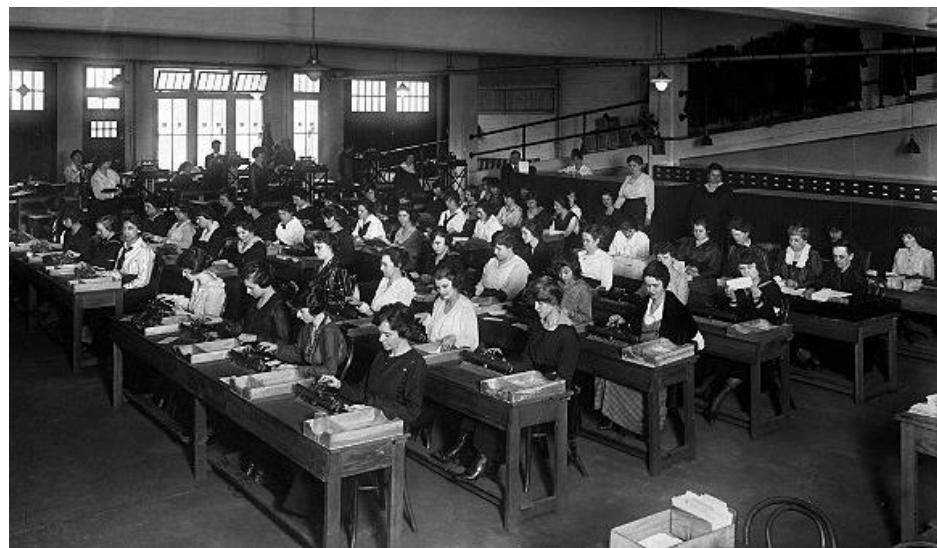
[French] computer: drawing  
calendars according to  
astronomical data

[Latin] com puto: arranging  
together

“I have read the truest computer of  
times...”

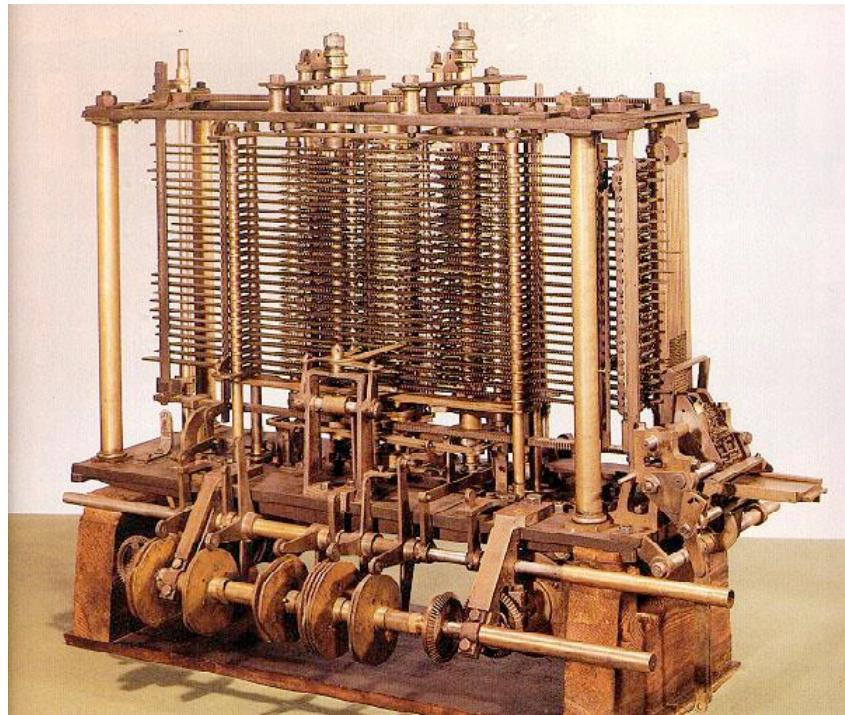
— Richard Brathwait, The Yong  
Mans Gleanings, 1613

Until the late 1800s, a computer  
was strictly a person, not a device.



# Mechanical vs. Electro-mechanical

*Analytical Engine* by Charles Babbage, 1837, never built

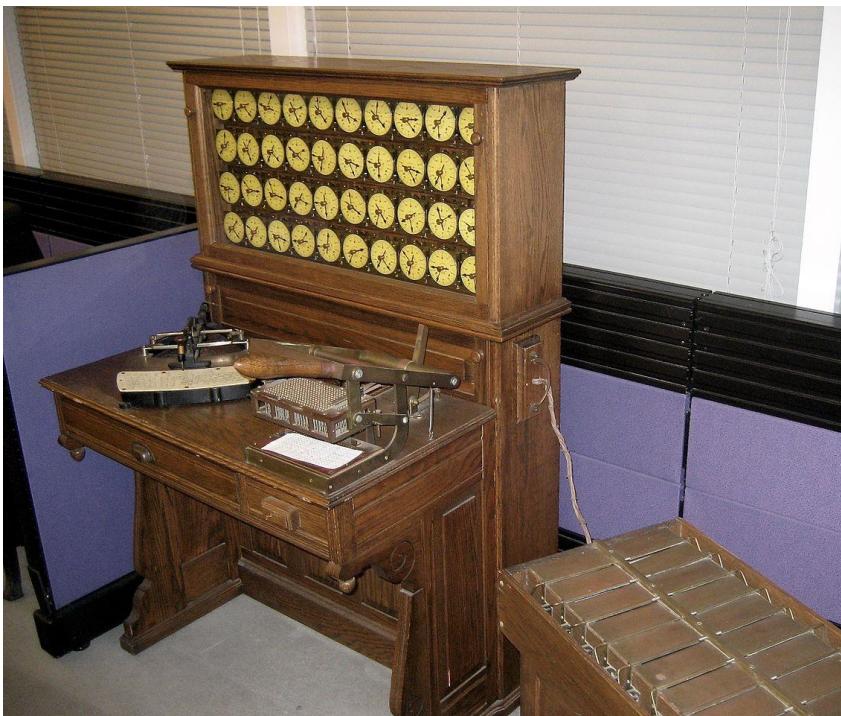


*Tabulating Machine* by Herman Hollerith, 1890

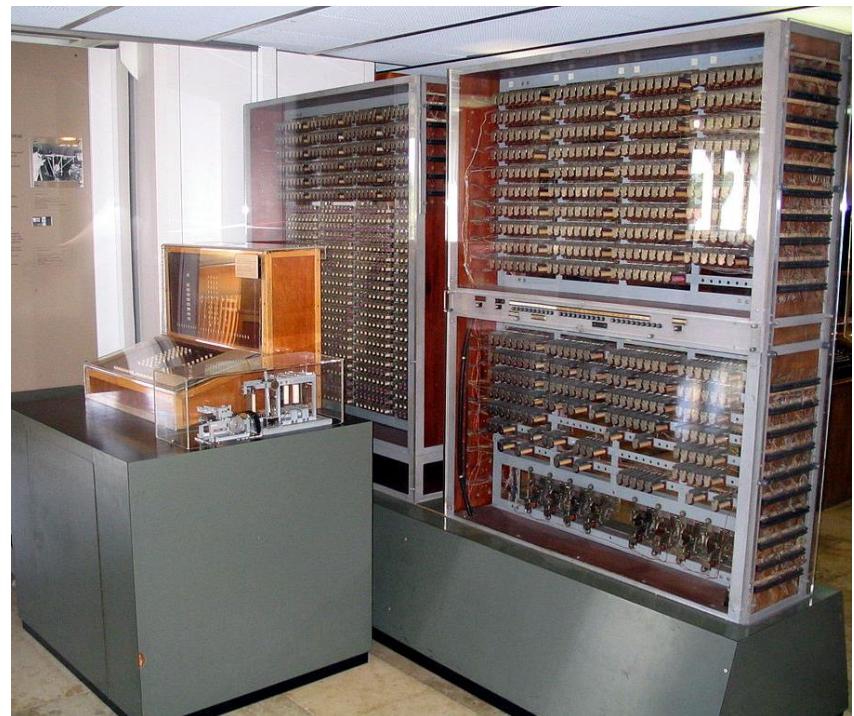


# Analog vs. Digital

*Tabulating Machine* by Herman Hollerith, 1890

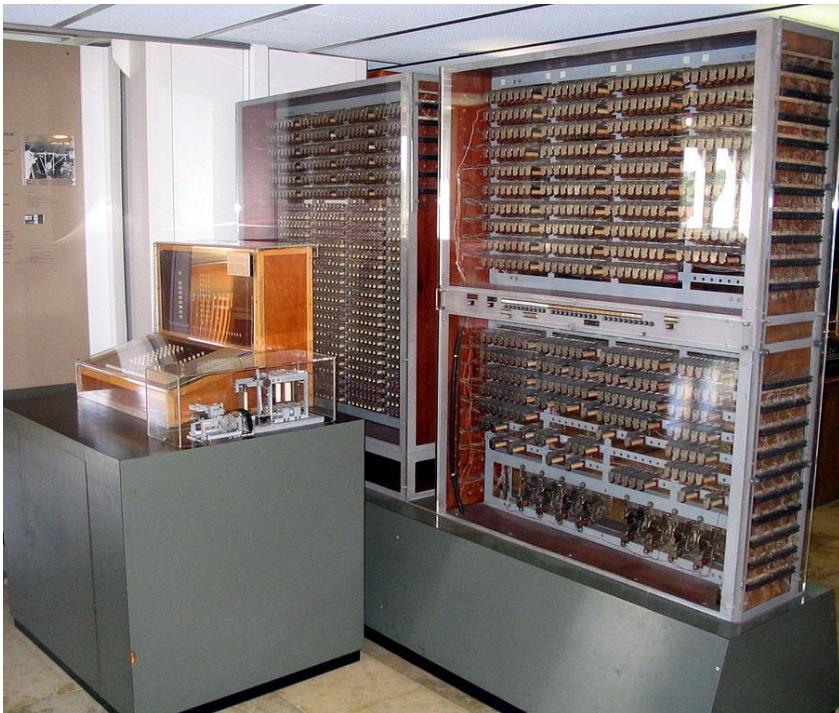


Z3 by Konrad Zuse, 1941

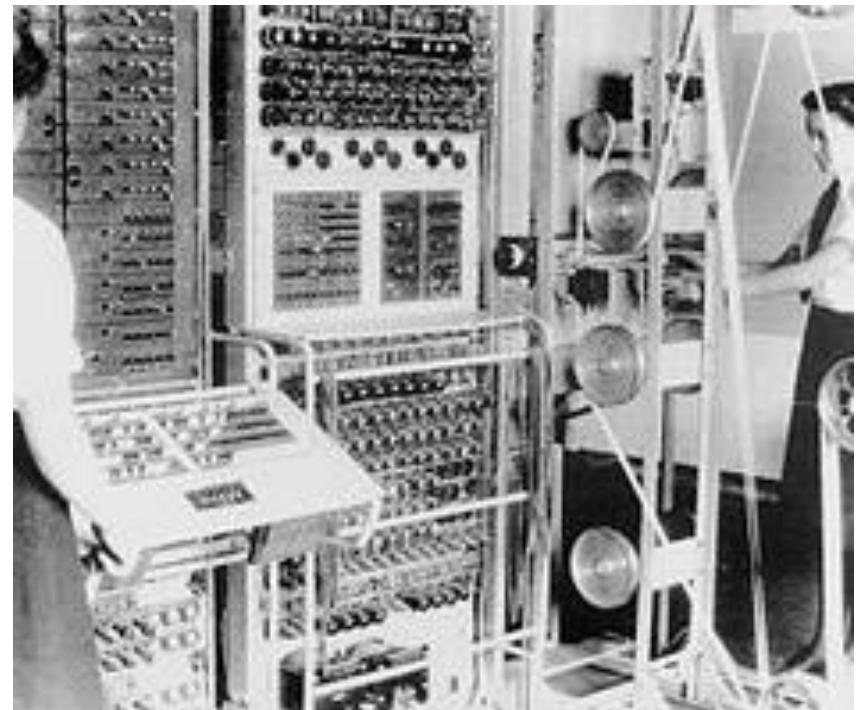


# Electro-mechanical vs. Electrical

Z3 by Konrad Zuse, 1941



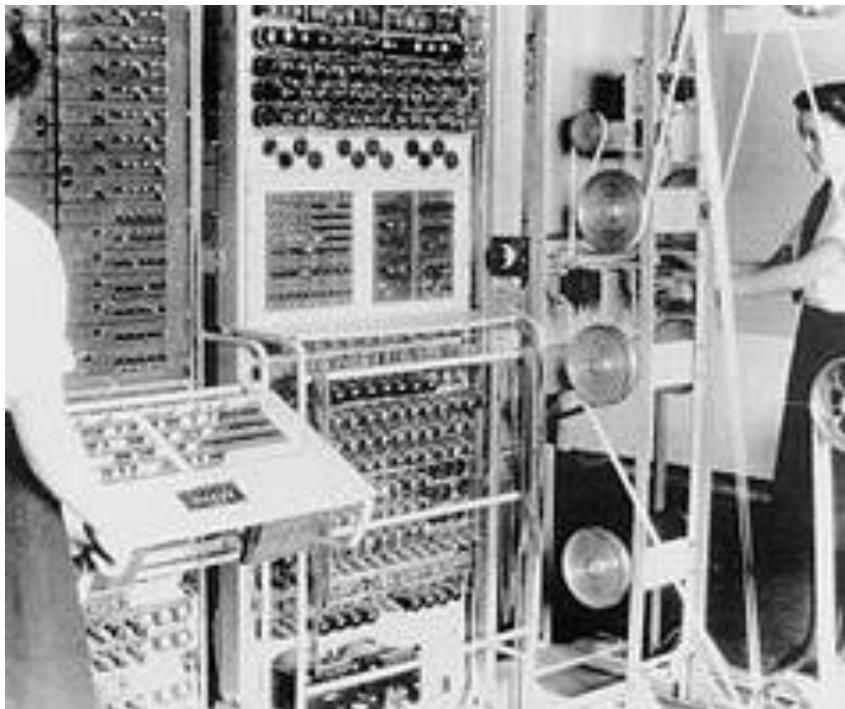
*Colossus* by Alan Turing, 1943



# Electrical vs. Integrated Circuits

*Colossus* by Alan Turing, 1943

*7070* by IBM, 1958

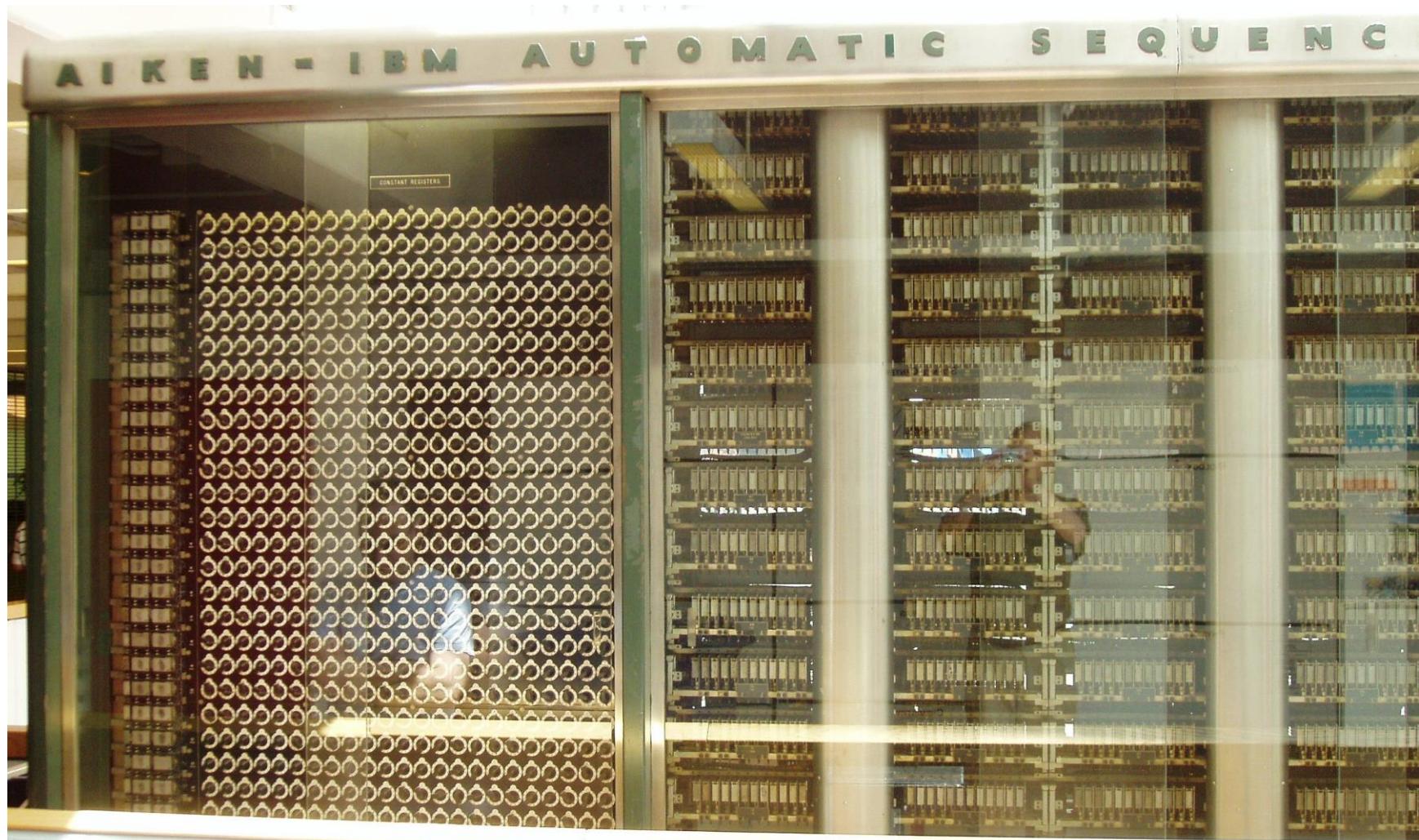


# A History of Early User Interfaces

U

CS 349

# Dials, Knobs, and Lights (until 1940s)



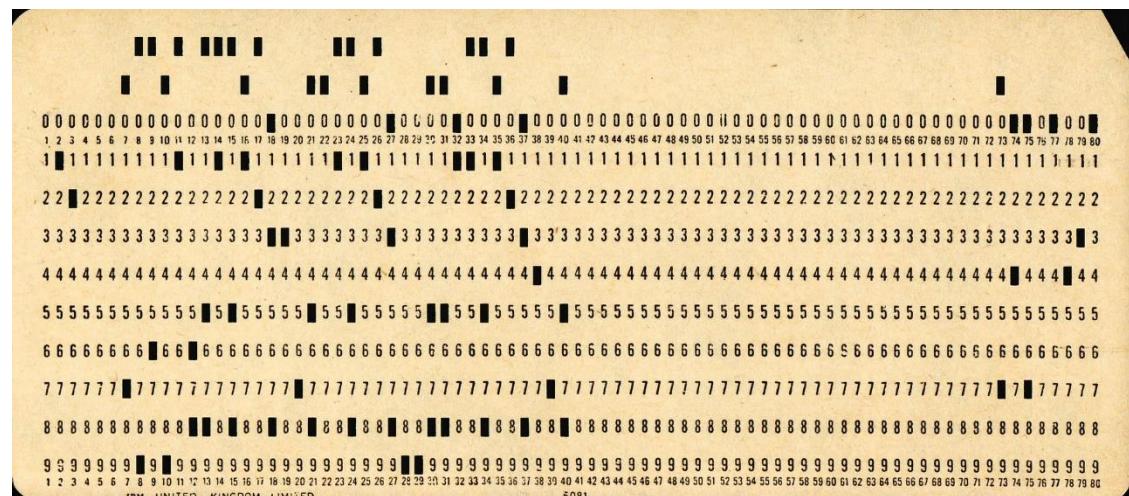
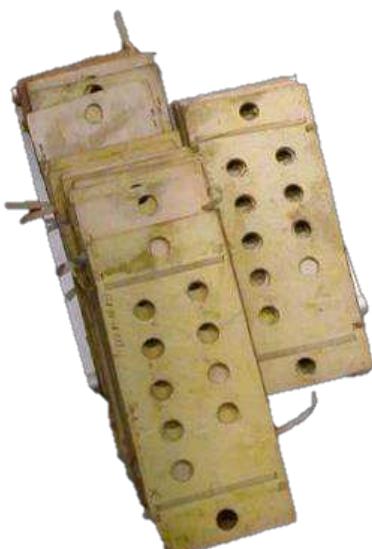
# Punch Cards and Batch Interface (1945 – 1965+)

## Interaction style

- Set of prepared instructions fed to computer via punch cards, paper tape, or magnetic tape
- Response typically received at the end via paper printout
- No real interaction possible while system executes instructions
- Responses received in hours or days

## Users

- Highly trained individuals



U

CS 349

# Command-Line Interfaces (CLIs)

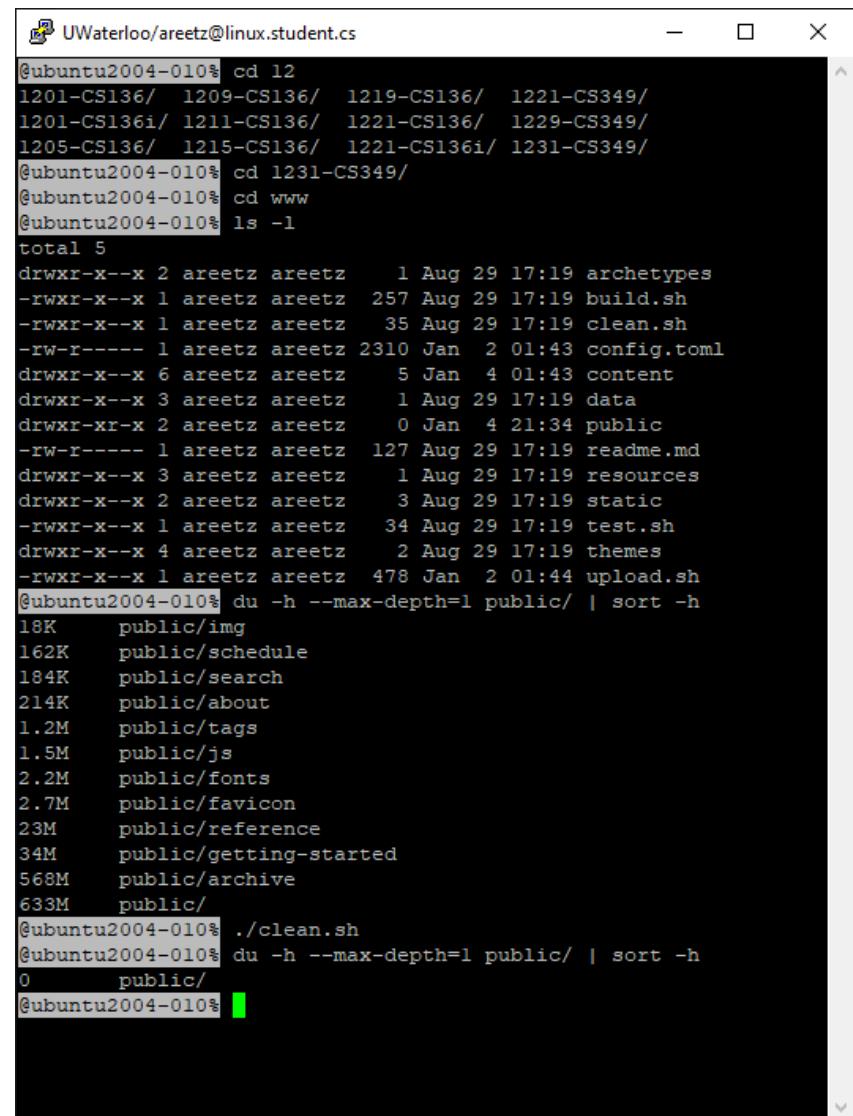
# Command-Line Interface (1965 – 1985+)

## Interaction style

- Commands are typed out via keyboard
- Feedback via screen, oftentimes given during execution
- Feedback received within seconds or minutes

## Users

- Trained experts



The screenshot shows a terminal window titled "UWaterloo/areetz@linux.student.cs". The user is navigating through a directory structure and performing file operations. The terminal output is as follows:

```
@ubuntu2004-010% cd 12
1201-CS136/ 1209-CS136/ 1219-CS136/ 1221-CS349/
1201-CS136i/ 1211-CS136/ 1221-CS136/ 1229-CS349/
1205-CS136/ 1215-CS136/ 1221-CS136i/ 1231-CS349/
@ubuntu2004-010% cd 1231-CS349/
@ubuntu2004-010% cd www
@ubuntu2004-010% ls -l
total 5
drwxr-x--x 2 areetz areetz 1 Aug 29 17:19 archetypes
-rwxr-x--x 1 areetz areetz 257 Aug 29 17:19 build.sh
-rwxr-x--x 1 areetz areetz 35 Aug 29 17:19 clean.sh
-rw-r----- 1 areetz areetz 2310 Jan 2 01:43 config.toml
drwxr-x--x 6 areetz areetz 5 Jan 4 01:43 content
drwxr-x--x 3 areetz areetz 1 Aug 29 17:19 data
drwxr-xr-x 2 areetz areetz 0 Jan 4 21:34 public
-rw-r----- 1 areetz areetz 127 Aug 29 17:19 readme.md
drwxr-x--x 3 areetz areetz 1 Aug 29 17:19 resources
drwxr-x--x 2 areetz areetz 3 Aug 29 17:19 static
-rwxr-x--x 1 areetz areetz 34 Aug 29 17:19 test.sh
drwxr-x--x 4 areetz areetz 2 Aug 29 17:19 themes
-rwxr-x--x 1 areetz areetz 478 Jan 2 01:44 upload.sh
@ubuntu2004-010% du -h --max-depth=1 public/ | sort -h
18K    public/img
162K   public/schedule
184K   public/search
214K   public/about
1.2M   public/tags
1.5M   public/js
2.2M   public/fonts
2.7M   public/favicon
23M    public/reference
34M    public/getting-started
568M   public/archive
633M   public/
@ubuntu2004-010% ./clean.sh
@ubuntu2004-010% du -h --max-depth=1 public/ | sort -h
0      public/
@ubuntu2004-010%
```

# Command-Line Interface – Advantages

## Powerful and highly flexible

- Many combinations of options can be supplied: `chmod -[ aAbBcCdDfFgG... ]`
- Piping from output to input: `ls -a -l | more`
- Batching, macroing: `#!/usr/bin/env bash`

Built-in documentation / man-pages:

`man ls`

In the original Unix tradition, command-line options are single letters preceded by a single hyphen...The original Unix style evolved on slow ASR-33 teletypes hat made terseness a virtue; thus the single-letter options.

— Eric Steven Raymond, The Art of Unix Programming

# Command-Line Interface – Disadvantages

Command names and their syntax is difficult to learn and need to be memorized.

Command-line interfaces are **not** explorable.

```
UWaterloo/areetz@linux.student.cs - □ X
@ubuntu2004-010% d
Display all 399 possibilities? (y or n)
darc
    dh_missing
dash
    dh_mkdirs
data2inc
    dh_movefiles
data2inc-3.0.4
    dh_numpy
date
    dh_numpy3
dawg2wordlist
    dh_perl
db2greg
    dh_perl_db
db2ls
    dh_perl_openssl
db5.3_archive
    dh_prep
db5.3_checkpoint
    dh_python2
db5.3_deadlock
```

```
UWaterloo/areetz@linux.student.cs - □ X
LS(1) User Commands LS(1)

NAME
    ls - list directory contents

SYNOPSIS
    ls [OPTION]... [FILE]...

DESCRIPTION
    List information about the FILES (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

    Mandatory arguments to long options are mandatory for short options too.

    -a, --all
        do not ignore entries starting with .

    -A, --almost-all
        do not list implied . and ..

    --author
        with -l, print the author of each file

    -b, --escape
        print C-style escapes for nongraphic characters

    --block-size=SIZE
        with -l, scale sizes by SIZE when printing them;
        e.g., '--block-size=M'; see SIZE format below

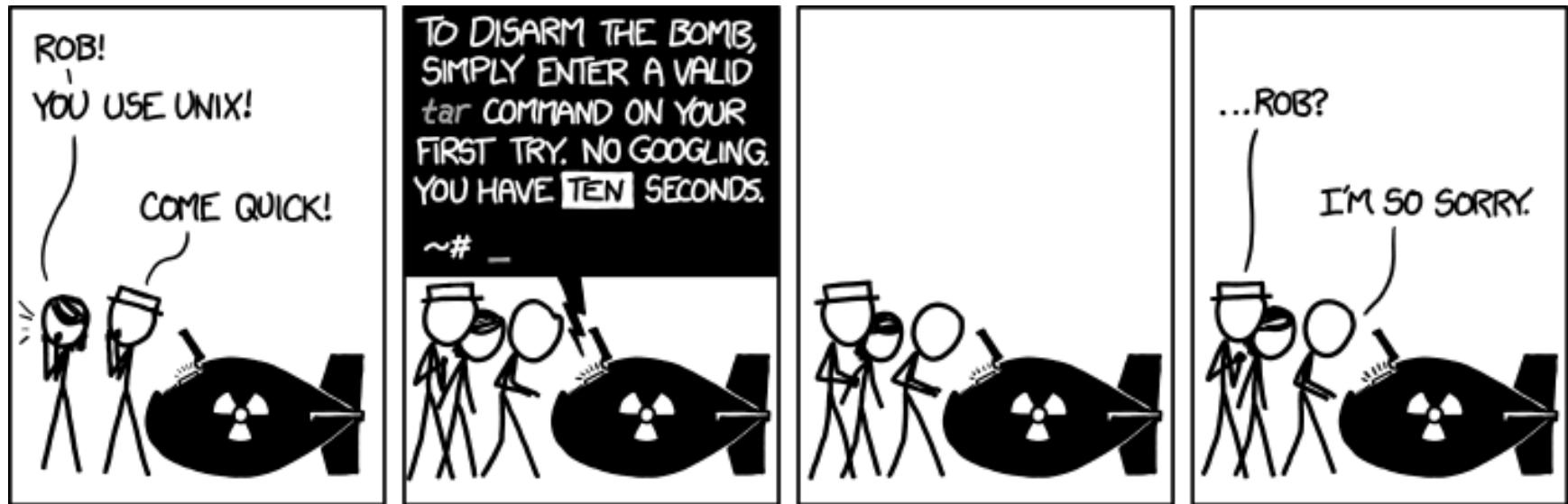
    -B, --ignore-backups
        do not list implied entries ending with ~

    -c
        with -lt: sort by, and show, ctime (time of last modification of file status information); with -l: show ctime and sort by name; otherwise: sort by ctime, newest first

    -C
        list entries by columns

Manual page ls(1) line 1 (press h for help or q to quit)
```

# Command-Line Interface – Disadvantages



*tar* by XKCD, <https://xkcd.com/1168/>

# Command-Line Interface – Conclusion

CLIs can be highly efficient for trained users but at the cost of

- being difficult to learn to use and
- being almost completely non-explorable.

They, however,

- Are biased towards expert users, intimidating for non-expert use
- Require recall of commands rather than recognition of capabilities

# Text-based User Interface

## Interaction style

- Commands are issued via keyboard shortcuts or arrow keys

## Users

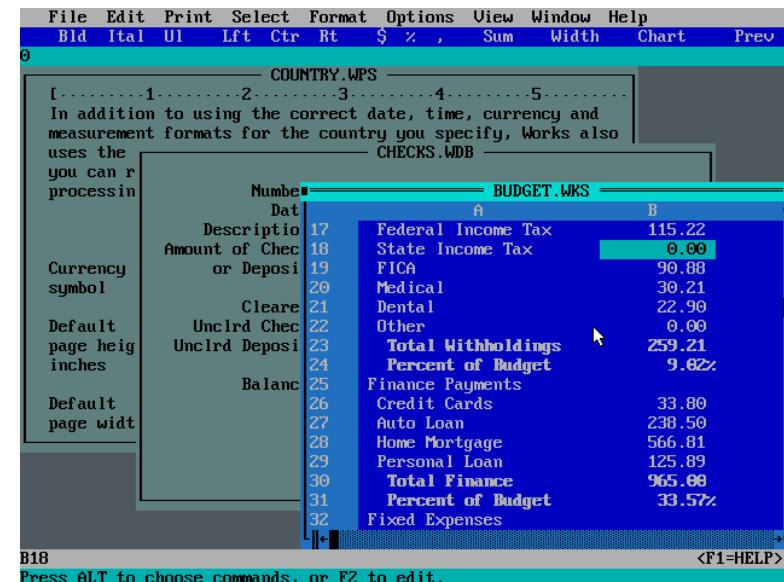
- Trained experts



Norton Commander 1.0 (1986)

# Text-based User Interface – Advantages

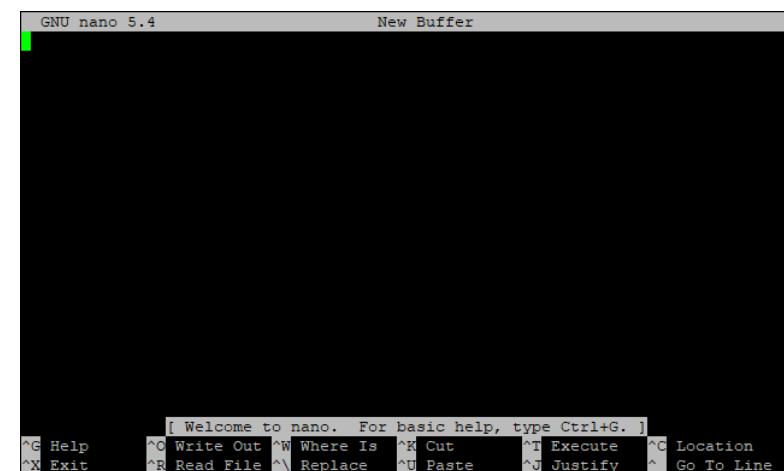
- Explorable
- Possible to provide graphical appearance to a text-based application



Word 3.0 (1986)



Rogue (1980)



GNU nano 5.4 (2022: 7.1)

-



X

# Graphical User Interfaces (GUIs) and the WIMP paradigm

U

CS 349

# Graphical User Interfaces

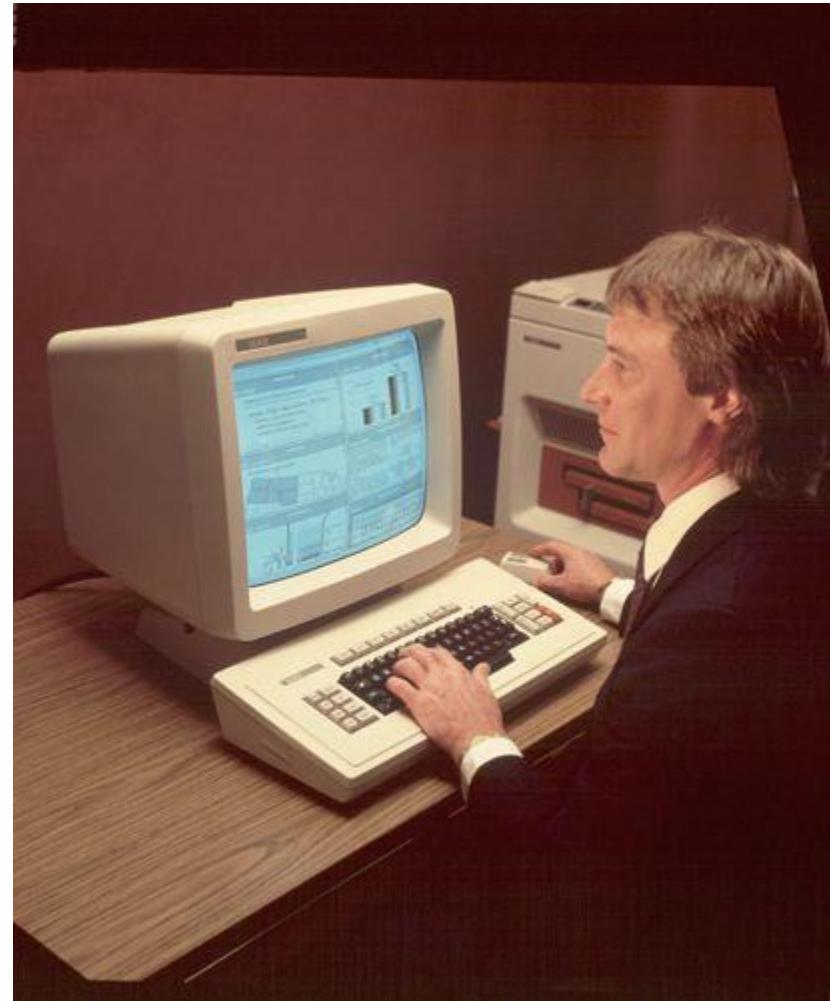
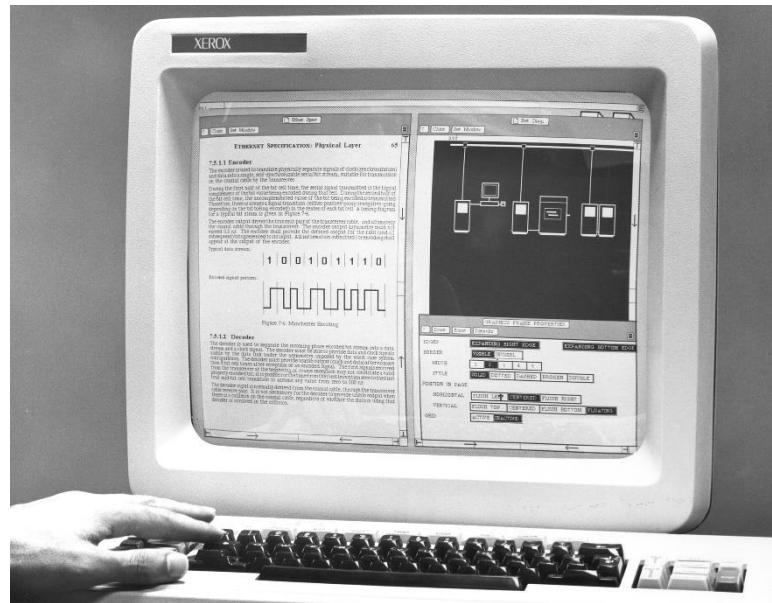
Xerox Alto (1973): first computer with a GUI

- Actions are input via keyboard and mouse
- GUI consisted of windows, icons and menus; files and folders; thus introducing the “desktop” metaphor



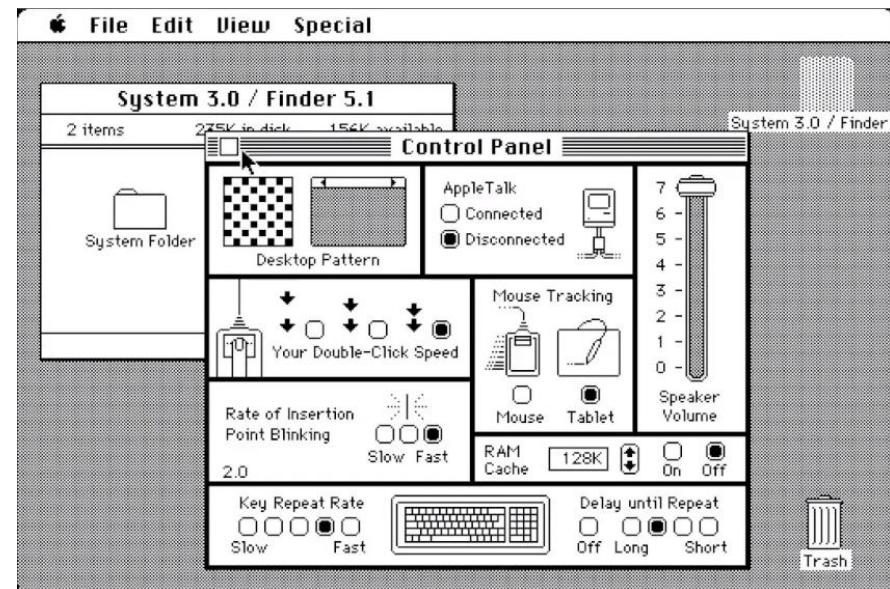
# Graphical User Interfaces

Xerox 8010 Information System (1981): second computer with a GUI



# Graphical User Interfaces

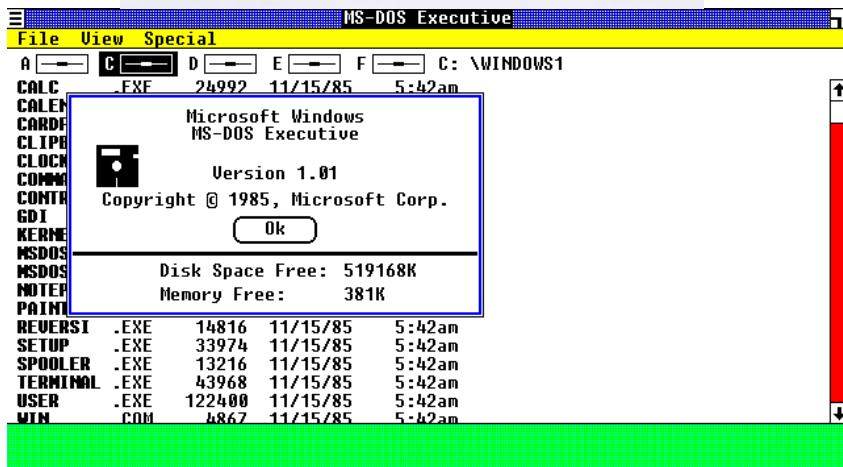
Apple Macintosh (1984): first commercially successful computer with a GUI



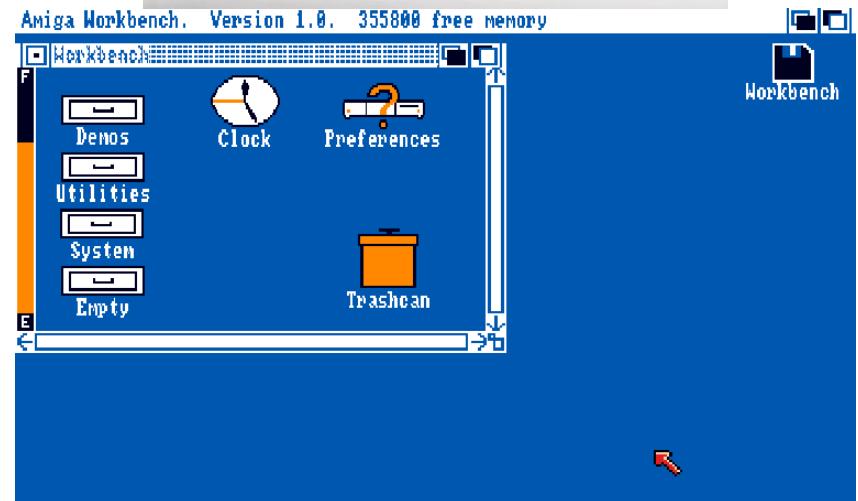
Apple System 3.0 (1986)

# Graphical User Interfaces

Microsoft Windows on IBM PC compatible (here: HP150II, 1985)



AmigaOS on the Amiga 1000 (1985)



# Graphical User Interfaces – Requirements

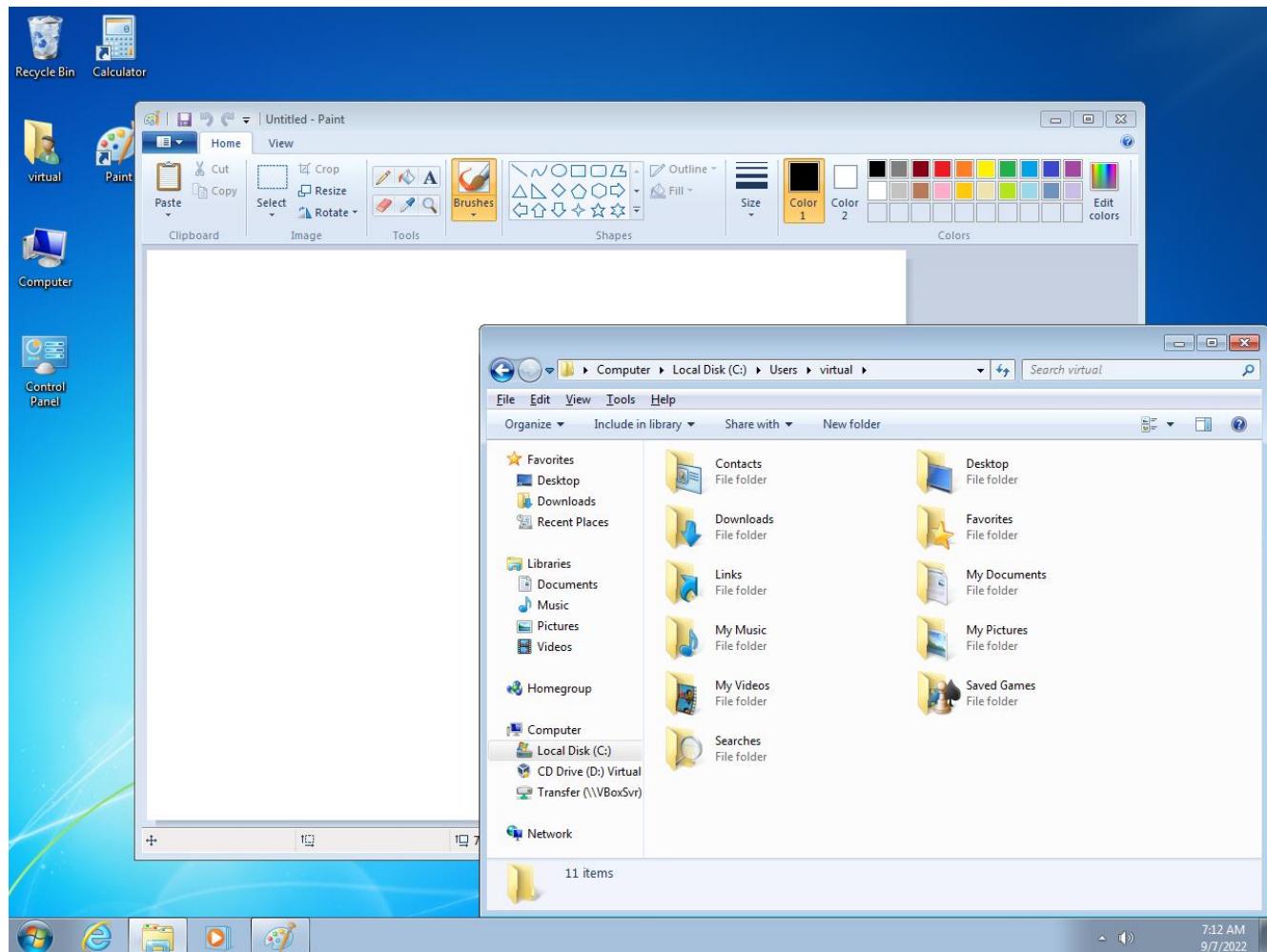
Today, we would consider the following as requirements for a GUI:

- Screen capable of graphics output
- Keyboard (mechanical, touchscreen, etc.)
- Pointing device (mouse, touchpad, graphics tablet, etc.)



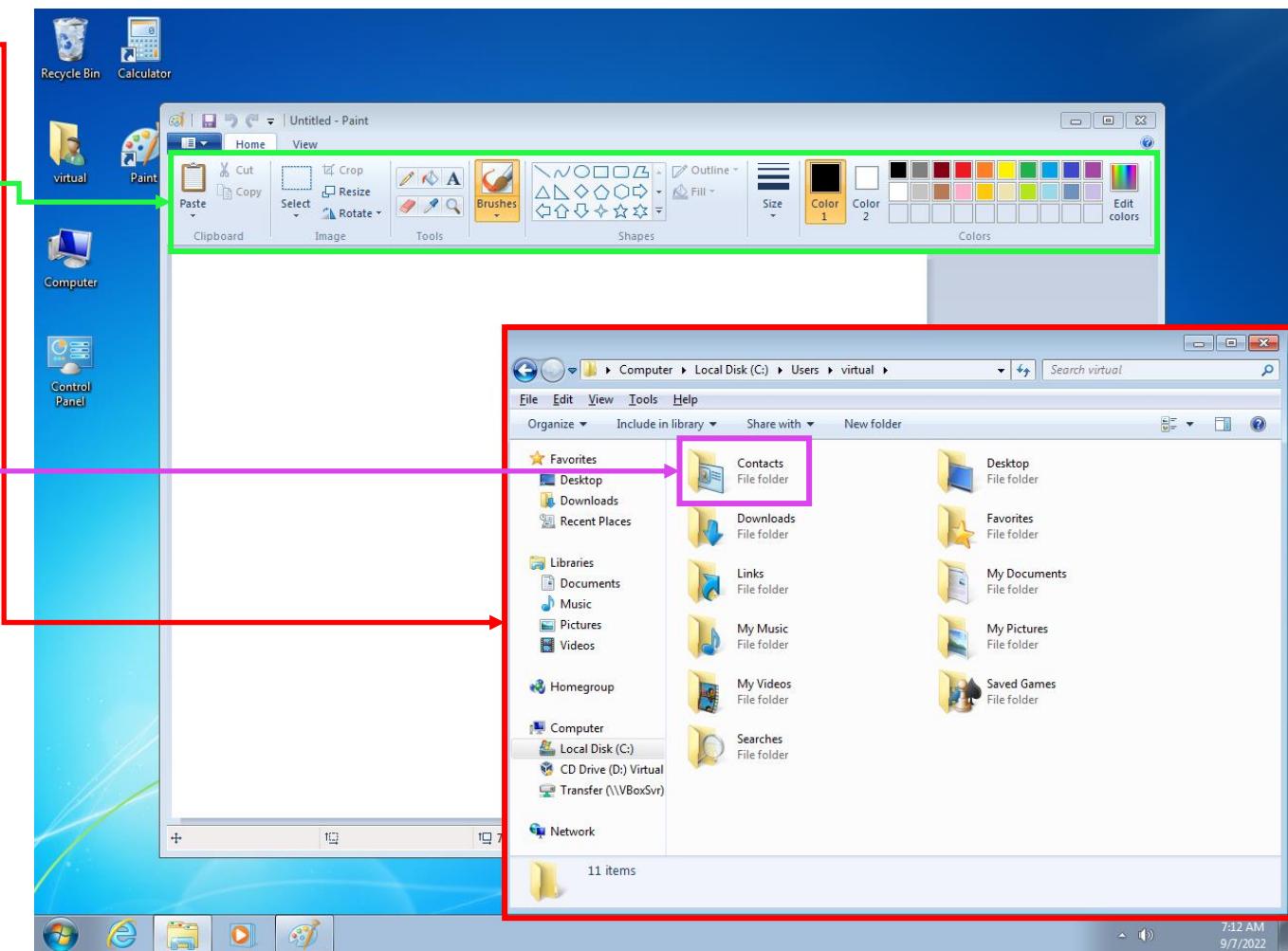
# The WIMP paradigm

Almost all current GUIs follow the “WIMP”-paradigm (Windows, Icons, Menus, Pointers).



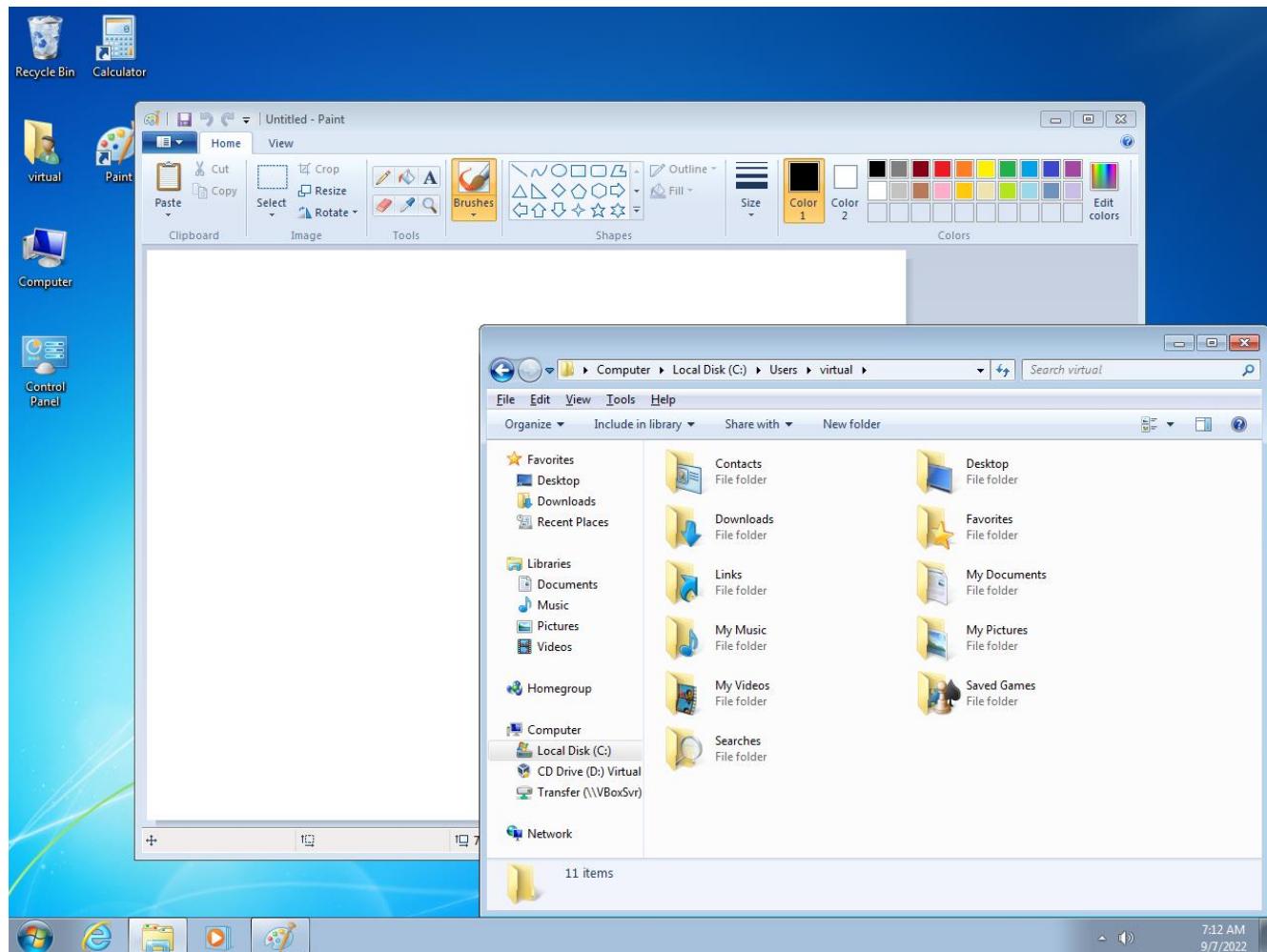
# The WIMP paradigm

- Windows
- Icons
- Menus
- Pointers



# The WIMP paradigm

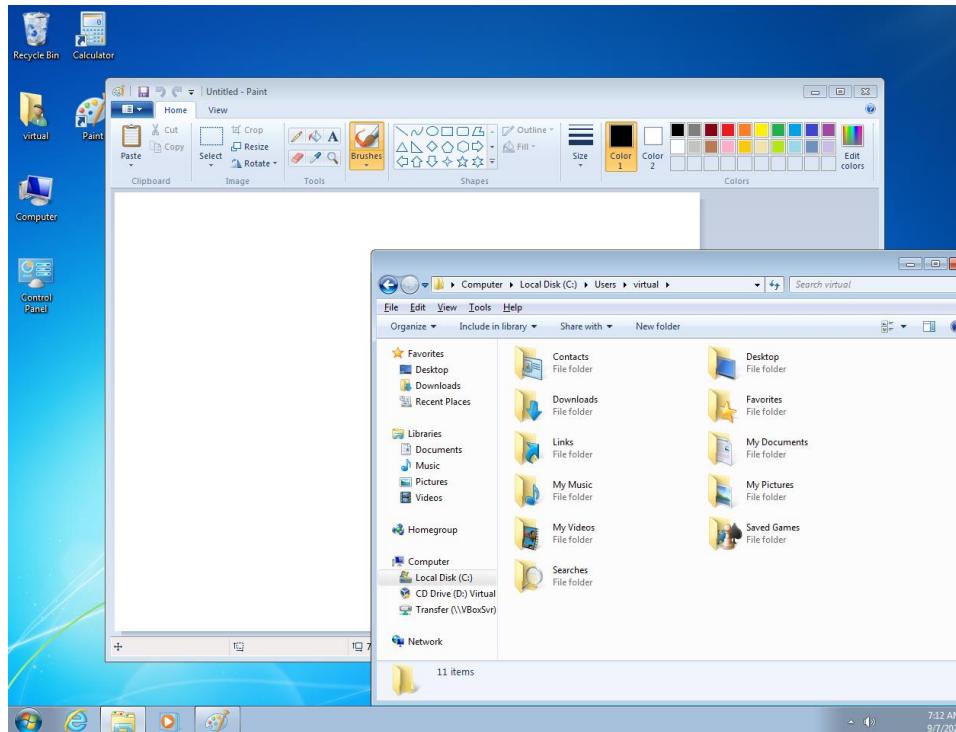
It is usually associated with the “desktop” metaphor and often includes a “Desktop” (or background).



# The WIMP paradigm

In addition to standard GUI capabilities:

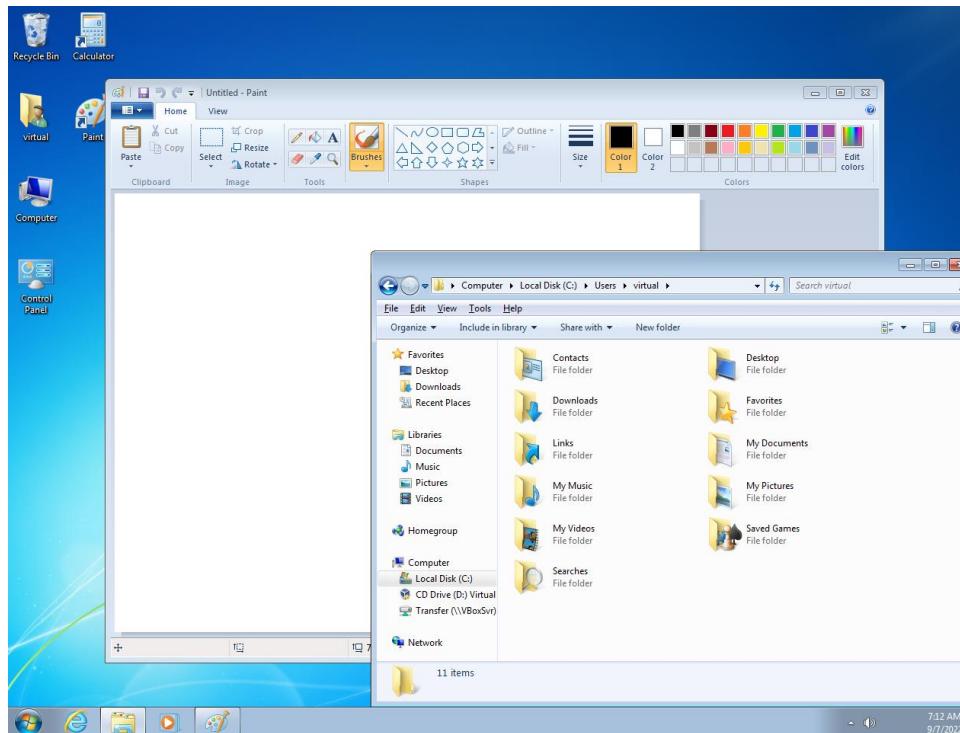
- Each application is isolated within its own window(s).
- System has methods to move, resize, re-order, re-draw windows.
- System supports common presentation of applications / elements.
- Provide common GUI elements for building apps (e.g., buttons).
- Emphasizes recognition of interface features over recall of commands.



# The WIMP paradigm

Windows are independent of one another:

- They do not need to know where they are located on the screen or what other apps are running
- They can be spatially re-arranged to facilitate viewing and manipulating data from multiple sources
- Input and output is directed to a specific window



# End of the Chapter



Please make sure to

- Understand the advantages and disadvantages of CLIs and GUIs
- Remember the components of WIMP



Any further questions?

-



X

# IntelliJ & Kotlin

Software Stack

Kotlin Crash Course

U

CS 349

January 11

U

CS 349

# Software Stack



# Software

**Git**, a distributed version control software.

**IntelliJ IDEA**, an integrated development environment (IDE) for developing computer software written in Java, Kotlin, Groovy, and other JVM-based languages.

- **Gradle**, a build automation tool.
- **Kotlin**, a cross-platform, statically typed, general-purpose programming language with type inference. Kotlin is designed to interoperate fully with Java, and the JVM version of Kotlin's standard library depends on the Java Class Library.
- **JavaFX**, a software platform for creating and delivering desktop applications.



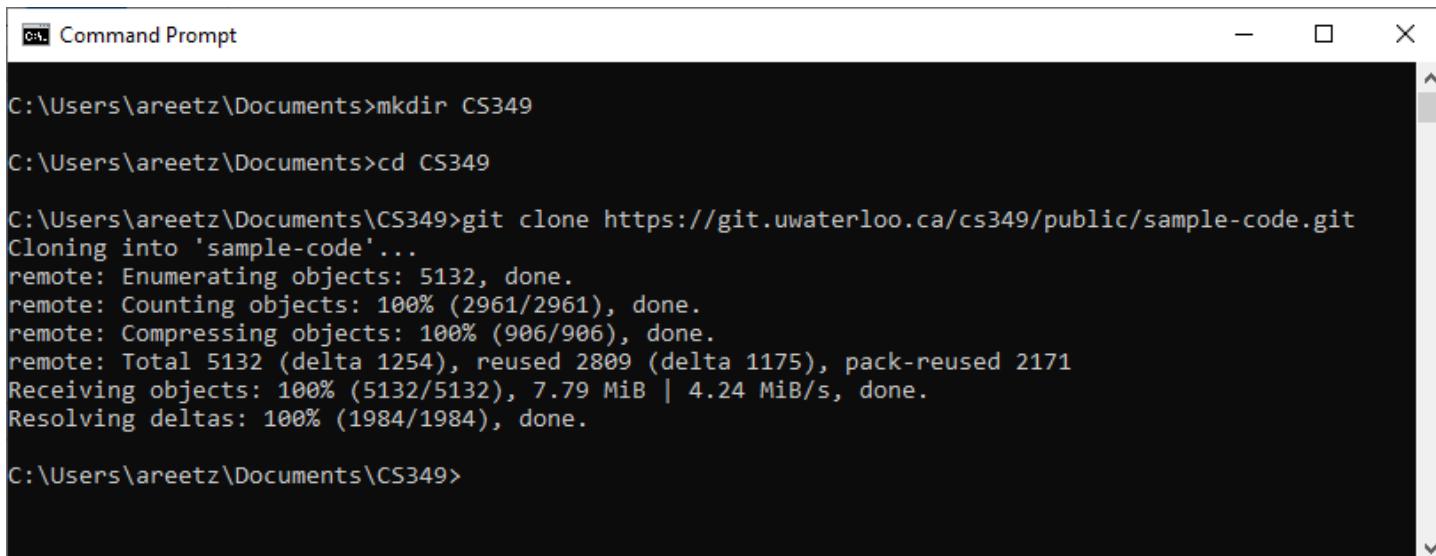
# Git

Install Git from <https://git-scm.com/download>, latest version is 2.39.0.

Install with default options, except preferred text editor.

Create a local copy of cs349-public:

```
git clone https://git.uwaterloo.ca/cs349/public/sample-code.git
```



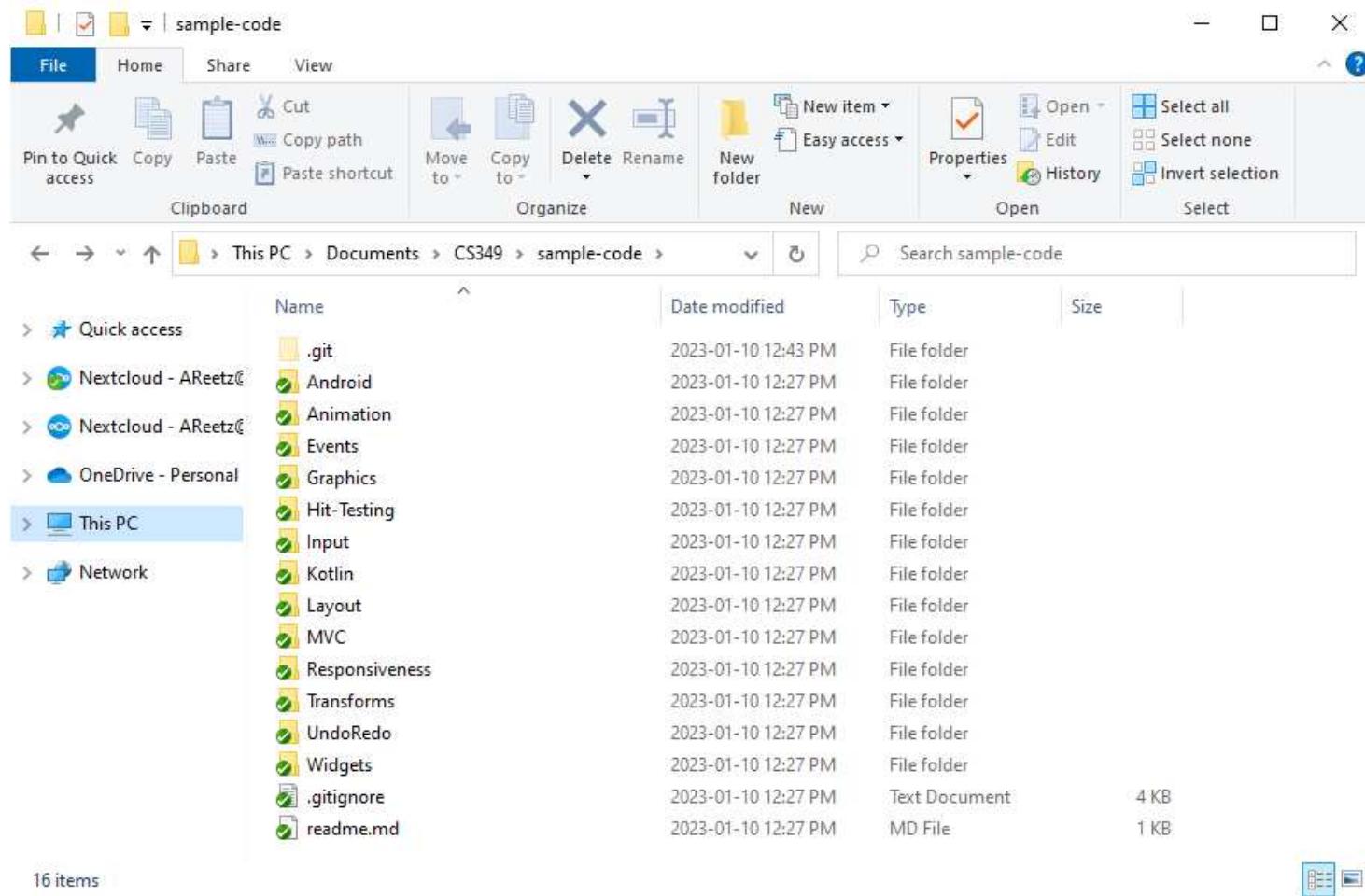
The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a dark background and light-colored text. The user has navigated to their "Documents" folder and created a new directory named "CS349". They then changed into this directory and ran the command "git clone https://git.uwaterloo.ca/cs349/public/sample-code.git". The output of the command shows the progress of cloning a repository, including the enumeration of objects, counting of objects, compression of objects, and the final statistics of the clone operation. The command prompt ends with the path "C:\Users\areetz\Documents\CS349".

```
C:\Users\areetz\Documents>mkdir CS349
C:\Users\areetz\Documents>cd CS349
C:\Users\areetz\Documents\CS349>git clone https://git.uwaterloo.ca/cs349/public/sample-code.git
Cloning into 'sample-code'...
remote: Enumerating objects: 5132, done.
remote: Counting objects: 100% (2961/2961), done.
remote: Compressing objects: 100% (906/906), done.
remote: Total 5132 (delta 1254), reused 2809 (delta 1175), pack-reused 2171
Receiving objects: 100% (5132/5132), 7.79 MiB | 4.24 MiB/s, done.
Resolving deltas: 100% (1984/1984), done.

C:\Users\areetz\Documents\CS349>
```

# Git

Your local copy directory should look like this:



# IntelliJ

Install IntelliJ IDEA Community Edition from  
<https://www.jetbrains.com/idea/download/>, latest version is 2022.3.1.

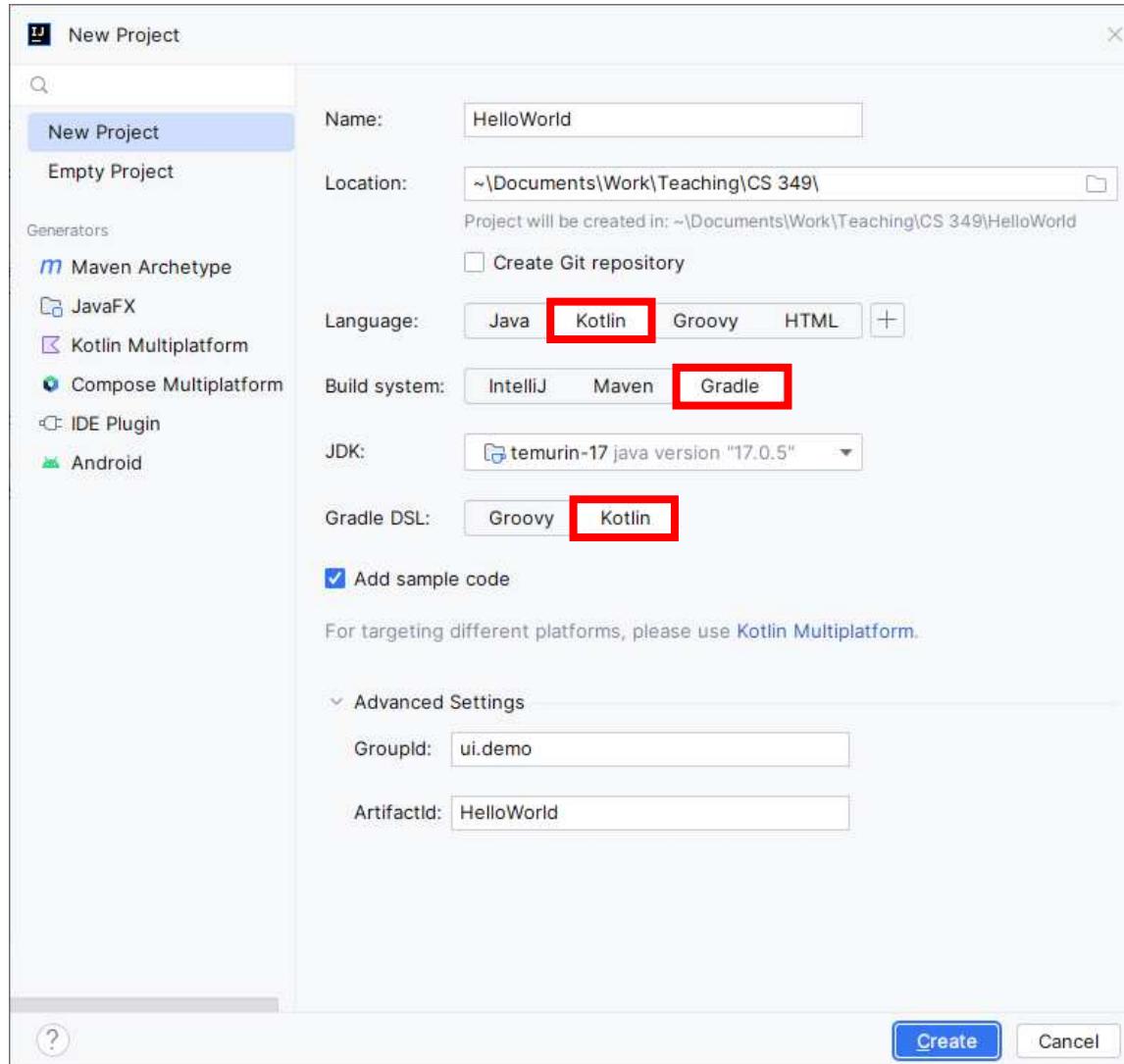
Install with default options.

Once installed, IntelliJ will install Gradle, Kotlin, and the Java SDK for you, either when you create a new project (preferred, as you can choose the JDK version) or when you open an existing project (not preferred, installs JDK version of the project).

- Java SDK: use version 17.0.5. SDK 17 is marked as LTS, so cs349-public is using this version.
- Gradle: use version 7.4.2+ (automatically installed by IntelliJ).
- Kotlin: use version 1.7.21 (automatically installed by IntelliJ).

# IntelliJ & Gradle – New Project

Creating a new project:



# IntelliJ & Gradle – Project Structure

The screenshot shows the IntelliJ IDEA interface with a project named "HelloWorld".

- Source code:** The file `Main.kt` is open in the editor, containing the following Kotlin code:

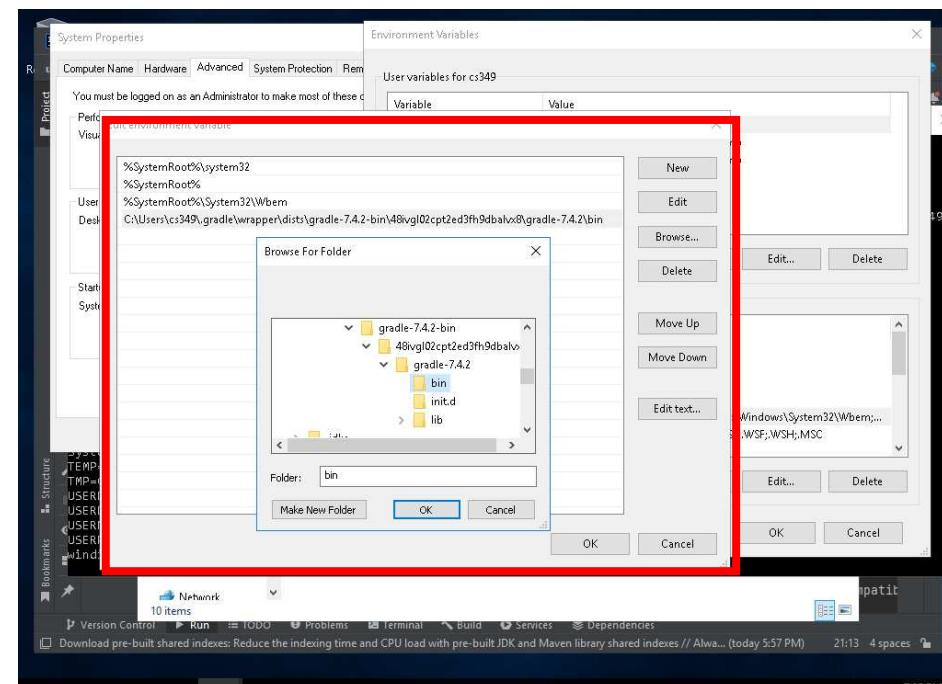
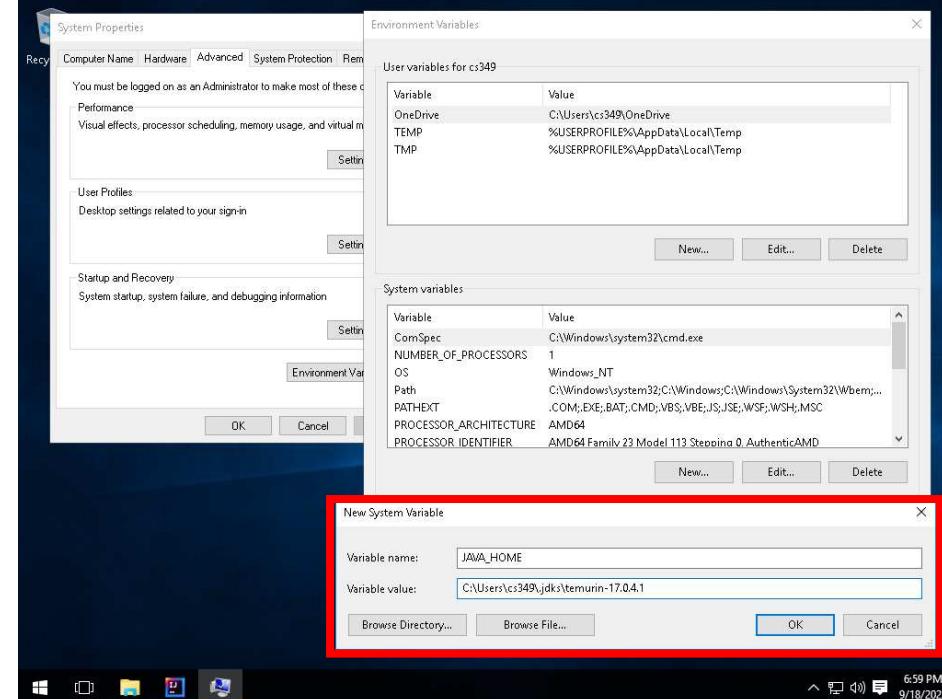
```
fun main(args: Array<String>) {
    println("Hello World!")
    println("Program arguments: ${args.joinToString()}")
}
```
- Configuration files:** The `src` and `test` directories are highlighted with red boxes. The `src/main/kotlin` directory contains `Main.kt`. The `test` directory contains `build.gradle.kts`, `gradle.properties`, `gradlew`, `gradlew.bat`, and `settings.gradle.kts`.
- Gradle window:** A red box highlights the "Grade window" tab in the top right corner.
- Run program:** A red box highlights the "run" task under the "Tasks" section in the Grade window.

# Command Line & Gradle – Run Application

We will run Gradle from the command line, so you might want to test if your assignment project runs from the command line before submitting.

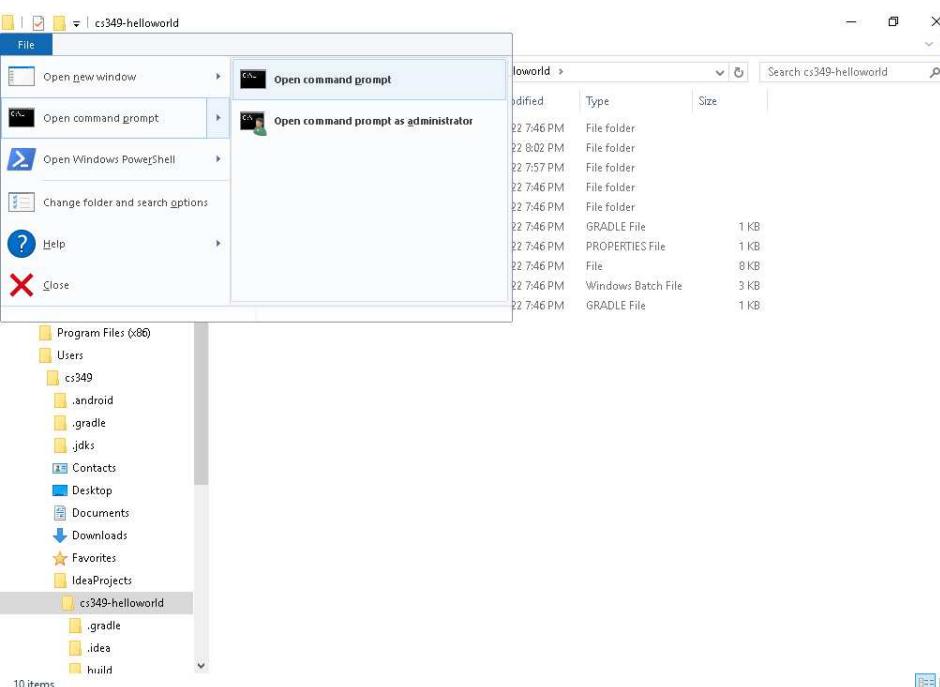
- For Windows

1. Set JAVA\_HOME to %USERPROFILE%\.jdks\...
2. Add %USERPROFILE%\gradle\wrapper\fists\gradle-7.X.X-bin\XXX\gradle-7.X.X\bin\ to PATH.



# Command Line & Gradle – Run Application

Open Explorer to project path,  
then open command prompt:



Execute gradle run:

A screenshot of a terminal window titled 'cmd.exe'. The command 'gradle run' is being executed. The output shows the application running and printing 'Hello World!'. The terminal window also displays deprecation warnings and a link to the Gradle documentation.

```
C:\Users\cs349\IdeaProjects\cs349-helloworld>gradle run
> Task :run
Hello World!
Program arguments:

Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.
You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or plugins.

See https://docs.gradle.org/7.4.2/userguide/command_line_interface.html#sec:command_line_warnings
BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
C:\Users\cs349\IdeaProjects\cs349-helloworld>
```

# Demo

Creating a new Kotlin project:

- <https://student.cs.uwaterloo.ca/~cs349/1231/getting-started/3-kotlin-project/>

Running sample code:

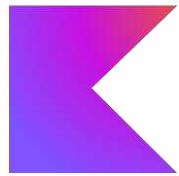
- <https://student.cs.uwaterloo.ca/~cs349/1231/getting-started/5-run-samples/>

Finding your repository

- <https://git.uwaterloo.ca>

# Kotlin Crash Course

U  
CS 349



# Kotlin – Why

There are literally [hundreds of programming languages](#) to choose from.  
How do you pick a language?

There are some non-trivial considerations when picking a language:

- Does it offer the features and capabilities that you require?
- Is it easy to work with? How productive can you be with it?
- How mature is the ecosystem around it? Does it have rich libraries and tooling?

# Kotlin – Features & Strengths

- Kotlin is designed for building applications.
- Class-based, object-oriented, general-purpose language.
- Supports imperative, object-oriented, and functional programming styles.
- Automatic memory management and GC; iterable collections; generics; Broad framework support (graphics, UI).
- Modern features: named arguments, default arguments; NULL handling.
- 100% interoperable with Java source and libraries.
- Multi-platform: Windows, Linux, Mac (JVM or native); Mobile: Android and iOS.

# Kotlin – Other languages

Kotlin can be compiled to native code, or to bytecode (intermediate representation) which is interpreted at runtime.

- **Kotlin/JVM** compiles Kotlin code to JVM bytecode, which can run on any Java virtual machine.
- **Kotlin/Android** compiles Kotlin code to native Android binaries, which leverage native versions of the Java Library and Kotlin standard libraries.
- **Kotlin/Native** compiles Kotlin code to native binaries, which can run without a virtual machine.
- **Kotlin/JS** transpiles (converts) Kotlin to JavaScript.

We will be using Kotlin/JVM and later Kotlin/Android for this course.

- Getting started with Kotlin (especially sections “Basics” and “Concepts”):  
<https://kotlinlang.org/docs/home.html>
- Kotlin playground: <https://play.kotlinlang.org>
- Learn Kotlin by example: <https://play.kotlinlang.org/byExample/overview>

# *Program Entry*

# Program Entry Point

It's tradition to write "Hello World" when learning a new programming language.

Here's the Kotlin version!

```
fun main() {  
    print("Hello ") // print prints parameter as string  
    println("World") // println prints parameter as string + newline  
}  
  
fun main(args: Array<String>) {  
    println(args.contentToString())  
}
```

# Program Entry Point

It's tradition to write "Hello World" when learning a new programming language.

Here's the Kotlin version!

```
fun main() {                                // argument(s):  
    print("Hello ")  
    println("World")                         // prints: Hello World  
}  
  
fun main(args: Array<String>) {           // argument(s): This is CS349  
    println(args.contentToString())           // prints: [This, is, CS349]  
}
```

*Data*

# Data

There are two flavours of data:

- Mutable data

```
var i: Int = 349
```

- Immutable data

```
val i: Int = 349
```

1. Mutability qualifier<sup>†</sup>: **var**
2. Identifier: **i**
3. : Type<sup>‡</sup>: **Int**
4. Assignment operator<sup>†</sup>: **=**
5. Value: **349**

<sup>†</sup>: see below

<sup>‡</sup>: optional

# Immutable Data – Values

**Immutable** data (*value*) cannot be reassigned after initialization. We use the `val` keyword to denote immutable data (`val` for value).

```
val i: Int = 349 // declaration & assignment  
  
i = 449          // compile-time error: Val cannot be reassigned  
  
val j: Int       // declaration only, type is required  
j = 349          // deferred assignment  
j = 449          // compile-time error: Val cannot be reassigned  
j = "449"        // compile-time error: Val cannot be reassigned &  
                  // type mismatch
```

# Data – Variables vs Values

Data should be immutable unless they absolutely need to change!

This follows best-practices in other languages, e.g., `final` in Java, `const` in C++.

# Data – Typing

Kotlin is *strongly static* typed.

- Static: type is verified from syntax (not data value)
- Strong: type is verified at compile time (not run time)

```
val i: Int = 349 // declaration & assignment:
```

```
val i = 349      // type is optional (inferred: Int)
```

```
val i: Int       // declaration only: type is required
```

# *Functions*

# Functions

```
fun meaning(): Int {  
    return 42  
}  
  
fun sum(a: Int, b: Int): Unit {  
    println("$a + $b = ${a + b}")  
}
```

1. Function keyword: **fun**
2. Identifier: **meaning**
3. (Parameter list):
4. : Return type<sup>‡</sup>: **Int**
5. {Function body}

<sup>‡</sup>: optional

# Functions – Return Type

The Unit object is a type with only one value. This type corresponds to void in many other programming languages.

```
fun sum(a: Int, b: Int): Unit { ... }
```

```
fun sum(a: Int, b: Int) { ... } // No return type defaults to Unit
```

Kotlin does not infer return types:

```
fun add1(i: Int) {  
    return i + 1 // Type mismatch. Required: Unit Found: Int  
}
```

# Functions – Calling

```
fun addn(i: Int, n: Int = 1): Int { // n has default value 1
    print("add$n($i) ")           // string template†
    return i + n
}

println(addn(5, 3))             // prints: add3(5) 8

println(addn(5))               // prints: add1(5) 6: uses default value for n

println(addn(n = 3, i = 5)) // prints: add3(5) 8: naming parameters
```

# Functions – Variable-length Argument Lists

Finally, we can have a list of undefined length (i.e., evaluated at runtime).

```
fun sum(vararg numbers: Int): Int {  
    var sum: Int = 0  
    for(number in numbers) {  
        sum += number  
    }  
    return sum  
}
```

```
println(sum(1))      // prints: 1  
println(sum(1,2,3)) // prints: 6
```

# Functions – Lambda

In Kotlin, functions are first-class language elements.

Functions can be stored as data:

```
var addn: (Int, Int) -> Int = {  
    i: Int, n: Int ->  
    print("add$n($i) ")  
    i + n  
}
```

1. Mutability qualifier: **var**
2. Identifier: **addn**
3. : Type: **(Int, Int) -> Int**
4. Assignment operator: **=**
5. Value: **{**
  1. **fun**
  2. **Identifier**
  3. (Parameter list): **i: Int, n: Int**
  4. ~~: Return type~~: inferred instead
  5. **-> Function body****}**

# Functions – Lambda

Functions can be passed as parameter from data:

```
var addn = {           // type (function signature) inferred from
    i: Int, n: Int -> // parameter list ...
    print("add$n($i) ")
    i + n             // ... and last statement in body (return type)
}

fun apply3(i: Int, func: (Int, Int) -> Int) : Int {
    return func(i, 3)
}

println(apply3(5, addn)) // prints: add3(5) 8
```

# Functions – Lambda

Functions can be passed as parameter from data:

```
var addn = {  
    i: Int, n: Int -> // type (function signature) inferred from  
    print("add$n($i) ")  
    i + n // ... and last statement in body (return type)  
}  
  
fun apply3(i: Int, func: (Int, Int) -> Int) : Int {  
    return func(i, 3)  
}  
  
println(apply3(5, addn)) // prints: add3(5) 8
```

# Functions – Lambda

Functions can be passed as parameter from an inline definition as long as the function is the last parameter in the parameter list of the higher-order function:

```
fun apply3(i: Int, func: (Int, Int) -> Int) : Int {  
    return func(i, 3)  
}  
  
println(apply3(5) { i: Int, m: Int ->  
    print("$i mod $m is ")  
    i % m  
})
```

# Functions – Lambda

If a function only has one parameter, it does not need to be declared, and -> can be omitted. The parameter will be implicitly declared under the name `it`:

```
fun printif(n: Int, pred: (Int) -> Boolean) {  
    if (pred(n)) {  
        println(n)  
    }  
}  
  
printif(349) { n: Int -> n % 2 == 1 } // prints: 349  
  
printif(349) { it % 2 == 1 }           // prints: 349
```

# Functions – Lambda

Functions can be passed as parameter from inline definition:

```
fun printif(n: Int, pred: (Int) -> Boolean) {  
    if (pred(n)) {  
        println(n)  
    }  
}  
  
(0..6).foreach { printif(it) { it % 2 == 1 } } // prints: 1, 3, 5
```

# Operators

Precedence	Title	Symbols	Example
Highest	Postfix	<code>++, --, ., ?., ?, !!</code>	<code>i++ obj.func() obj?.field obj!!</code>
	Prefix	<code>-, +, ++, --, !, label</code>	<code>-i --i !b</code>
	Type RHS	<code>: , as, as?</code>	<code>var i: Int objA as TypeB</code>
	Multiplicative	<code>*, /, %</code>	<code>i * j</code>
	Additive	<code>+, -</code>	<code>i + j</code>
	Range	<code>..</code>	<code>for (i in 1 .. 10)</code>
	Infix function	<code>simpleIdentifier</code>	<code>val b1 = b2 or b3 var lst = 1 until 10</code>
	Elvis	<code>?:</code>	<code>val i = obj?.field ?: -1</code>
	Named checks	<code>in, !in, is, !is</code>	<code>5 in lst objA is TypeA</code>
	Comparison	<code>&lt;, &gt;, &lt;=, &gt;=</code>	<code>a &lt; b a &gt;= b</code>
	Equality	<code>==, !=, ===, !==</code>	<code>a != b objA === objB</code>
	Conjunction	<code>&amp;&amp;</code>	<code>b1 &amp;&amp; b2</code>
	Disjunction	<code>  </code>	<code>b1    b2</code>
	Spread operator	<code>*</code>	
Lowest	Assignment	<code>=, +=, -=, *=, /=, %=</code>	<code>var i = 349 i += j</code>

# Standard Types

Category	Type	Range	Examples & Conversions
Numbers	Byte	-128 to 127	127 128.toByte() // yields: -128
	UByte	0 to 255	
	Short	-32768 to 32767	349.99.toInt().toShort() // yields: 349
	UShort	0 to 65535	
Floating point	Int	- $2^{31}$ to $2^{31}-1$	1000003 0x349A // yields 13466
	UInt	0 to $2^{32}-1$	
Text	Long	- $2^{63}$ to $2^{63}-1$	349L 349.toLong()
	ULong	0 to $2^{64}-1$	
Other	Float	23b sign, 8b exp	349.99f 349.99.toFloat()
	Double	52b sign, 11b exp	349.99 1E6 // yields: 1000000.0 3.49E-5 // yields: 0.000349
	Char		'a' '\t' '\n' '\"' '\"' '\\' '\?' \u03BB // yields '\lambda'
	String		"Hello \"World\"\\?\n" """\ Hello 'World'? """\.trimMargin() // yields "Hello \"World\"\\?\n"
	Boolean	false, true	"true".toBoolean()

-



X

# IntelliJ & Kotlin

Software Stack

Kotlin Crash Course

U

CS 349

January 11

U

CS 349

# Software Stack



# Software

**Git**, a distributed version control software.

**IntelliJ IDEA**, an integrated development environment (IDE) for developing computer software written in Java, Kotlin, Groovy, and other JVM-based languages.

- **Gradle**, a build automation tool.
- **Kotlin**, a cross-platform, statically typed, general-purpose programming language with type inference. Kotlin is designed to interoperate fully with Java, and the JVM version of Kotlin's standard library depends on the Java Class Library.
- **JavaFX**, a software platform for creating and delivering desktop applications.



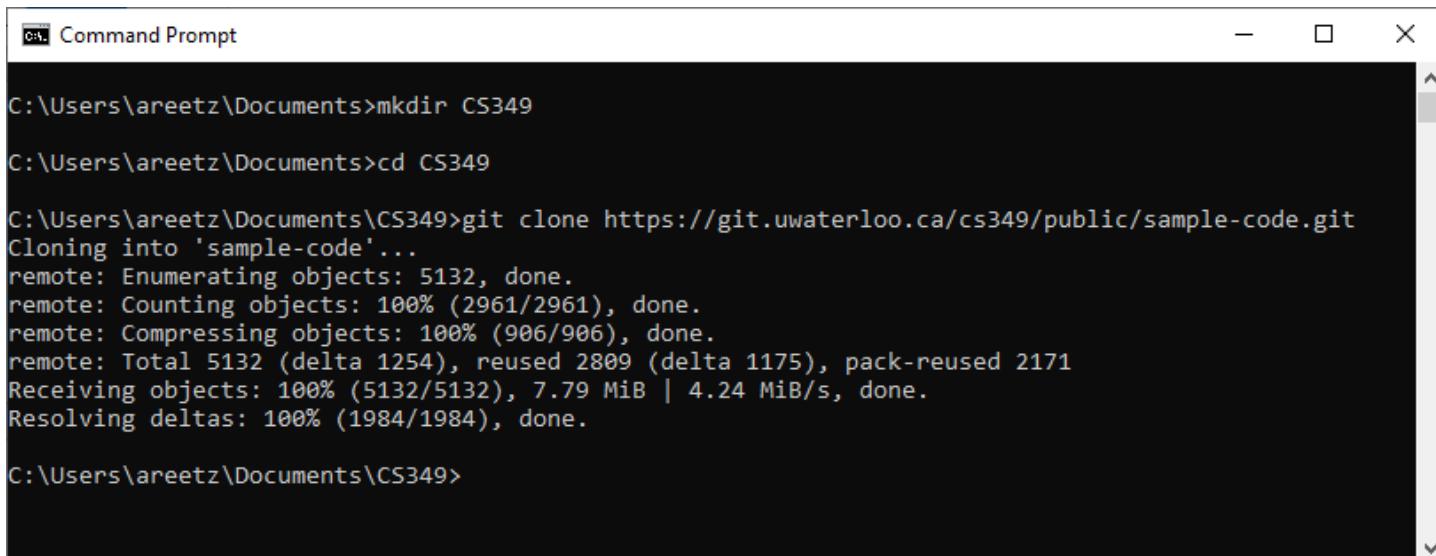
# Git

Install Git from <https://git-scm.com/download>, latest version is 2.39.0.

Install with default options, except preferred text editor.

Create a local copy of cs349-public:

```
git clone https://git.uwaterloo.ca/cs349/public/sample-code.git
```



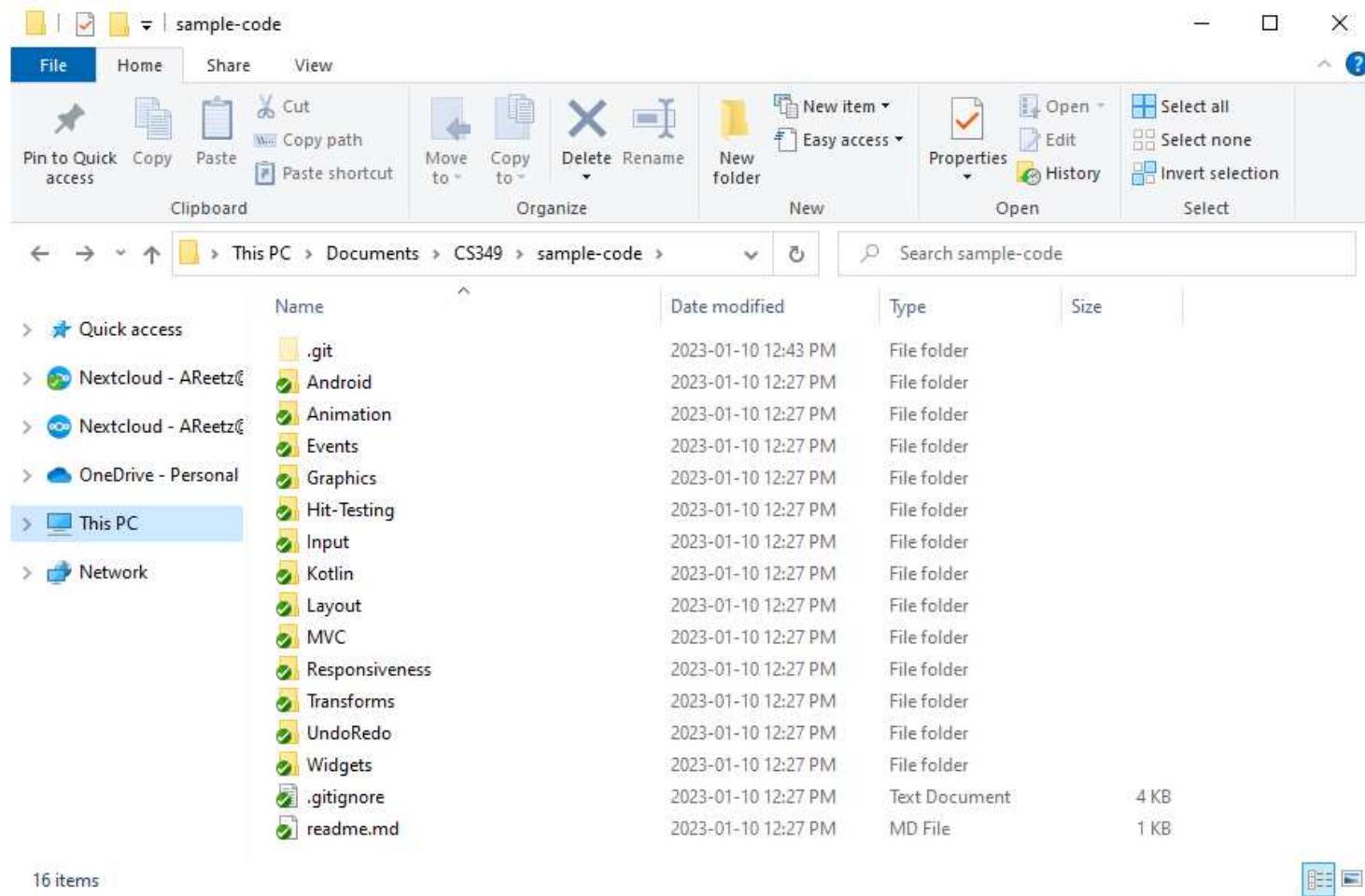
The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a dark background and light-colored text. The user has navigated to their "Documents" folder and created a new directory named "CS349". They then changed into this directory and ran the command "git clone https://git.uwaterloo.ca/cs349/public/sample-code.git". The output of the command shows the progress of cloning a repository with 5132 objects, including object enumeration, counting, compressing, and receiving, followed by delta compression and resolution steps. The process is completed successfully.

```
C:\Users\areetz\Documents>mkdir CS349
C:\Users\areetz\Documents>cd CS349
C:\Users\areetz\Documents\CS349>git clone https://git.uwaterloo.ca/cs349/public/sample-code.git
Cloning into 'sample-code'...
remote: Enumerating objects: 5132, done.
remote: Counting objects: 100% (2961/2961), done.
remote: Compressing objects: 100% (906/906), done.
remote: Total 5132 (delta 1254), reused 2809 (delta 1175), pack-reused 2171
Receiving objects: 100% (5132/5132), 7.79 MiB | 4.24 MiB/s, done.
Resolving deltas: 100% (1984/1984), done.

C:\Users\areetz\Documents\CS349>
```

# Git

Your local copy directory should look like this:



# IntelliJ

Install IntelliJ IDEA Community Edition from  
<https://www.jetbrains.com/idea/download/>, latest version is 2022.3.1.

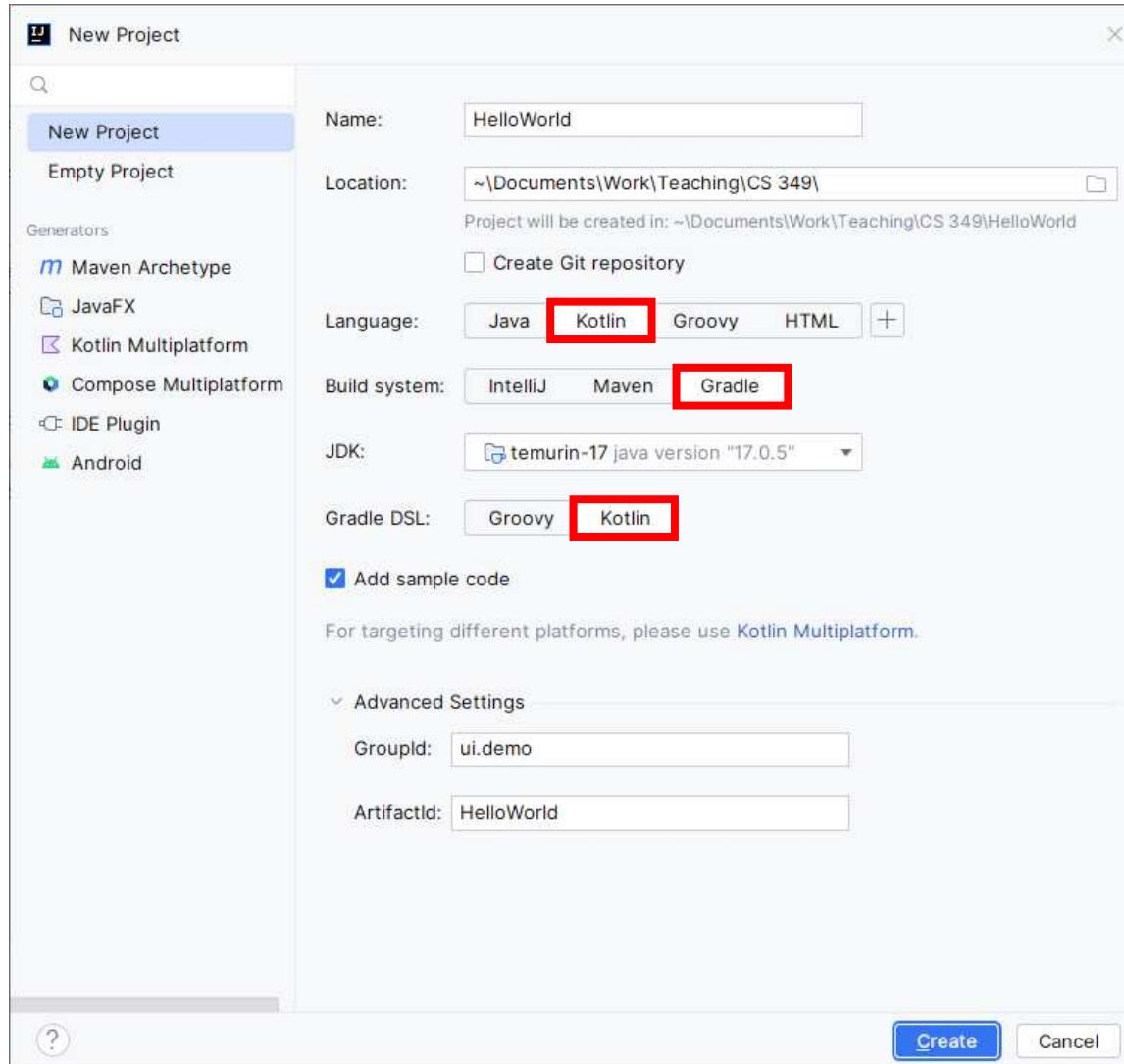
Install with default options.

Once installed, IntelliJ will install Gradle, Kotlin, and the Java SDK for you, either when you create a new project (preferred, as you can choose the JDK version) or when you open an existing project (not preferred, installs JDK version of the project).

- Java SDK: use version 17.0.5. SDK 17 is marked as LTS, so cs349-public is using this version.
- Gradle: use version 7.4.2+ (automatically installed by IntelliJ).
- Kotlin: use version 1.7.21 (automatically installed by IntelliJ).

# IntelliJ & Gradle – New Project

Creating a new project:



# IntelliJ & Gradle – Project Structure

The screenshot shows the IntelliJ IDEA interface with a project named "HelloWorld".

- Source code:** The file `Main.kt` is open in the editor, containing the following Kotlin code:

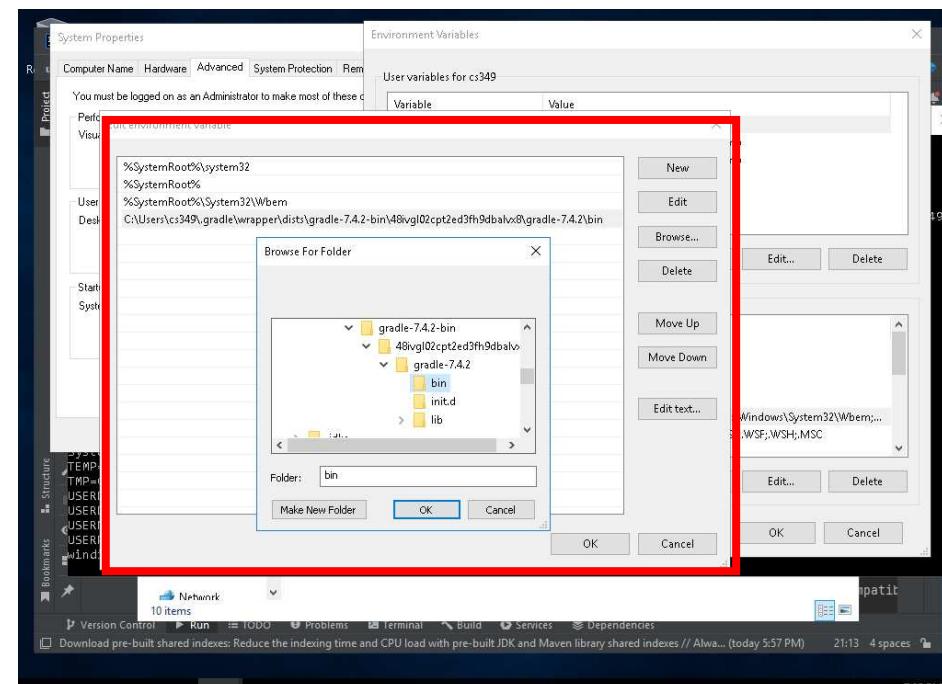
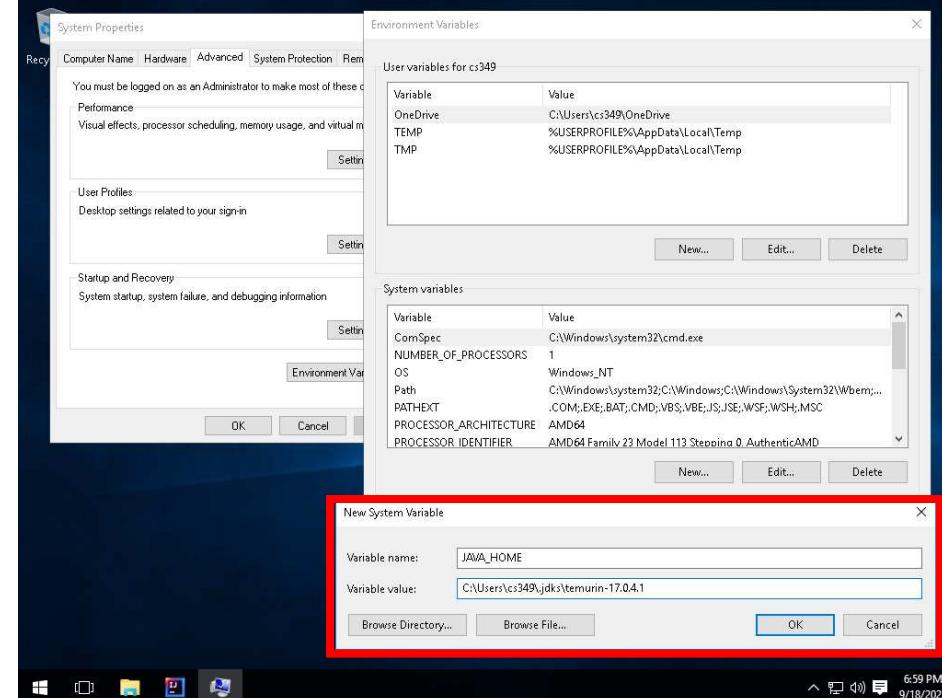
```
fun main(args: Array<String>) {
    println("Hello World!")
    println("Program arguments: ${args.joinToString()}")
}
```
- Configuration files:** The `src` and `test` directories are highlighted with red boxes. The `src/main/kotlin` directory contains `Main.kt`. The `test` directory contains `build.gradle.kts`, `gradle.properties`, `gradlew`, `gradlew.bat`, and `settings.gradle.kts`.
- Gradle window:** A red box highlights the "Grade window" tab in the top right corner.
- Run program:** A red box highlights the "run" task under the "Tasks" section in the Grade window.

# Command Line & Gradle – Run Application

We will run Gradle from the command line, so you might want to test if your assignment project runs from the command line before submitting.

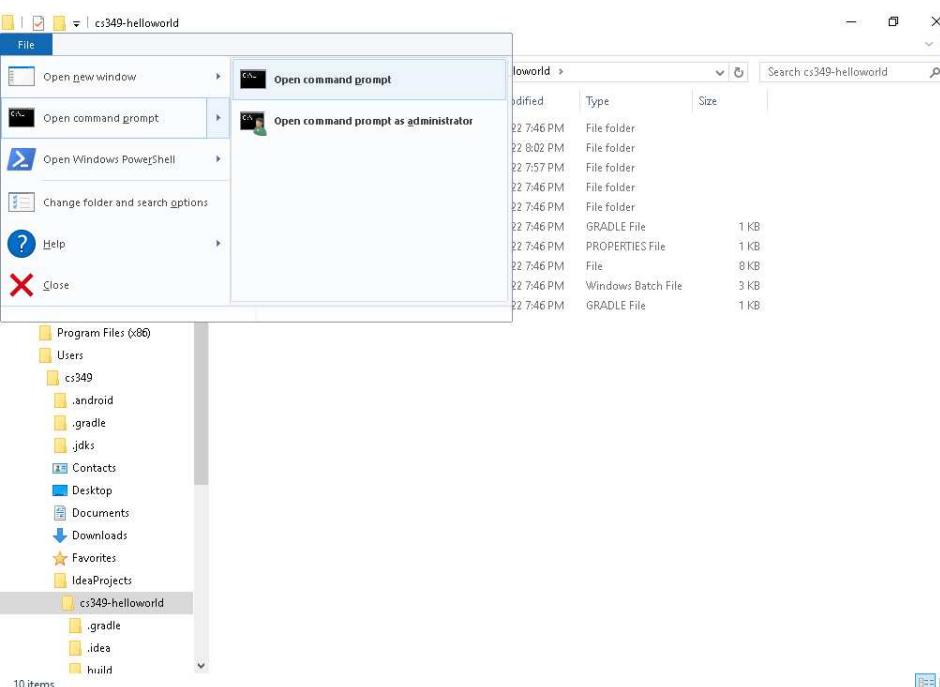
- For Windows

1. Set JAVA\_HOME to %USERPROFILE%\.jdks\...
2. Add %USERPROFILE%\gradle\wrapper\fists\gradle-7.X.X-bin\XXX\gradle-7.X.X\bin\ to PATH.



# Command Line & Gradle – Run Application

Open Explorer to project path,  
then open command prompt:



Execute gradle run:

The screenshot shows a terminal window with the command `gradle run` being executed. The output shows the application running and printing "Hello World!". It also provides information about deprecated features and a link for more details. The terminal window is titled "cmd.exe".

```
C:\Users\cs349\IdeaProjects\cs349-helloworld>gradle run
> Task :run
Hello World!
Program arguments:

Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.
You can use '--warning-mode all' to show the individual deprecation warnings and determine if they come from your own scripts or plugins.

See https://docs.gradle.org/7.4.2/userguide/command_line_interface.html#sec:command_line_warnings

BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
C:\Users\cs349\IdeaProjects\cs349-helloworld>
```

# Demo

Creating a new Kotlin project:

- <https://student.cs.uwaterloo.ca/~cs349/1231/getting-started/3-kotlin-project/>

Running sample code:

- <https://student.cs.uwaterloo.ca/~cs349/1231/getting-started/5-run-samples/>

Finding your repository

- <https://git.uwaterloo.ca>

# Kotlin Crash Course

U  
CS 349



# Kotlin – Why

There are literally [hundreds of programming languages](#) to choose from.  
How do you pick a language?

There are some non-trivial considerations when picking a language:

- Does it offer the features and capabilities that you require?
- Is it easy to work with? How productive can you be with it?
- How mature is the ecosystem around it? Does it have rich libraries and tooling?

# Kotlin – Features & Strengths

- Kotlin is designed for building applications.
- Class-based, object-oriented, general-purpose language.
- Supports imperative, object-oriented, and functional programming styles.
- Automatic memory management and GC; iterable collections; generics; Broad framework support (graphics, UI).
- Modern features: named arguments, default arguments; NULL handling.
- 100% interoperable with Java source and libraries.
- Multi-platform: Windows, Linux, Mac (JVM or native); Mobile: Android and iOS.

# Kotlin – Other languages

Kotlin can be compiled to native code, or to bytecode (intermediate representation) which is interpreted at runtime.

- **Kotlin/JVM** compiles Kotlin code to JVM bytecode, which can run on any Java virtual machine.
- **Kotlin/Android** compiles Kotlin code to native Android binaries, which leverage native versions of the Java Library and Kotlin standard libraries.
- **Kotlin/Native** compiles Kotlin code to native binaries, which can run without a virtual machine.
- **Kotlin/JS** transpiles (converts) Kotlin to JavaScript.

We will be using Kotlin/JVM and later Kotlin/Android for this course.

- Getting started with Kotlin (especially sections “Basics” and “Concepts”):  
<https://kotlinlang.org/docs/home.html>
- Kotlin playground: <https://play.kotlinlang.org>
- Learn Kotlin by example: <https://play.kotlinlang.org/byExample/overview>

# *Program Entry*

# Program Entry Point

It's tradition to write "Hello World" when learning a new programming language.

Here's the Kotlin version!

```
fun main() {  
    print("Hello ") // print prints parameter as string  
    println("World") // println prints parameter as string + newline  
}  
  
fun main(args: Array<String>) {  
    println(args.contentToString())  
}
```

# Program Entry Point

It's tradition to write "Hello World" when learning a new programming language.

Here's the Kotlin version!

```
fun main() {                                // argument(s):  
    print("Hello ")  
    println("World")                         // prints: Hello World  
}  
  
fun main(args: Array<String>) {           // argument(s): This is CS349  
    println(args.contentToString())           // prints: [This, is, CS349]  
}
```

*Data*

# Data

There are two flavours of data:

- Mutable data

```
var i: Int = 349
```

- Immutable data

```
val i: Int = 349
```

1. Mutability qualifier<sup>†</sup>: **var**
2. Identifier: **i**
3. : Type<sup>‡</sup>: **Int**
4. Assignment operator<sup>†</sup>: **=**
5. Value: **349**

<sup>†</sup>: see below

<sup>‡</sup>: optional

# Immutable Data – Values

**Immutable** data (*value*) cannot be reassigned after initialization. We use the `val` keyword to denote immutable data (`val` for value).

```
val i: Int = 349 // declaration & assignment  
  
i = 449          // compile-time error: Val cannot be reassigned  
  
val j: Int       // declaration only, type is required  
j = 349          // deferred assignment  
j = 449          // compile-time error: Val cannot be reassigned  
j = "449"        // compile-time error: Val cannot be reassigned &  
                  // type mismatch
```

# Data – Variables vs Values

Data should be immutable unless they absolutely need to change!

This follows best-practices in other languages, e.g., `final` in Java, `const` in C++.

# Data – Typing

Kotlin is *strongly static* typed.

- Static: type is verified from syntax (not data value)
- Strong: type is verified at compile time (not run time)

```
val i: Int = 349 // declaration & assignment:
```

```
val i = 349      // type is optional (inferred: Int)
```

```
val i: Int       // declaration only: type is required
```

# *Functions*

# Functions

```
fun meaning(): Int {  
    return 42  
}  
  
fun sum(a: Int, b: Int): Unit {  
    println("$a + $b = ${a + b}")  
}
```

1. Function keyword: **fun**
2. Identifier: **meaning**
3. (Parameter list):
4. : Return type<sup>‡</sup>: **Int**
5. {Function body}

<sup>‡</sup>: optional

# Functions – Return Type

The Unit object is a type with only one value. This type corresponds to void in many other programming languages.

```
fun sum(a: Int, b: Int): Unit { ... }
```

```
fun sum(a: Int, b: Int) { ... } // No return type defaults to Unit
```

Kotlin does not infer return types:

```
fun add1(i: Int) {  
    return i + 1 // Type mismatch. Required: Unit Found: Int  
}
```

# Functions – Calling

```
fun addn(i: Int, n: Int = 1): Int { // n has default value 1
    print("add$n($i) ")           // string template†
    return i + n
}

println(addn(5, 3))             // prints: add3(5) 8

println(addn(5))               // prints: add1(5) 6: uses default value for n

println(addn(n = 3, i = 5))    // prints: add3(5) 8: naming parameters
```

# Functions – Variable-length Argument Lists

Finally, we can have a list of undefined length (i.e., evaluated at runtime).

```
fun sum(vararg numbers: Int): Int {  
    var sum: Int = 0  
    for(number in numbers) {  
        sum += number  
    }  
    return sum  
}
```

```
println(sum(1))      // prints: 1  
println(sum(1,2,3)) // prints: 6
```

# Functions – Lambda

In Kotlin, functions are first-class language elements.

Functions can be stored as data:

```
var addn: (Int, Int) -> Int = {  
    i: Int, n: Int ->  
    print("add$n($i) ")  
    i + n  
}
```

1. Mutability qualifier: **var**
2. Identifier: **addn**
3. : Type: **(Int, Int) -> Int**
4. Assignment operator: **=**
5. Value: **{**
  1. **fun**
  2. **Identifier**
  3. (Parameter list): **i: Int, n: Int**
  4. ~~: Return type~~: inferred instead
  5. **-> Function body****}**

# Functions – Lambda

Functions can be passed as parameter from data:

```
var addn = {           // type (function signature) inferred from
    i: Int, n: Int -> // parameter list ...
    print("add$n($i) ")
    i + n             // ... and last statement in body (return type)
}

fun apply3(i: Int, func: (Int, Int) -> Int) : Int {
    return func(i, 3)
}

println(apply3(5, addn)) // prints: add3(5) 8
```

# Functions – Lambda

Functions can be passed as parameter from data:

```
var addn = {  
    i: Int, n: Int -> // type (function signature) inferred from  
    print("add$n($i) ")  
    i + n // ... and last statement in body (return type)  
}  
  
fun apply3(i: Int, func: (Int, Int) -> Int) : Int {  
    return func(i, 3)  
}  
  
println(apply3(5, addn)) // prints: add3(5) 8
```

# Functions – Lambda

Functions can be passed as parameter from an inline definition as long as the function is the last parameter in the parameter list of the higher-order function:

```
fun apply3(i: Int, func: (Int, Int) -> Int) : Int {  
    return func(i, 3)  
}  
  
println(apply3(5) { i: Int, m: Int ->  
    print("$i mod $m is ")  
    i % m  
})
```

# Functions – Lambda

If a function only has one parameter, it does not need to be declared, and -> can be omitted. The parameter will be implicitly declared under the name `it`:

```
fun printif(n: Int, pred: (Int) -> Boolean) {  
    if (pred(n)) {  
        println(n)  
    }  
}  
  
printif(349) { n: Int -> n % 2 == 1 } // prints: 349  
  
printif(349) { it % 2 == 1 }           // prints: 349
```

# Functions – Lambda

Functions can be passed as parameter from inline definition:

```
fun printif(n: Int, pred: (Int) -> Boolean) {  
    if (pred(n)) {  
        println(n)  
    }  
}  
  
(0..6).foreach { printif(it) { it % 2 == 1 } } // prints: 1, 3, 5
```

# Operators

Precedence	Title	Symbols	Example
Highest	Postfix	<code>++, --, ., ?., ?, !!</code>	<code>i++ obj.func() obj?.field obj!!</code>
	Prefix	<code>-, +, ++, --, !, label</code>	<code>-i --i !b</code>
	Type RHS	<code>: , as, as?</code>	<code>var i: Int objA as TypeB</code>
	Multiplicative	<code>*, /, %</code>	<code>i * j</code>
	Additive	<code>+, -</code>	<code>i + j</code>
	Range	<code>..</code>	<code>for (i in 1 .. 10)</code>
	Infix function	<code>simpleIdentifier</code>	<code>val b1 = b2 or b3 var lst = 1 until 10</code>
	Elvis	<code>?:</code>	<code>val i = obj?.field ?: -1</code>
	Named checks	<code>in, !in, is, !is</code>	<code>5 in lst objA is TypeA</code>
	Comparison	<code>&lt;, &gt;, &lt;=, &gt;=</code>	<code>a &lt; b a &gt;= b</code>
	Equality	<code>==, !=, ===, !==</code>	<code>a != b objA === objB</code>
	Conjunction	<code>&amp;&amp;</code>	<code>b1 &amp;&amp; b2</code>
	Disjunction	<code>  </code>	<code>b1    b2</code>
	Spread operator	<code>*</code>	
Lowest	Assignment	<code>=, +=, -=, *=, /=, %=</code>	<code>var i = 349 i += j</code>

# Standard Types

Category	Type	Range	Examples & Conversions
Numbers	Byte	-128 to 127	127 128.toByte() // yields: -128
	UByte	0 to 255	
	Short	-32768 to 32767	349.99.toInt().toShort() // yields: 349
	UShort	0 to 65535	
Floating point	Int	- $2^{31}$ to $2^{31}-1$	1000003 0x349A // yields 13466
	UInt	0 to $2^{32}-1$	
Text	Long	- $2^{63}$ to $2^{63}-1$	349L 349.toLong()
	ULong	0 to $2^{64}-1$	
Other	Float	23b sign, 8b exp	349.99f 349.99.toFloat()
	Double	52b sign, 11b exp	349.99 1E6 // yields: 1000000.0 3.49E-5 // yields: 0.000349
	Char		'a' '\t' '\n' '\"' '\"' '\\' '\?' \u03BB // yields '\lambda'
	String		"Hello \"World\"\\?\n" """\ Hello 'World'? """\.trimMargin() // yields "Hello \"World\"\\?\n"
	Boolean	false, true	"true".toBoolean()

-



X

# IntelliJ & Kotlin

Kotlin Crash Course, continued

U

CS 349

January 16

# Kotlin Crash Course, continued

U

CS 349

# **Classes**

# Classes

Classes in Kotlin are declared using the keyword `class`.

```
class Pos

fun main() {
    var myPos = Pos()
}
```

# Classes – Primary Constructor

The primary constructor is a part of the class header.

```
class Pos(x: Double, y: Double)
```

```
fun main() {  
    var myPos = Pos(3.0, 4.0)  
}
```

# Classes – Initializer

The primary constructor cannot contain any code. Initialization code can be placed in initializer blocks prefixed with the `init` keyword.

```
class Pos(x: Double, y: Double) {  
    init {  
        println("x: $x; y: $y")  
    }  
}  
  
fun main() {  
    var myPos = Pos(3.0, 4.0) // prints: x: 3.0; y: 4.0  
}
```

# Classes – Properties

Properties of a class can be listed in its declaration by adding the `val` or `var` keyword, or in its body.

```
class Pos(val x: Double, val y: Double) {  
    val eucl = Math.sqrt(x * x + y * y)  
}  
  
fun main() {  
    var myPos = Pos(3.0, 4.0)  
    println("x: ${myPos.x}; y: ${myPos.y}; eucl: ${myPos.eucl}")  
    // prints: x: 3.0; y: 4.0; eucl: 5.0  
}
```

# Classes – Properties & Initializer block(s)

Properties and initializers of a class are evaluated top to bottom.

```
class Pos(val x: Double, val y: Double) {  
    val eucl = Math.sqrt(x * x + y * y)  
  
    init {  
        println("x: $x; y: $y; eucl: $eucl")  
    }  
}  
  
fun main() {  
    var myPos = Pos(3.0, 4.0) // prints: x 3.0; y: 4.0; eucl: 5.0  
}
```

# Classes – Secondary Constructor(s)

A class can also declare secondary constructors, which are prefixed with `constructor`. Each secondary constructor needs to delegate, directly or indirectly, to the primary constructor.

```
class Pos(val x: Double, val y: Double) {
    val eucl = Math.sqrt(x * x + y * y)

    constructor(pos: Pos) : this(pos.x, pos.y) {
        println("Copy constructor")
    }
}

fun main() {
    var myPos = Pos(3.0, 4.0)
    var myCopy = Pos(myPos)    // prints: Copy constructor
}
```

Code in initializer blocks can be considered part of the primary constructor.

# Classes – Anonymous Classes

Anonymous classes are created using the `object` keyword.

```
fun main() {  
  
    val myPos = object { // prints: myPos created  
        var x = 3.0  
        var y = 4.0  
        init {  
            println("Anonymous class instantiated...")  
        }  
        fun getLength(): Double {  
            return sqrt(x * x + y * y)  
        }  
    }  
  
    println(myPos.getLength()) // prints: 5.0  
}
```

Like anonymous functions, these classes are useful for one-time use.

# Classes – Scope Function *apply*

Scope functions execute code within the context of an object.

```
class Course(val name: String) {  
    var grade = -1  
    fun print() {  
        println("Course: $name, ${if (grade != -1) "$grade" else "n/a"})  
    }  
    fun reset() {  
        grade = -1  
    }  
}  
  
fun main() {  
    val cs246 = Course("CS246").apply {  
        this.grade = 92  
    }  
}
```

All code within *apply* is executed on the object of class Course; within *apply*, the object has the identifier *this*; *apply* yields the object.

# Classes – Scope Function *apply* and *also*

Some additional examples:

```
fun main() {  
    val cs246 = Course("CS246").apply {  
        grade = 92  
    }  
    cs246.apply {  
        grade = 95  
    }.print() // prints: Course: CS246, 95  
}
```

The scope function *also* works as *apply*, but the object is labelled **it**:

```
fun main() {  
    val cs246 = Course("CS246").apply {  
        grade = 92  
    }  
    cs246.also {  
        println("Resetting mark of ${it.name}...")  
    }.reset()  
}
```

# Classes – Inheritance, properties

Parent classes and overwritable properties must be declared `open`.  
Overwriting properties must be modified with `override`.

```
open class Pos(val x: Double, val y: Double) {
    open val eucl = Math.sqrt(x * x + y * y)
}

class Pos3D(x: Double, y: Double, val z: Double, w: Double = 1.0):
Pos(x, y) {
    override val eucl = Math.sqrt((x * x + y * y + z * z) / w)
}

fun main() {
    val myPos = Pos3D(3.0, 4.0, 12.0)
    println("eucl: ${myPos.eucl}")      // prints: eucl: 13.0
}
```

# Classes – Inheritance, functions

Overwritable functions must be declared `open`. Overwriting functions must be modified with `override`; access to the parent class via the keyword `super`.

```
open class Pos(val x: Double, val y: Double) {
    open fun calcLen() : Double {
        return Math.sqrt(x * x + y * y)
    }
}

class Pos3D(x: Double, y: Double, val z: Double, w: Double = 1.0):
Pos(x, y) {
    override fun calcLen() : Double {
        return Math.sqrt(super.calcLen() * super.calcLen() + z * z)
    }
}

fun main() {
    val myPos = Pos3D(3.0, 4.0, 12.0)
    println(myPos.calcLen()) // prints: 13.0
}
```

# Classes – Abstract classes and interfaces

```
interface Comparable {  
    fun isEqual(other: Any): Boolean  
}  
  
abstract class Pos(var x: Double, var y: Double) {  
    abstract fun printFun()  
}  
  
class Pos3D(x: Double, y: Double, var z: Double): Pos(x, y), Comparable {  
    override fun printFun() {  
        println("CS349!")  
    }  
    override fun isEqual(other: Any): Boolean {  
        return when (other) {  
            is Pos -> (x == other.x) and (y == other.y) and (z == other.z)  
            else -> false  
        }  
    }  
}
```

# Classes – Anonymous Classes

Anonymous classes can sub-class existing open or abstract classes, or interfaces:

```
abstract class Pos(var x: Double, var y: Double) {  
    abstract fun printFun()  
}  
  
val myPos = object: Pos(3.0, 4.0) {  
    fun getLength(): Double {  
        return sqrt(super.x.pow(2) + super.y.pow(2))  
    }  
    override fun printFun() {  
        println("CS349!")  
    }  
  
    println(myPos.getLength()) // prints: 5.0  
    println(myPos.printFun()) // prints: CS349!
```

## Classes – Data classes

A class whose main purpose is to hold data. Kotlin automatically provides a **copy** and an enhanced **toString** function.

```
data class Pos(val x: Double, val y: Double)

fun main() {
    var myPos = Pos(3.0, 4.0)
    println("${pos.x},${pos.y}") // prints: (3.0,4.0)

    val anotherPos = myPos.copy(y=5.0)
    println(anotherPos)           // prints: Pos(x=3.0, y=5.0)
}
```

# Classes – Enum classes

```
enum class Direction {  
    NORTH, EAST, SOUTH, WEST  
}
```

Enumeration entries behave like sub-classes of the Enum class:

```
enum class Direction {  
    NORTH { override fun turnRight(): Direction { return EAST } },  
    EAST { override fun turnRight() = SOUTH }, // shortened from above  
    SOUTH { override fun turnRight() = WEST },  
    WEST { override fun turnRight() = NORTH };  
  
    abstract fun turnRight(): Direction  
}  
  
fun main(args: Array<String>) {  
    val direction = Direction.SOUTH  
    println(direction.turnRight()) // prints: WEST
```

# Scope

Annotations or visibility modifiers go before the constructor or function name.

Kotlin defaults to “public” scope if you omit the modifier (which we will often do in examples).

Modifier	Scope
<code>public</code>	Everywhere
<code>protected</code>	Class and subclasses
<code>private</code>	Class
<code>internal</code>	Module

# *Strings*

# Strings

Strings are represented by the **String** class and are immutable. A string value is a sequence of characters in double quotes (" "):

```
var str = "CS349"  
println(str.lowercase()) // prints: cs349  
println(str)           // prints: CS349
```

Strings can be accessed via the indexing operator and iterated over:

```
println(str[0])          // prints: C  
  
for (c: Char in str)    // prints: C_S_3_4_9_  
    print("$c_")  
  
str.forEach { print(it.lowercaseChar()) } // prints: cs349
```

# Strings

Strings are represented by the `String` class and are immutable. A string value is a sequence of characters in double quotes (" "):

```
val myCourseCode = "CS"  
val myCourseNumber = "349"
```

Strings can be concatenated using the `+`-operator:

```
println(myCourseCode + myCourseNumber) // prints: CS349  
println("$myCourseCode $myCourseNumber") // prints: CS 349
```

# String Templates

String literals may contain template expressions – pieces of code that are evaluated and whose results are concatenated into the string.

A template expression starts with a dollar sign (\$) and consists of either a name or an expression in curly braces ({}).

```
fun sum(a: Int, b: Int): Unit {  
    println("$a + $b = ${a + b}")  
}  
sum(349, 42) // prints: 349 + 42 = 391  
  
val str = "CS349"  
println("$str.length is ${str.length}") // prints: CS349.Length is 5
```

# *Collections*

# Collections

Kotlin distinguished between normal (immutable) and mutable collections:

Immutable:

- List
- Set
- Map

Mutable:

- MutableList
- MutableSet
- MutableMap

# Collections – Lists

`List<T>` stores elements in a specified order and provides indexed access to them:

```
var fruits = listOf("apple", "banana", "cherry")
fruits.forEach { println(it) }           // prints: apple, banana, cherry
fruits = listOf("apricot", "blueberry", "coconut")
println(fruits[2])                      // prints: coconut
println(fruits.get(0))                  // prints: apricot
println(fruits.indexOf("blueberry"))    // prints: 1
println(fruits.size)                   // prints: 3

val animals = mutableListOf("ape", "beaver", "camel")
animals.forEach { println(it) }        // prints: ape, beaver, camel
animals.removeAt(0)
animals.remove("camel")
println(animals)                      // prints: [beaver]
animals.add("comoran")
animals.add(0, "ant")
println(animals)                      // prints: [ant, beaver, comoran]
```

# Collections – Sets

Set<T> stores unique elements; their order is generally undefined. null elements are unique as well: a Set can contain only one null. Two sets are equal if they have the same size, and for each element of a set there is an equal element in the other set.

```
val set1 = setOf(1, 1, 2, null)
val set2 = setOf(1, 2, null, null, null)
val set3 = mutableSetOf(1, 1, 2, 3)

println(set1 == set2)           // prints: true
println(set1 == set3)           // prints: false
println(set1.union(set3))     // prints: [1, 2, null, 3]
println(set1.intersect(set3)) // prints: [1, 2]
println(set1.minus(set2))     // prints: []
```

# Collections – Maps

Map<K, V> stores key-value pairs (or entries); keys are unique, but different keys can be paired with equal values.

```
val days = mapOf(1 to "Monday", 4 to "Thursday", 6 to "Saturday")
println(days[3])           // prints: null

val langs = mutableMapOf("135" to "Racket",
                        "136" to "C",
                        "349" to "Kotlin")
println(langs["349"]) // prints: Kotlin
langs["246"] = "C++"
langs["135"] = "Python"
println(langs)          // prints: {135=Python, 136=C, 349=Kotlin, 246=C++}
```

# Collections – Higher Order Functions

```
(0..4).filter { it % 2 == 0 }          // yields: [0, 2, 4]
(0..4).map { it * 2 }                 // yields: [0, 2, 4, 6, 8]
(0..4).fold(0) { acc, elem -> acc + elem } // yields: 10
(0..4).reduce { acc, elem -> acc + elem } // yields: 10
(0..4).foreach { print(it) }           // prints: 01234, yields: Unit
(0..4).onEach { print(it) }            // prints: 01234, yields: 0..4

val (even, odd) = (0..4).partition { it % 2 == 0 }
                           // yields: [0, 2, 4], [1, 3]

var factors = (0..10).groupBy { it % 3 }
                           // contains: {0=[0, 3, 6, 9], 1=[1, 4, 7, 10], 2=[2, 5, 8]}

var primes = (1..21).groupBy { candidate: Int ->
    (2 until candidate).none { test: Int ->
        candidate % test == 0 } }
// contains: {true=[1, 2, 3, 5, 7, 11, 13, 17, 19],
//             false=[4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21]}

println(primes.map { it.value.count() } ) // prints: [9, 12]
```

# *Conditionals*

# Conditionals – `if`

`if`-conditionals yield a value:

```
fun parity(i: Int): Unit {  
    if (i % 2 == 0) {  
        println("$i is even")  
    } else {  
        println("$i is odd")  
    }  
}
```

```
fun sign(i: Int): Int {  
    return if (i > 0)  
        1  
    else if (i < 0)  
        -1  
    else  
        0  
}
```

## Conditionals – `when`

`when` defines a conditional expression with multiple branches, it may yield a value:

```
fun selector(course: String): Unit {  
    when (course.take(2)) {  
        "CS" -> println("Computer Science course")  
        "BU" -> println("Business course")  
        else -> println("unknown course")  
    }  
}  
  
fun selector(course: String): String {  
    return when (course.take(2)) {  
        "CS" -> "Computer Science course"  
        "BU" -> "Business course"  
        else -> "unknown course"  
    }  
}
```

# *Loops*

## Loops – `for` and `forEach`

`for` loops through any data that provides an iterator.

```
for (c in "CS349") { // prints: CS349
    print(c)
}

for (i in 0..9) {      // prints: 0123456789
    print(i)
}
```

`forEach` iterates through any data that provides an iterator.

```
val printInt: (Int) -> Unit = { i: Int -> print(i) }
(0..9).forEach(printInt)      // prints: 0123456789
```

A more concise (i.e., common, Kotlin-like, “sophisticate”) way to express the above is using an anonymous function:

```
(0..9).forEach { print(it) } // prints: 0123456789
```

## Loops – **for** and **forEach**

**for** loops through any data that provides an iterator.

```
val fruits = listOf("Apple", "Banana", "Cherry")

for (i in 0 until fruits.size) { // i runs from 0 to fruits.size - 1
    println("Element at $i is \">${fruits[i]}\"")
}
// prints: Element at 0 is "Apple"
//           Element at 1 is "Banana"
//           Element at 2 is "Cherry"
```

While the code above is working correctly, it is not an elegant way for list traversal in Kotlin.

```
for ((index, value) in fruits.withIndex()) {
    println("Element at $index is \"$value\"")
}
```

Generally, avoid indices if not strictly necessary.

## Loops – `for` and `forEach`

`for` loops through any data that provides an iterator.

```
val fruits = mapOf("Apple" to 2.99, "Banana" to 0.69, "Cherry" to 4.99)

for ((key, value) in fruits) {
    println("Price for $key is \$value")
}
```

`forEach` iterates through any data that provides an iterator.

```
fruits.forEach {
    (key, value) -> println("Price for $key is \$value")
}
```

# *Ranges and Progressions*

# Ranges and Progressions

A range defines a closed interval in the mathematical sense: it is defined by its two endpoint values which are both included in the range.

```
var range = 5..10          // contains: 5, 6, 7, 8, 9, 10
range = 5 until 10         // contains: 5, 6, 7, 8, 9

for (value in range)      // prints: 5, 6, 7, 8, 9
    println(value)

val existing = 8 in range // true
val missing = 42 !in range // true
```

Ranges can be defined for **any comparable data type**.

# Ranges and Progressions

A progression is defined by its start and end points, as well as a non-zero step.

```
var prog = 10 downTo 5    // contains: 10, 9, 8, 7, 6, 5
prog = 10 downTo 5 step 2 // contains: 10, 8, 6

for (value in range)      // prints: 5, 6, 7, 8, 9
    println(value)

val i = 7
println("$i is in${if (i !in prog) " not " else " }in
        progression")          // prints: 7 is in not in progression
```

# ***Nullable Data***

# Nullable Data

A reference must be explicitly marked as nullable when null value is possible. Nullable type names have a ? at the end:

```
var myString: String = "CS349"  
println(myString.length)           // prints: 5  
var maybeNullString: String? = null
```

? . is the safe call operator. It only continues with dereferentiation if the data is non-null:

```
var maybeNullString: String? = null  
println(maybeNullString?.length) // prints: null  
maybeNullString = myString  
println(maybeNullString?.length) // prints: 5
```

# Nullable Data

`? :` is a ternary operator for `null` (also called the Elvis operator). It will return the LHS if non-`null`, otherwise it returns the RHS of the expression.

```
var maybeNullString : String? = null  
println(maybeNullString?.length ?: 0) // prints: 0
```

`!!` converts any nullable value to a non-`null` type and throws an exception if the value is `null`. Only use this operator if you are certain that a value is not `null`!

```
var maybeNullString: String? = "abc"  
var nonNullString: String = maybeNullString!! // will fail if  
var nonNullStringLength = maybeNullString!!.length // string is null
```

# Nullable Data

By using the ?: operator, we can “remove” nullable data; and by using the `is` operator, we can “weaken” the strict typing of Kotlin.

```
var myList = listOf(1, 2, 3, null) // myList is of type List<Int?>
myList.forEach { // prints: 1, 2, 3, null
    println(it) // it is of type Int?
}

println(myList.fold(0) { acc, elem -> acc + (elem ?: 0) }) // prints: 6
// type of acc is Int, type of elem is Int?, type of (elem ?: 0) is Int

var myList = listOf(1, 2, 3, null, "Four")
println(myList.fold(0) { acc, elem -> // prints: null 9
    acc + when (elem) { // is of type Int; when(elem) yields an Int
        is Int -> elem
        is String -> elem.length
        null -> { println("null")
                    -1 }
        else -> 0
    } })
}
```

# End of the Chapter



- Get your tool-chain up and running!
- While Kotlin works with many programming styles, it is best to adopt a “functional” mindset with your implementation.
- Higher-order functions and anonymous functions (and classes) rule!
- Try to remember “all the small things” that can make your implementation shorter / more legible / “slicker”



Any further questions?

-



X

# JavaFX

The GUI Stack

JavaFX

# U

CS 349

# January 18

-



X

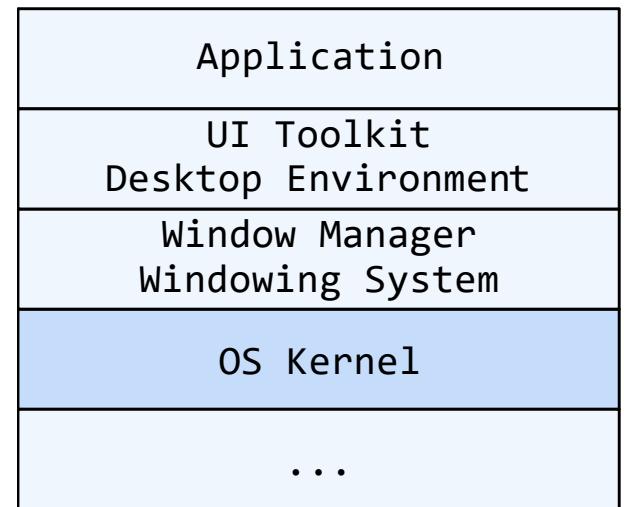
# The GUI Stack

U

CS 349

# GUI Stack Components

OS Kernel: hardware access,  
device management, see CS350

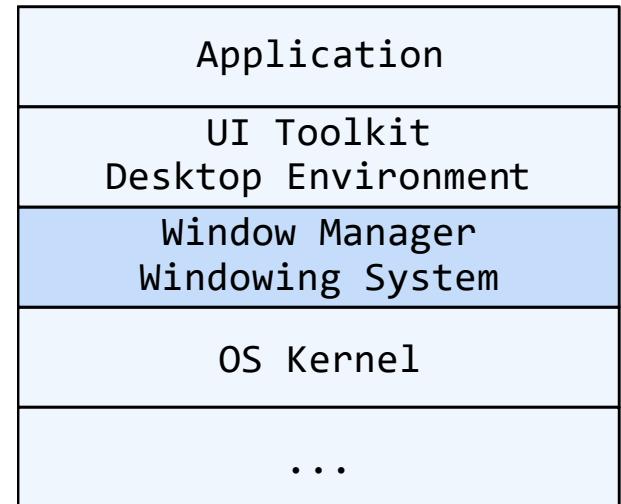


# Window Manager (WM)

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.



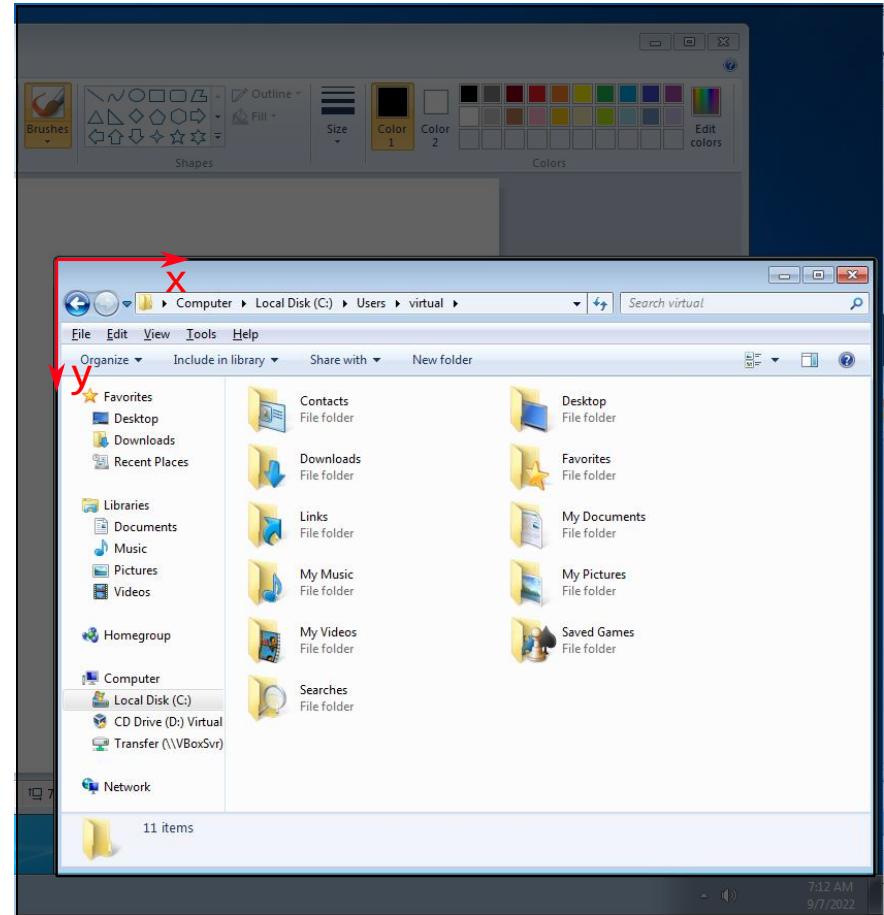
# Window Manager – Canvas Abstraction

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

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

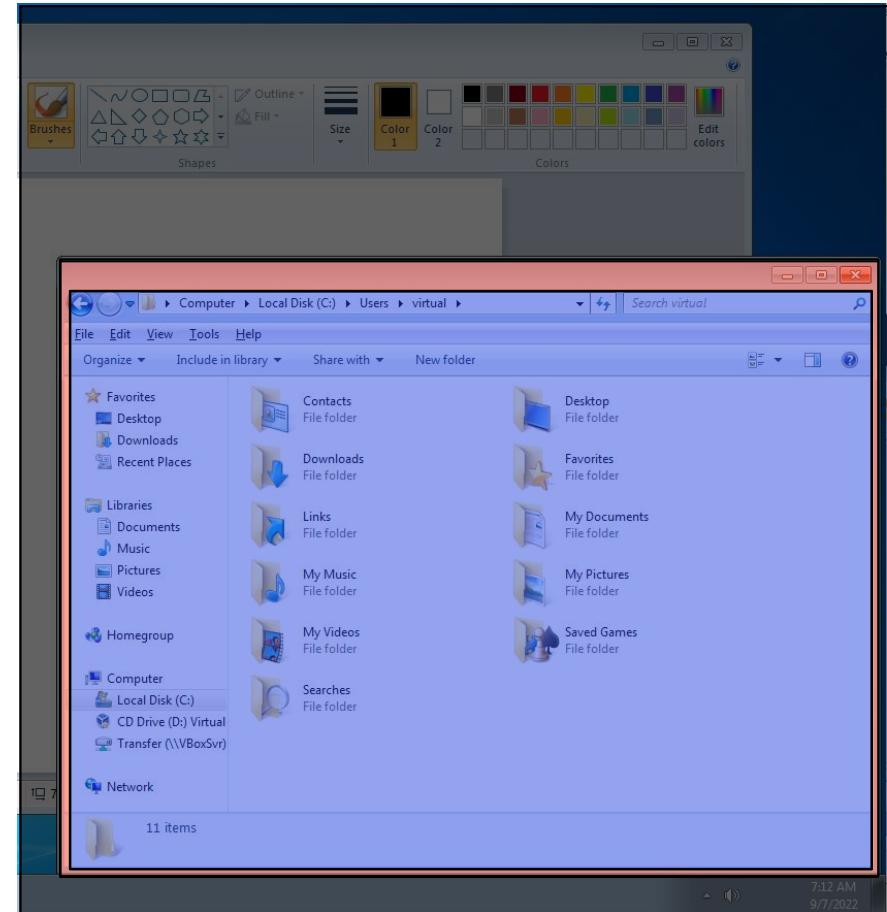
Each window has its **own coordinate system**:

- 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

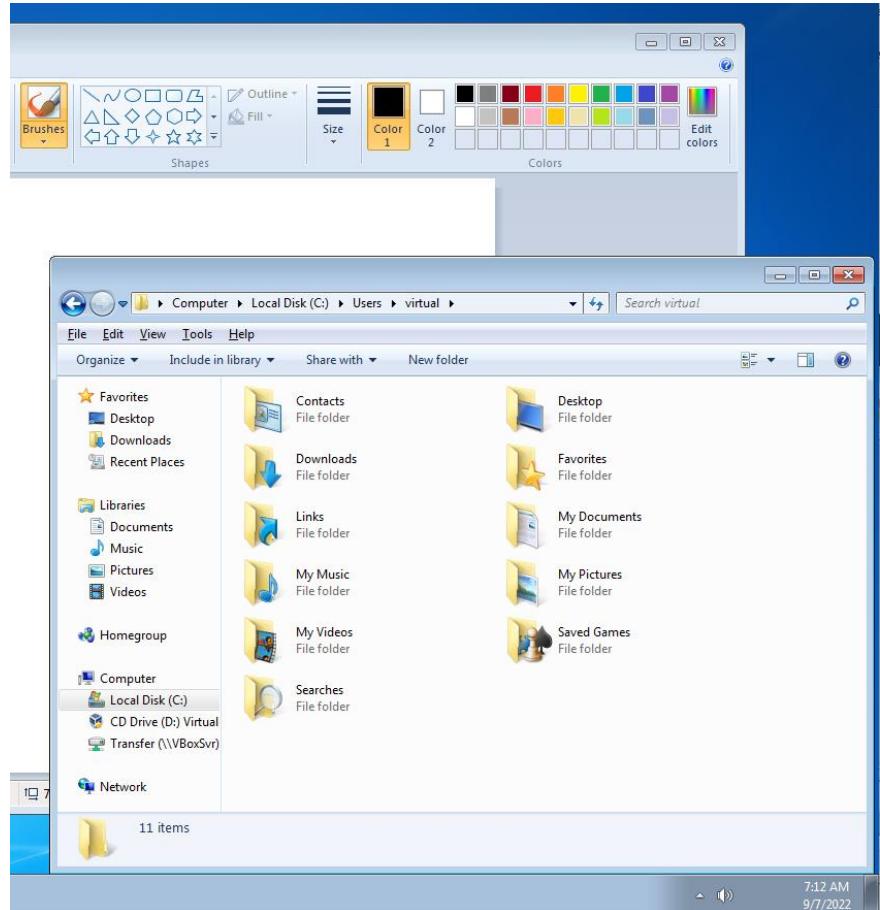
While the **windows manager** “owns” the application window, the **application** “owns” the content of the application window.



# Window Manager – Additional Functionality

A window manager also provides:

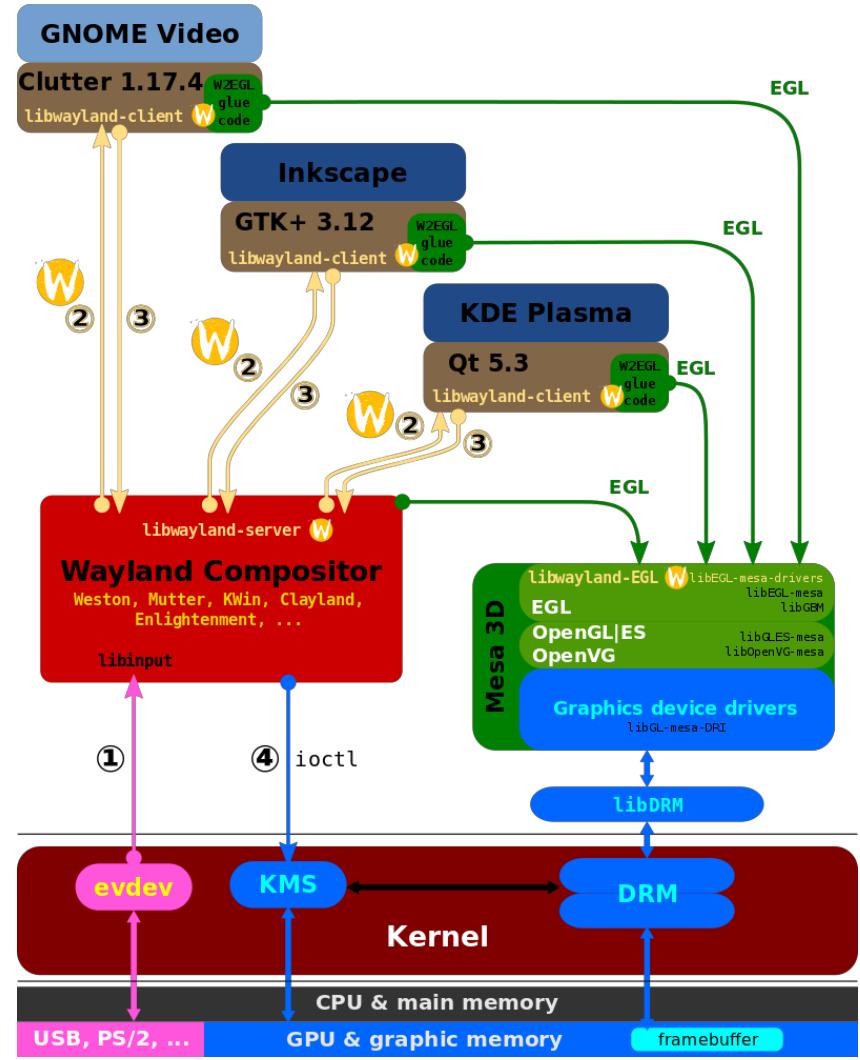
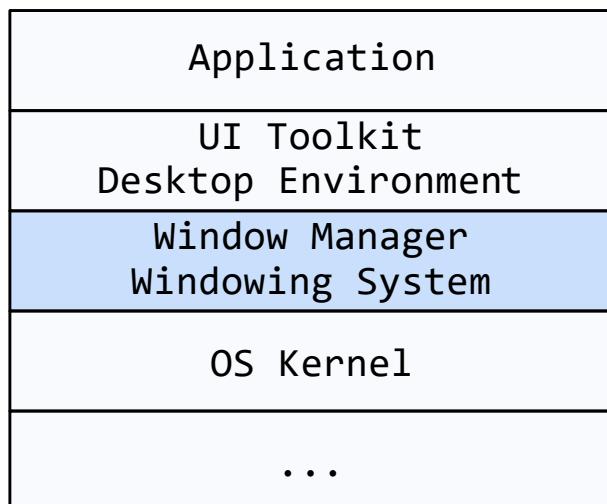
- Facilities to modify size and location of each window (resize handle, move handle, etc.)
- Window-related interactive components (close button, minimize button, etc.)



# Window Manager – Architectures

Examples for Window Manager architectures:

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



# Window Manager – Examples

## Examples for Window Managers:

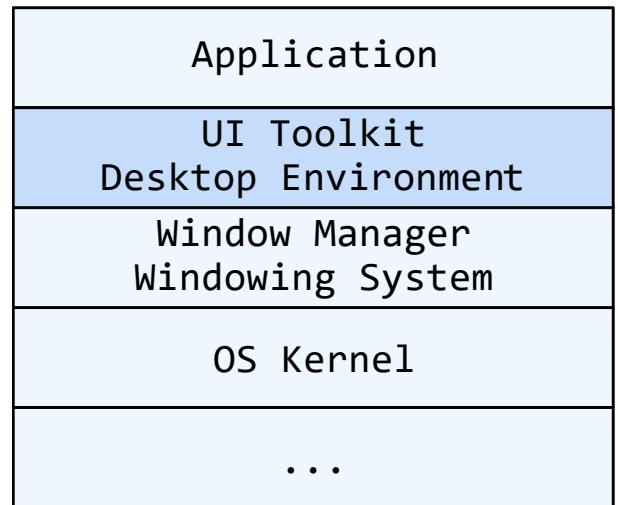
- Windows 1.0
  - 1985
  - Tiling
  - Integrated
- Window Maker
  - 1997
  - Stacking
  - X11 as *Windowing System*; e.g., GNUstep as *Desktop Environment*
- Mutter
  - 2011
  - Compositing
  - Wayland as *Windowing System*; e.g., GNOME as *Desktop Environment*



# UI Toolkits

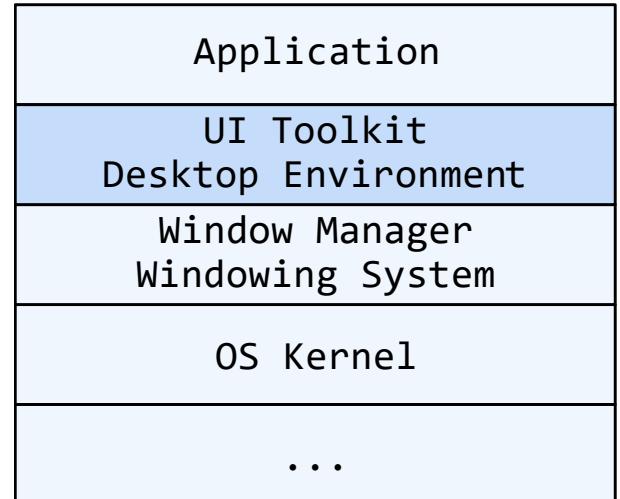
Window Managers include only  
**basic capabilities for input,  
output, and window management.**

For implementing the actual  
content of a UI, we need a UI  
Toolkit.



# UI Toolkits

A UI toolkit is set of classes for building User Interfaces.



Low-level (or native or heavyweight) toolkits:

- Built into or tightly integrated with the underlying OS. Examples include Win32 on Windows, Xlib on Unix, and Cocoa on Mac
- Provided by the OS vendor and strictly tied to the underlying OS

High-level (or lightweight) toolkits:

- Sit “above” the operating system, with no tight integration / coupling. Examples include Qt, Gtk+, wxWidgets, Swing, and JavaFX
- Often provided by a third-party, May or may not have a “native” look-and-feel for a platform

# UI Toolkits

Toolkits provide components and capabilities, including:

- Output
  - User interface widgets
  - Graphics primitives, e.g., shapes and images
  - Animation
  - Media playback, i.e., sound and video
- Input
  - Standard input handling, e.g., mouse and keyboard
  - Access to cameras, sensors, etc.

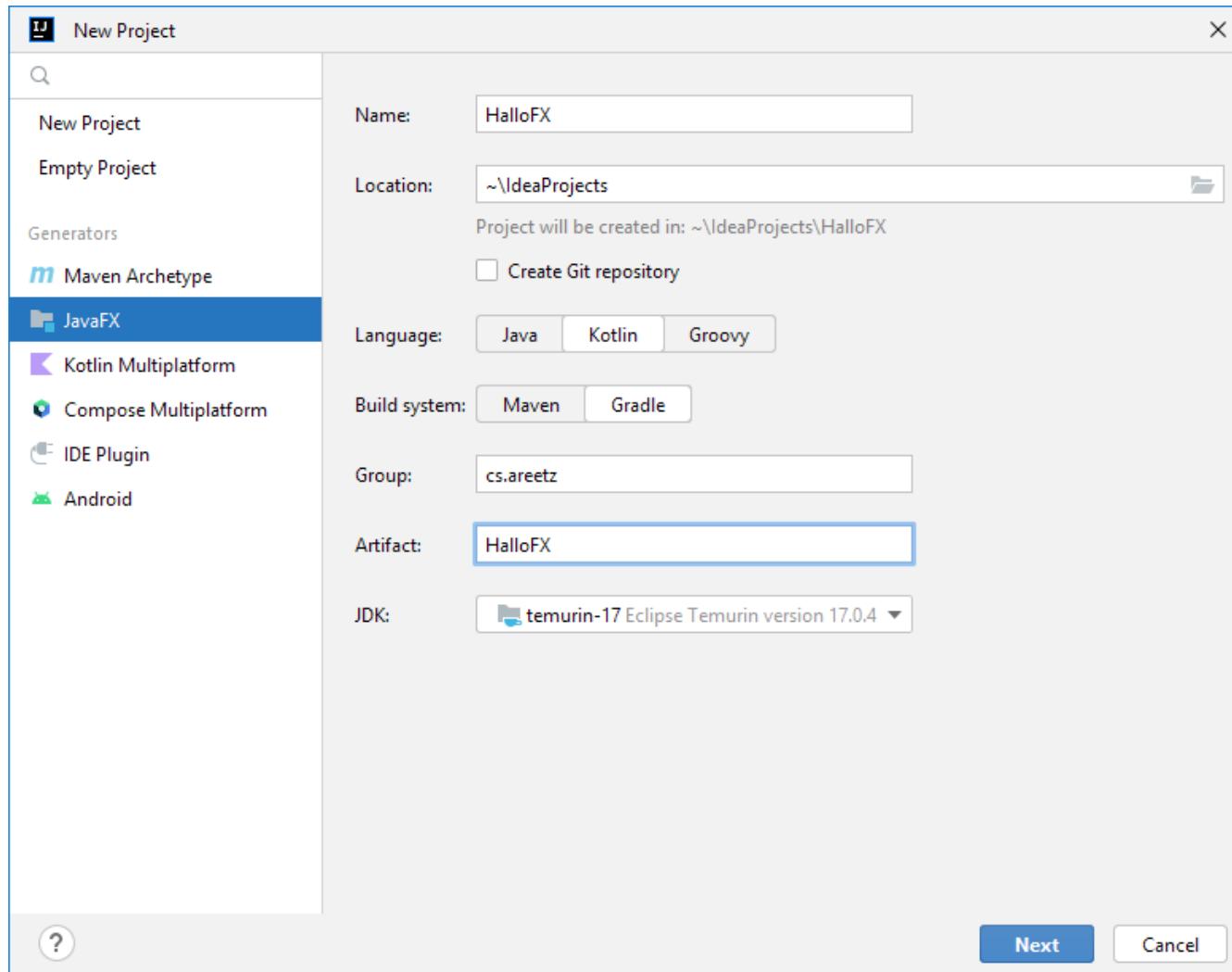


JavaFX

U  
CS 349

# Create a JavaFX Project

Creating a new project:

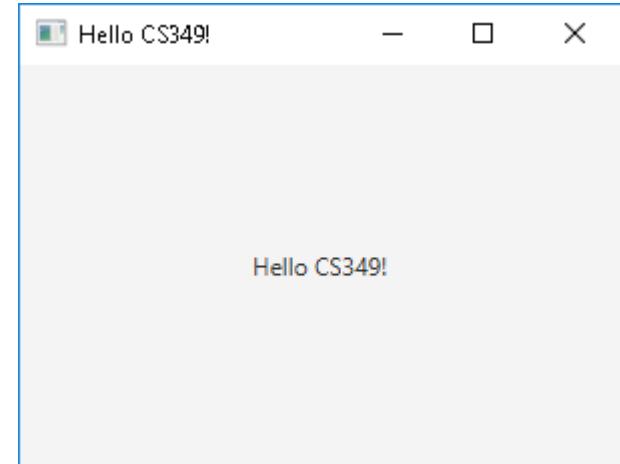


# Hello JavaFX

```
package ui.lectures.hellofx

import ...

class HelloFXApplication : Application() {
    override fun start(stage: Stage) {
        val welcome = Label("Hello CS349!")
        val root = StackPane(welcome)
        stage.apply {
            title = "Hello CS349!"
            scene = Scene(root, 300.0, 200.0)
        }.show()
    }
}
```



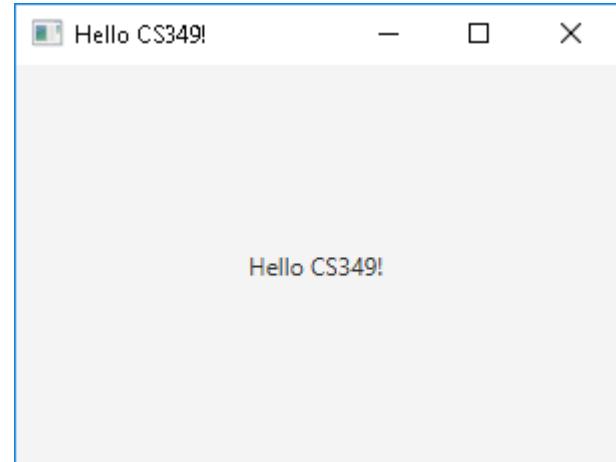
# Hello JavaFX

```
package ui.lectures.hellofx

import ...

class HelloFXApplication : Application() {
    override fun start(stage: Stage) {

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



This implementation has a different style in that the StackPane and the Label remain anonymous.

# Application Lifecycle

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

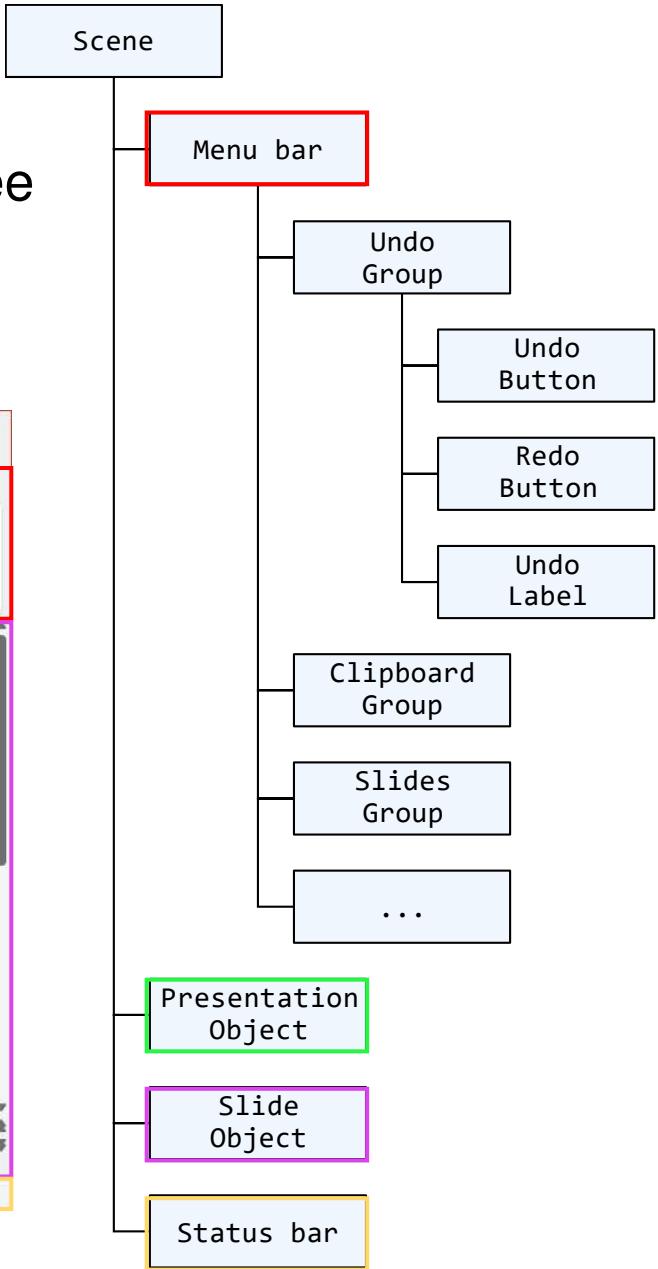
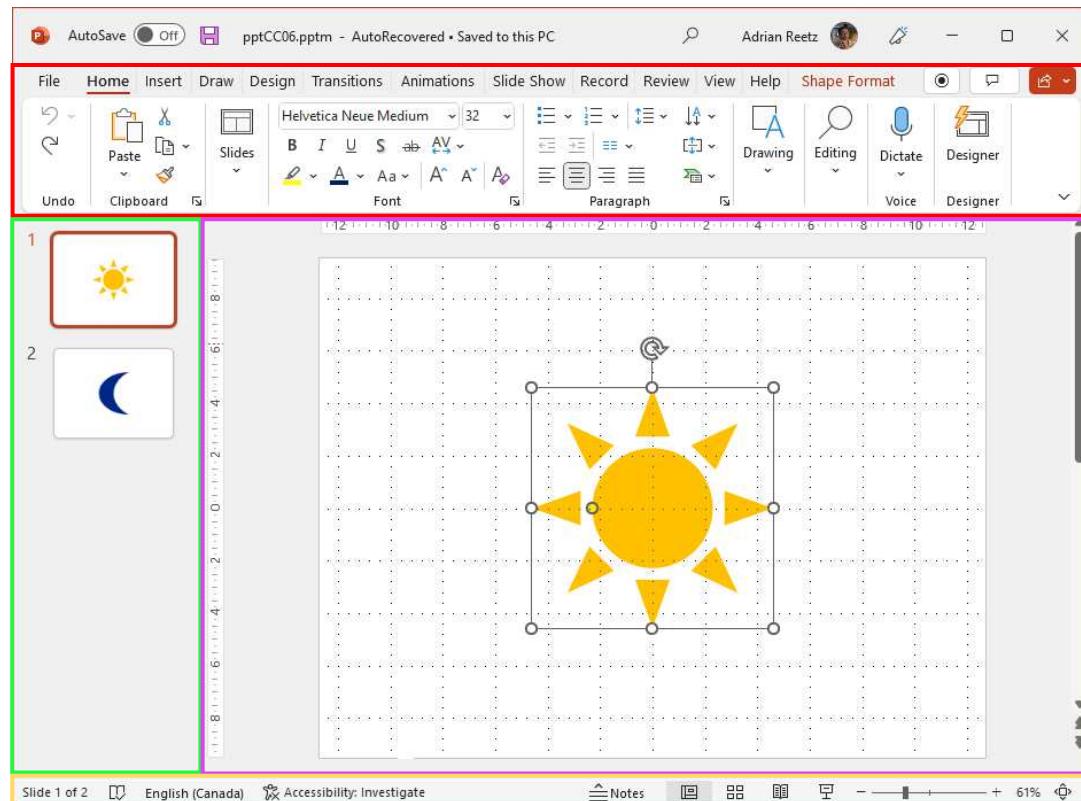
The JavaFX runtime does the following when an application is launched:

- 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.

The `start()` method is abstract and must be overridden. The `init()` and `stop()` methods are optional but may be overridden.

# Scene Graph

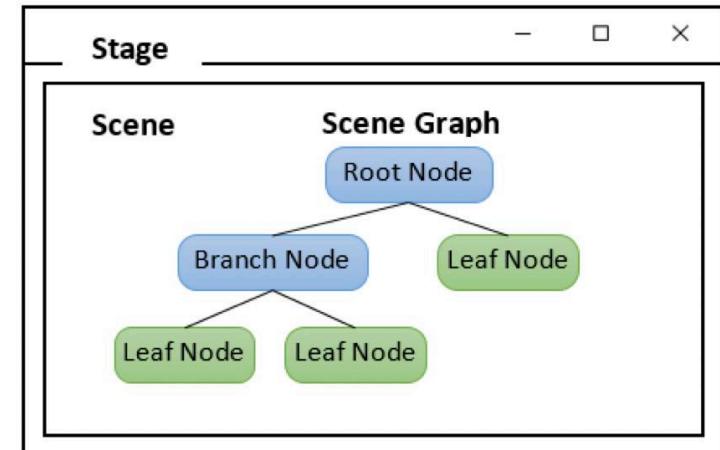
In computer graphics, a **scene graph** is a tree structure that arranges all the elements of a screen into a hierarchy.



# 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



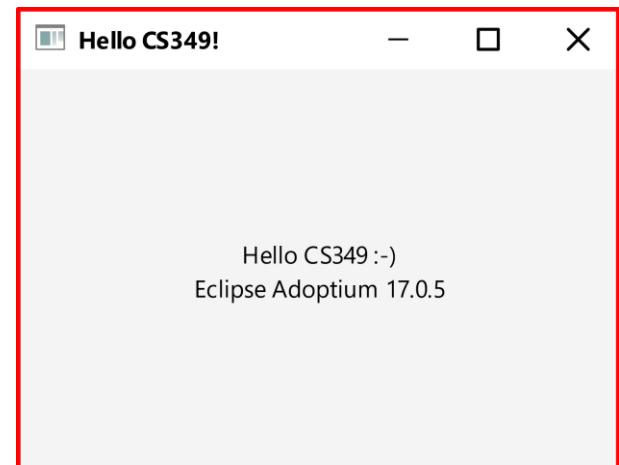
JavaFX stores an interface as a scene graph.

- 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.

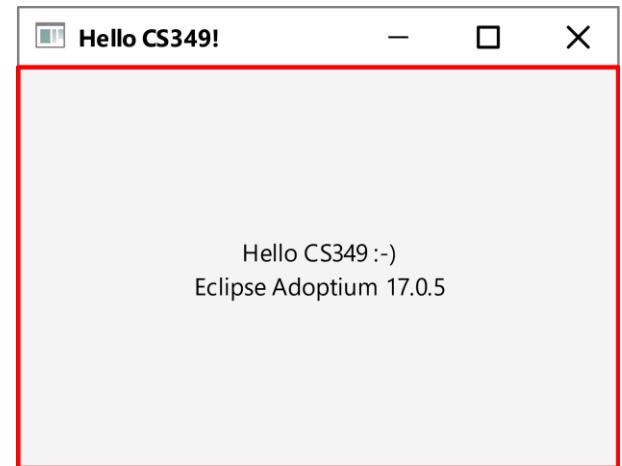
```
override fun start(stage: Stage) {  
    val greeting = Label("Hello CS349 :-)")  
  
    val vendor = Label(System.getProperty("java.vendor"))  
    val version = Label(System.getProperty("java.version"))  
    val javaInfo = HBox(vendor, version).apply {  
        alignment = Pos.CENTER  
    }  
  
    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**

-



X

U

CS 349

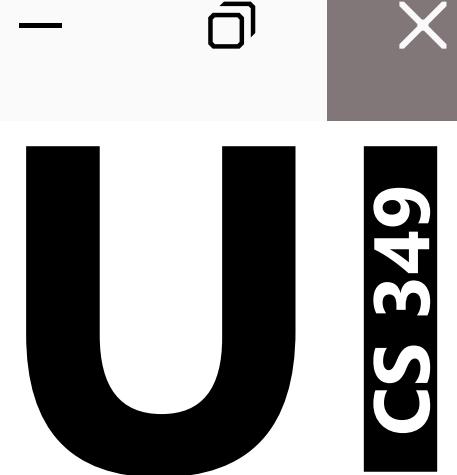
# Widgets

Purpose of Widgets

Widgets in JavaFX

Properties & Property Binding

January 18



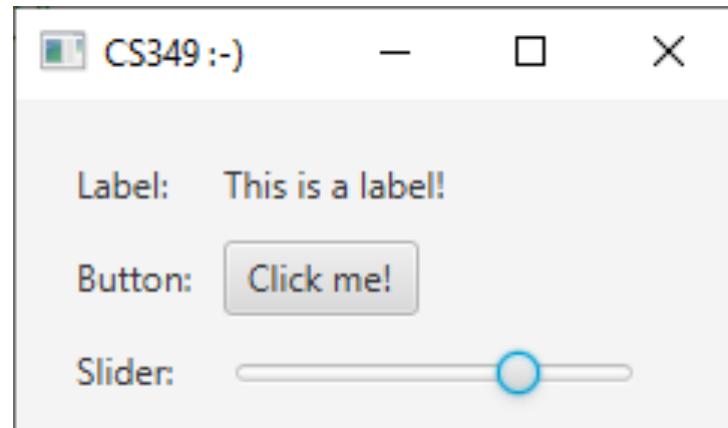
# Purpose of Widgets

# User Interface Widgets

Widgets are parts of an interface that have **their own behavior** (e.g., buttons, drop-down menus, spinners, file dialog boxes, progress bars, slider). They are also called *components*, *controls*, or *UI elements*.

They can perform **four essential functions**:

- capture user input
- generate events
- provide user feedback
- maintain state



# User Interface Widgets

## Capture user input

- Capture user input in various forms
- The type of input varies with the widget

## Generate events

- They generate events (i.e., messages) that can be sent to other parts of your application to indicate that the user has done something

## Provide feedback

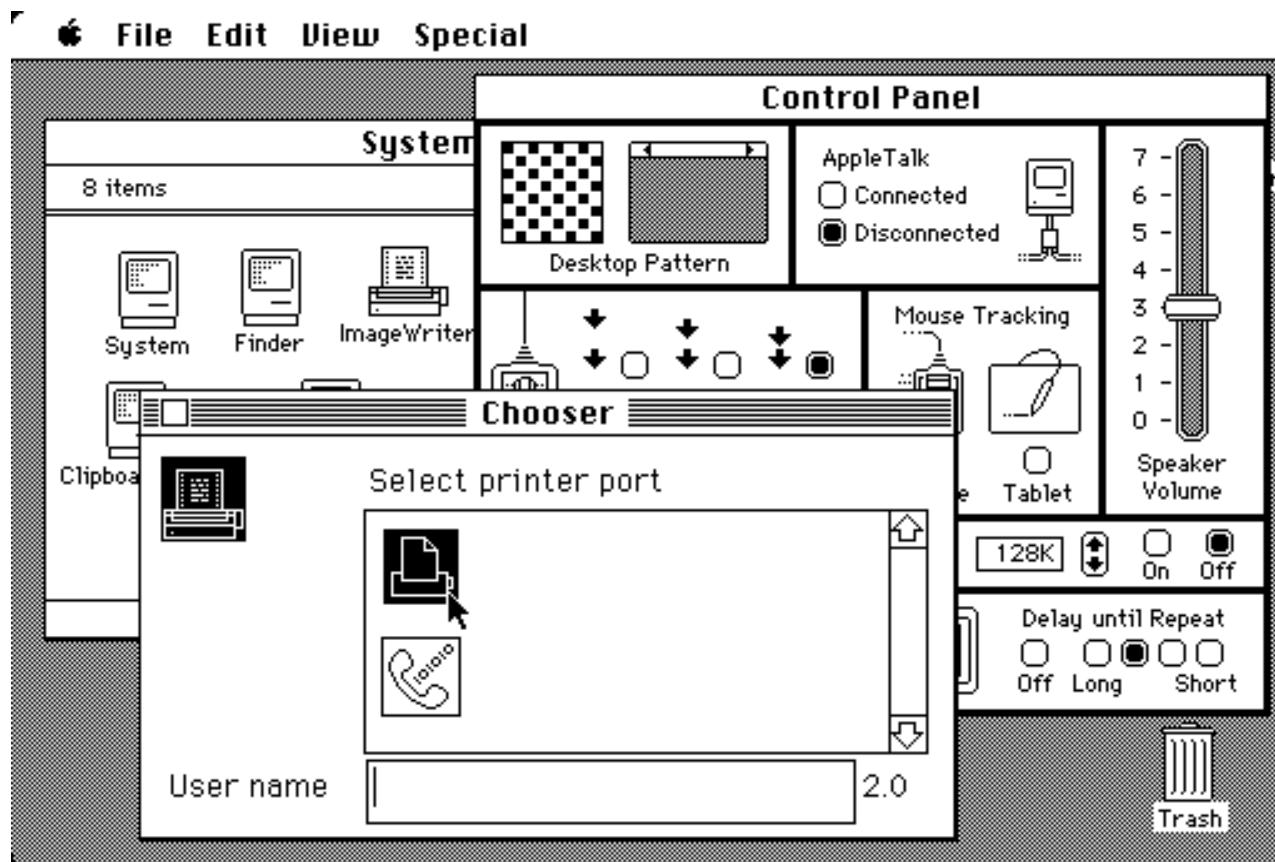
- Provide user feedback indicating that they have been activated (whatever that means for that particular widget)

## Maintain state

- They may have state or data that they retain and control, that can represent state to the user

# User Interface Widgets

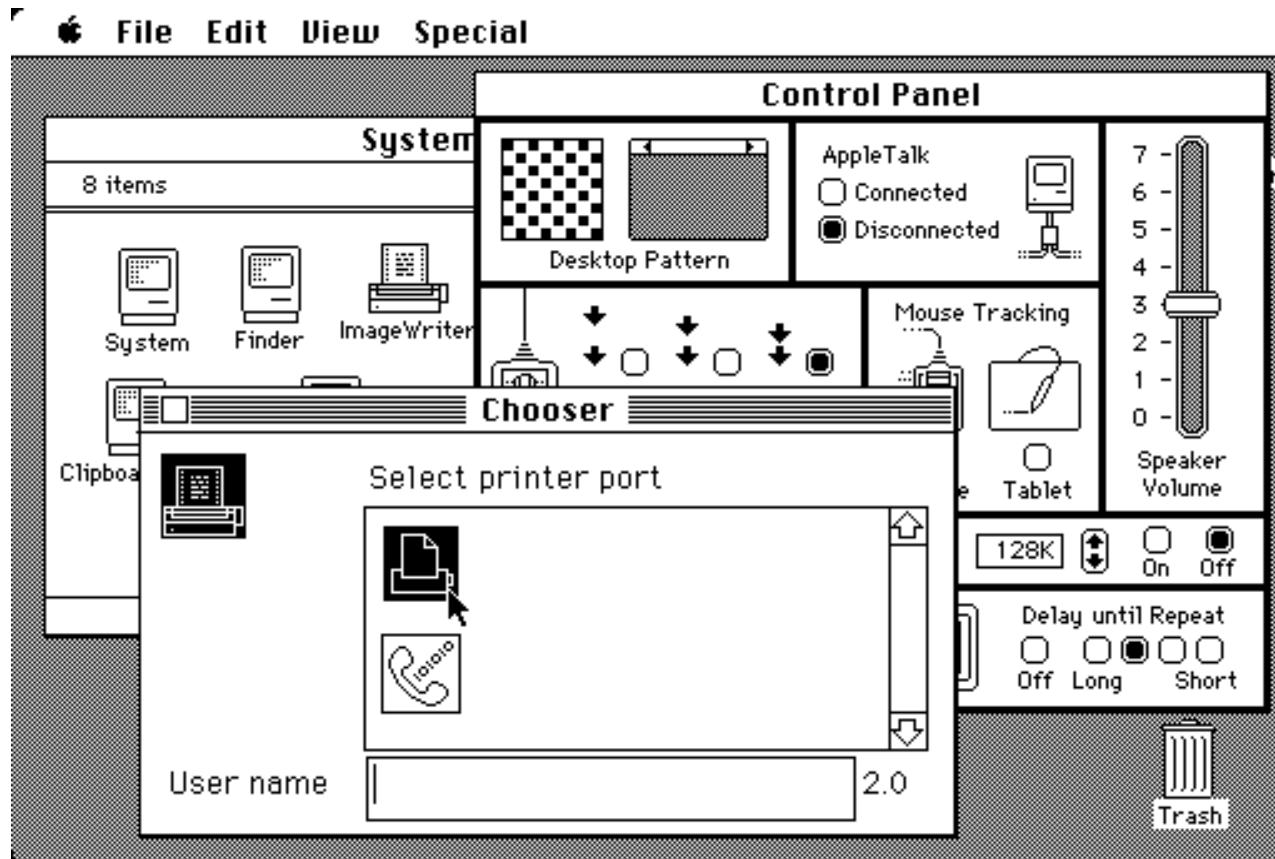
The original **8 widgets**: buttons, menu, radio buttons, checkbox, slider, textbox, scrollbar, spinner



# User Interface Widgets

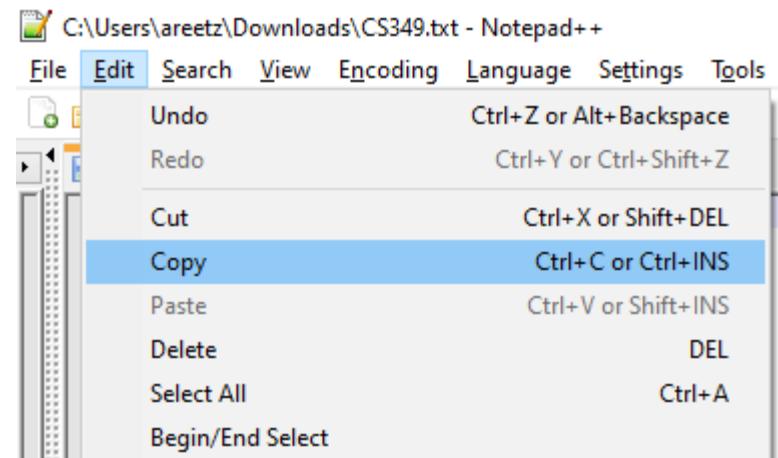
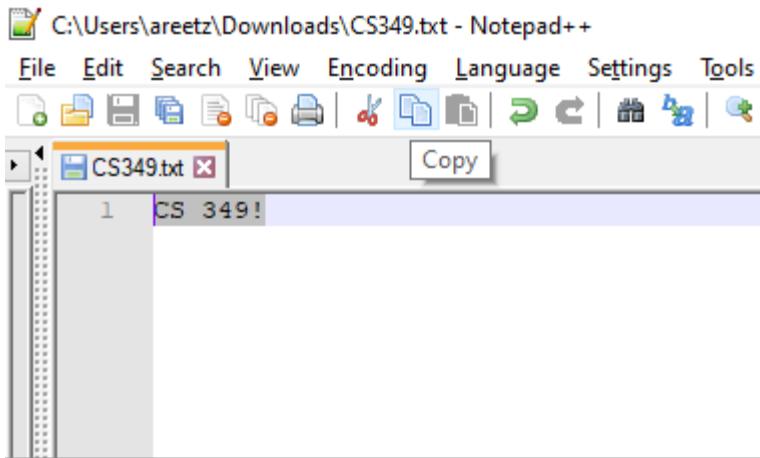
Packaged in GUI toolkits, such as, SwiftUI, WinUI, Gtk+, Qt, and JavaFX

Widget toolkits vary in presentation, but all include “standard” widgets.



# User Interface Widgets

Different widgets can provide the same conceptual functionality in different ways.



# Logical Inputs vs. Widgets

Logical inputs describe the underlying functionality (i.e., the type of input or interaction that they support). This includes state and events.

- State: what data does the widget need to store?
  - e.g., label holds a string, slider holds min / current / max values
- Events: what messages does the widget generate when activated?
  - e.g., buttons generate “activated”; sliders generate “changed” events

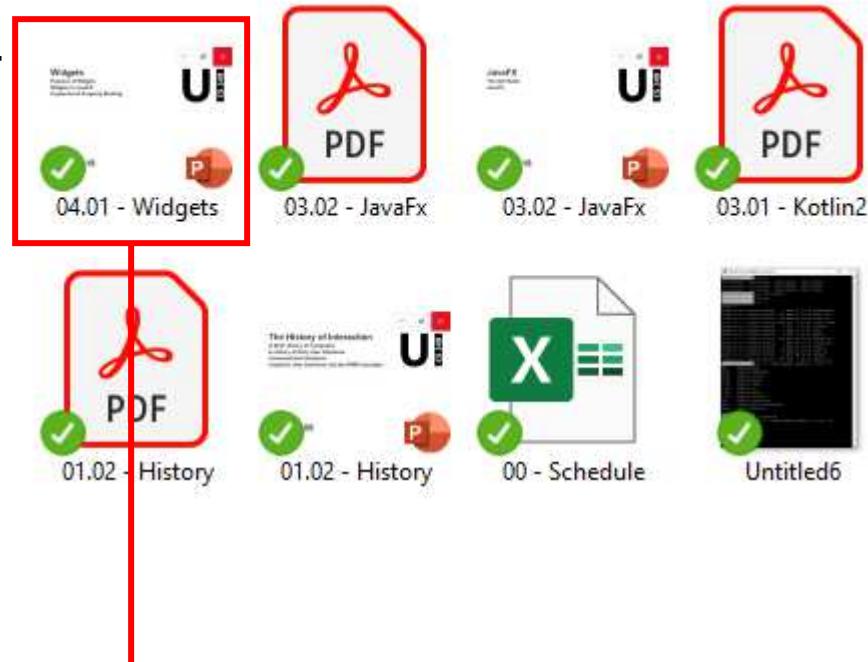
Widgets are implementations of logical inputs, and define their appearance. They add properties to logical input.

- Properties: values that determine how the widget is presented. Properties may be general (e.g., position and size) or specific (e.g., text)
  - Common properties: position (x,y), size (width, height), color
  - Custom properties: specific to a logical input, e.g., orientation for slider

# Logical Display

Displays text or images to the user. Purpose is displaying data or providing feedback.

- States:
  - Text: String
- Events:
  - None
- Examples:
  - Labels
  - Images



Selection top left: -367, -195. Bounding rectangle size: 1920 × 1080. Area: 317,867 pixels square

781 × 407

848, -19

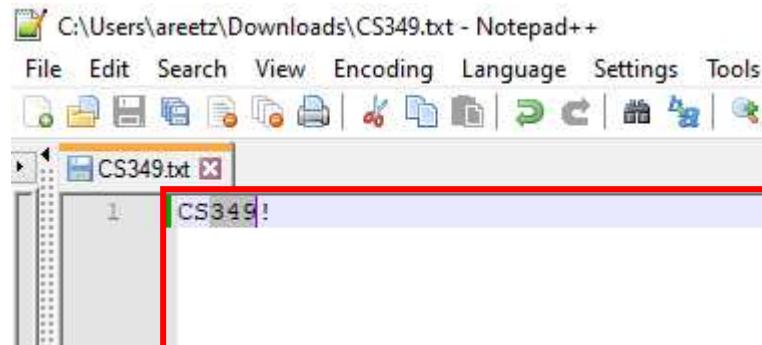
px 100%



# Logical Text Entry

Allows users to enter text and displays the current state.

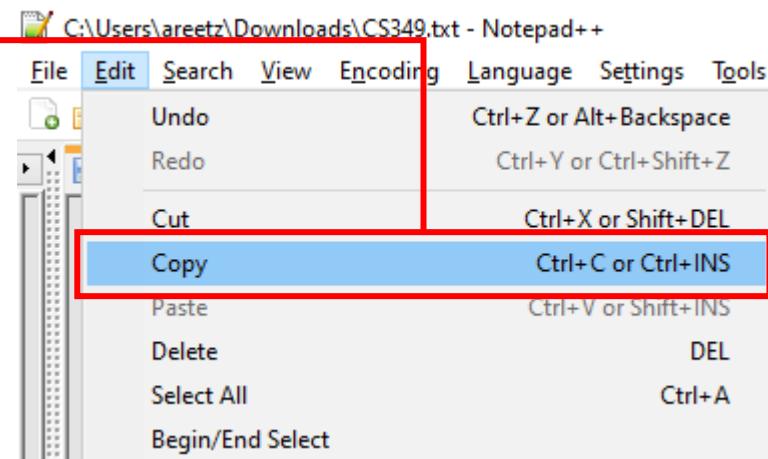
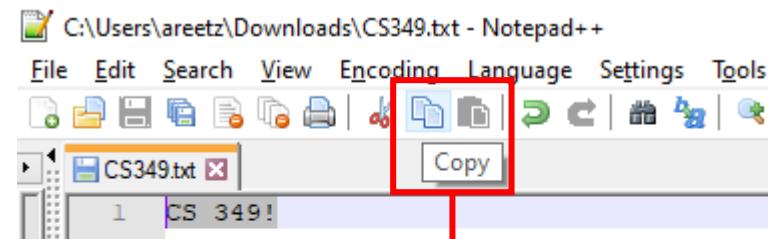
- States:
  - Text: String
  - Selection: Range
- Events:
  - Text changed
  - Entry complete
  - Selection changed
- Examples:
  - Text fields
  - Text areas



# Logical Button

Enables users to perform a simple interaction, with a single fixed action.

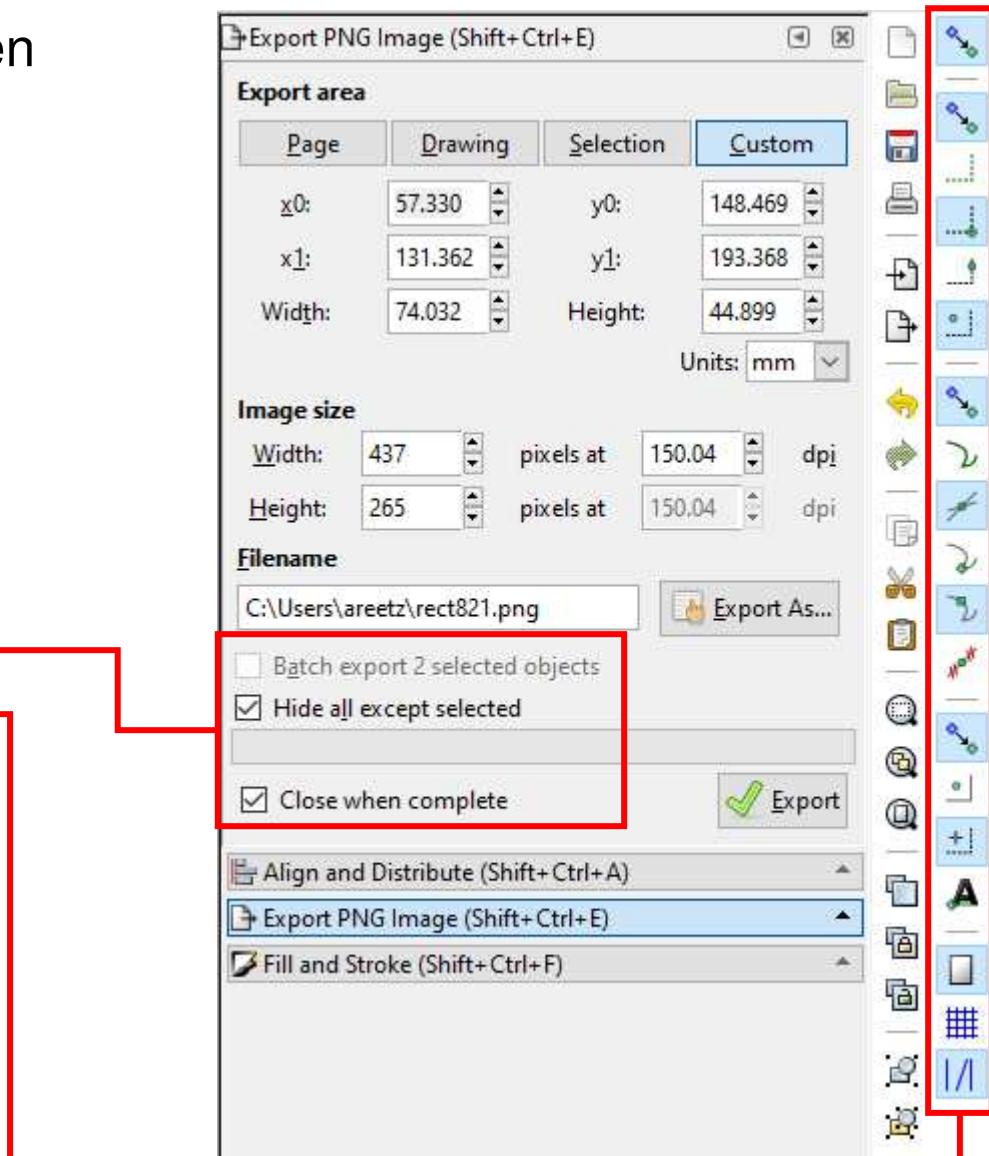
- States:
  - none
- Events:
  - Button activated
- Examples:
  - Buttons
  - Menus



# Logical Boolean Selection

Allows users to select between two states and displays the current state.

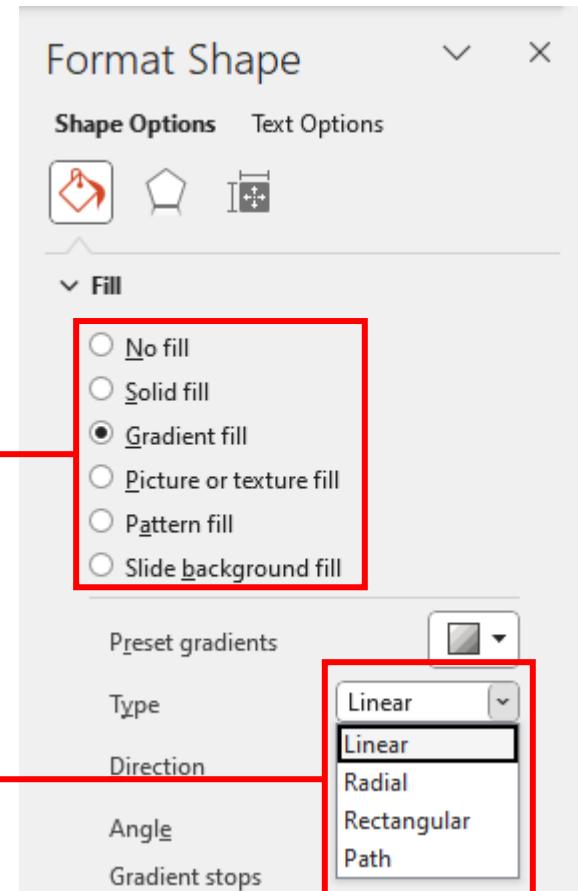
- States:
  - Selection: Boolean
- Events:
  - Selection changed
- Examples:
  - Checkboxes
  - Toggle buttons



# Logical Discrete Selection

Allows users to select one entry from an arbitrary list of discrete elements and displays the current state.

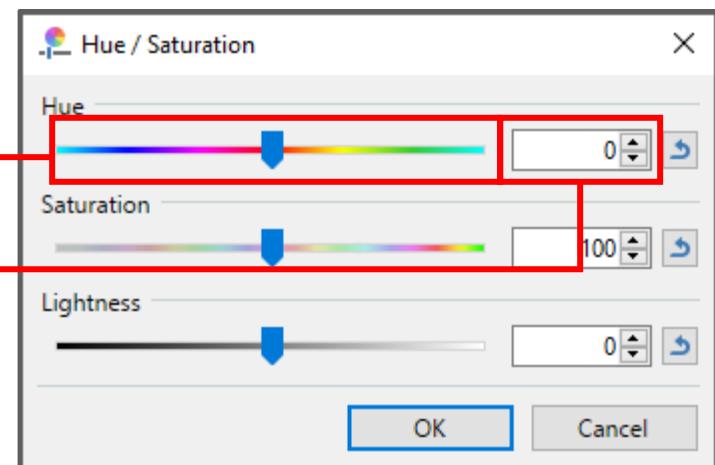
- States:
  - Selection: Index, Element
- Events:
  - Selection changed
- Examples:
  - Radio buttons
  - Choice boxes



# Logical Continuous Selection

Allows users to select one value from a continuous range of values and displays the current state.

- States:
  - Value: integer, real number
- Events:
  - Value changed
- Examples:
  - Sliders
  - Spinners

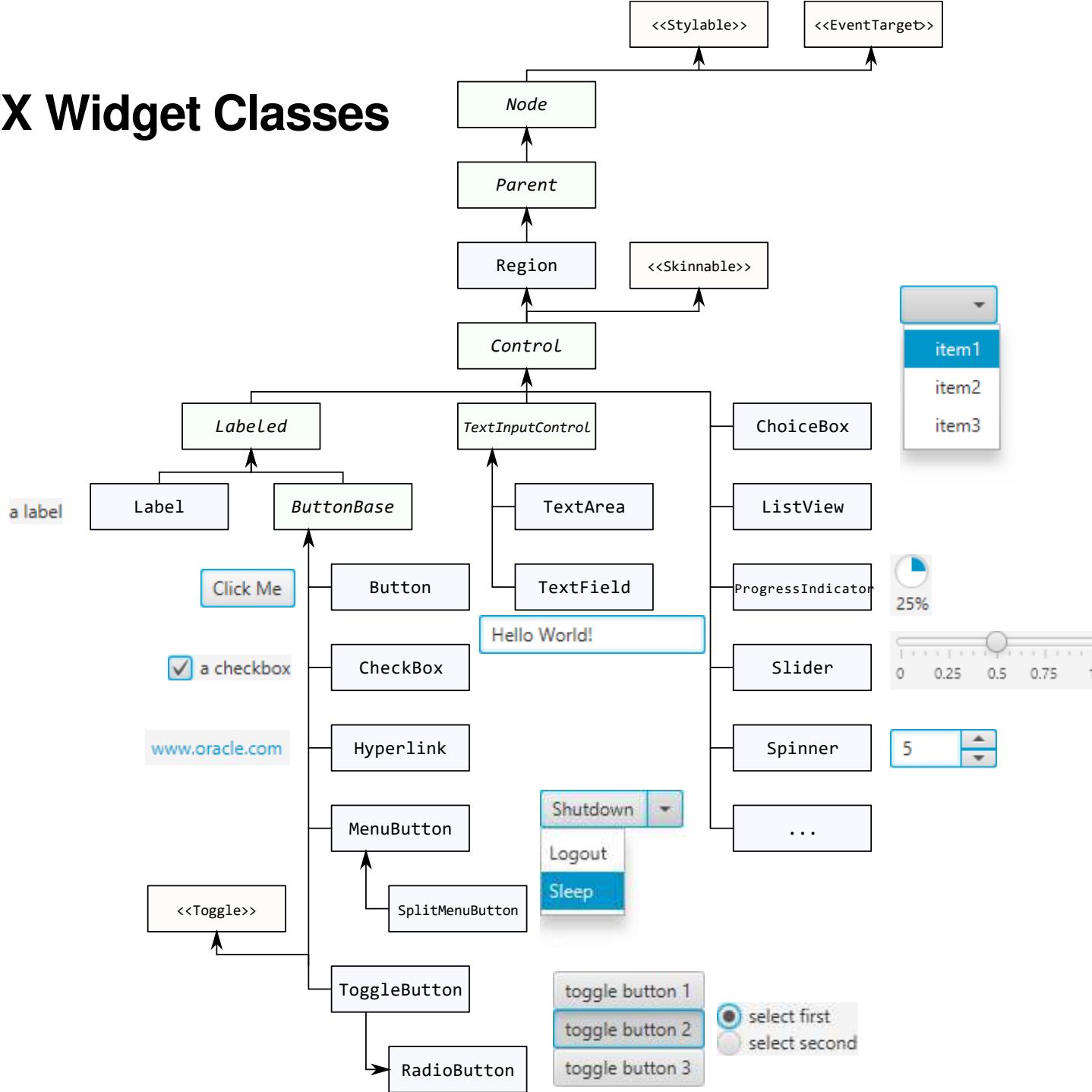


# Widgets in JavaFX

U

CS 349

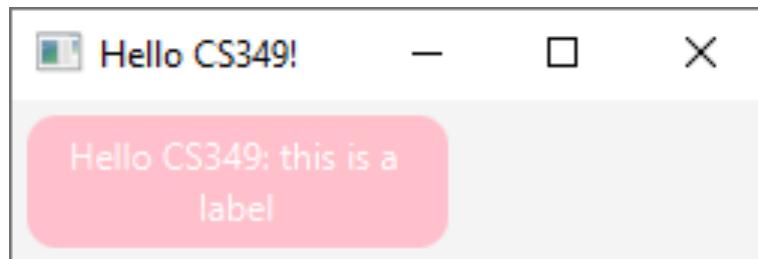
# JavaFX Widget Classes



# Display Widgets

Display widgets display some (static) information, such as, text or images. These widgets include Label and ImageView.

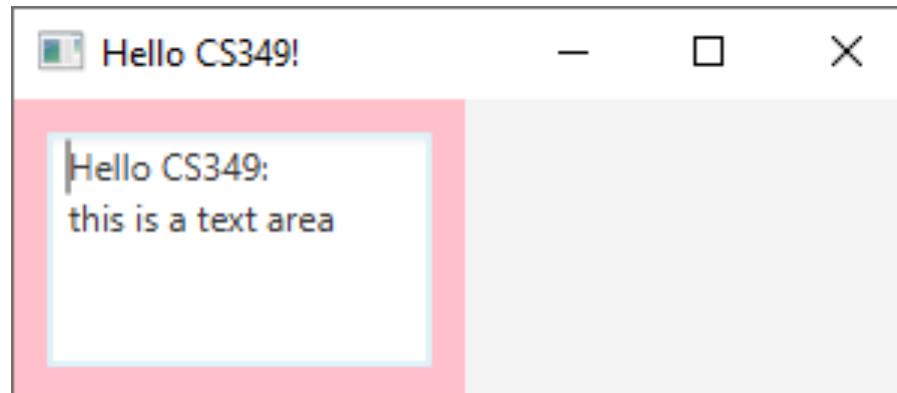
```
val myText = Label("Hello CS349: this is a label").apply {
    padding = Insets(10.0)
    background = Background(BackgroundFill(Color.PINK,
        CornerRadii(10.0),
        Insets(5.0)))
    textFill = Color.WHITE
    textAlign = TextAlign.CENTER
    isWrapText = true
    maxWidth = 150.0
}
```



# Text Entry Widgets

Text entry widgets allow users to input text. These widgets include **TextField** (single-line) and **TextArea** (multi-line).

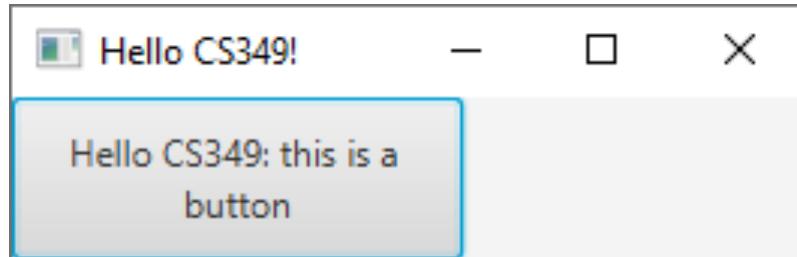
```
val myText = TextArea("Hello CS349:\nthis is a text area").apply {  
    padding = Insets(10.0)  
    background = Background(BackgroundFill(Color.PINK, null, null))  
    maxWidth = 150.0  
    maxHeight = 100.0  
    textProperty().addListener { _, _, newValue -> // String  
        stage.title = newValue  
    }  
}
```



# Button Widgets

Button widgets allow users to perform a single action. These widgets include Button and MenuItem.

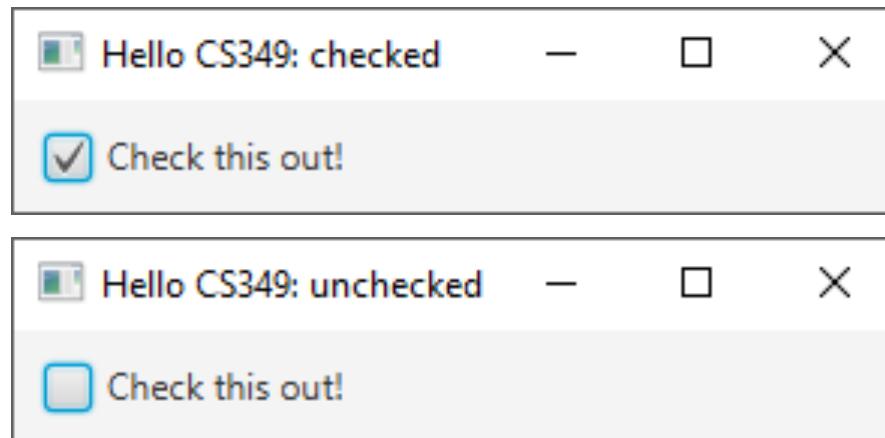
```
val myButton = Button("Hello CS349: this is a button").apply {  
    padding = Insets(10.0)  
    isWrapText = true  
    textAlignment = TextAlignment.CENTER  
    maxWidth = 150.0  
    maxHeight = 100.0  
    onAction = EventHandler {  
        stage.title = "${stage.title}!"  
    }  
}
```



# Boolean Selection Widgets

Boolean selection widgets allow users to select between two states. These widgets include CheckBox and ToggleButton.

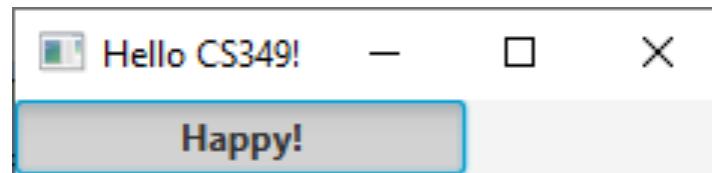
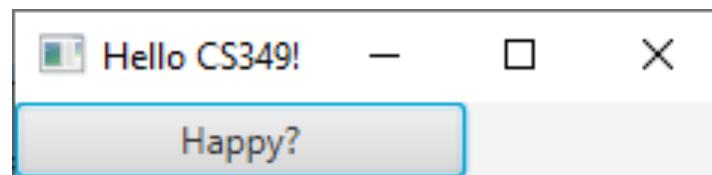
```
val myCheck = CheckBox("Check this out!").apply {
    isSelected = true
    padding = Insets(10.0)
    selectedProperty().addListener { _, _, newValue -> // Boolean
        stage.title =
            "Hello CS349: ${if (newValue.not()) "un" else ""}checked"
    }
}
```



# Boolean Selection Widgets

Boolean selection widgets allow users to select between two states. These widgets include CheckBox and ToggleButton.

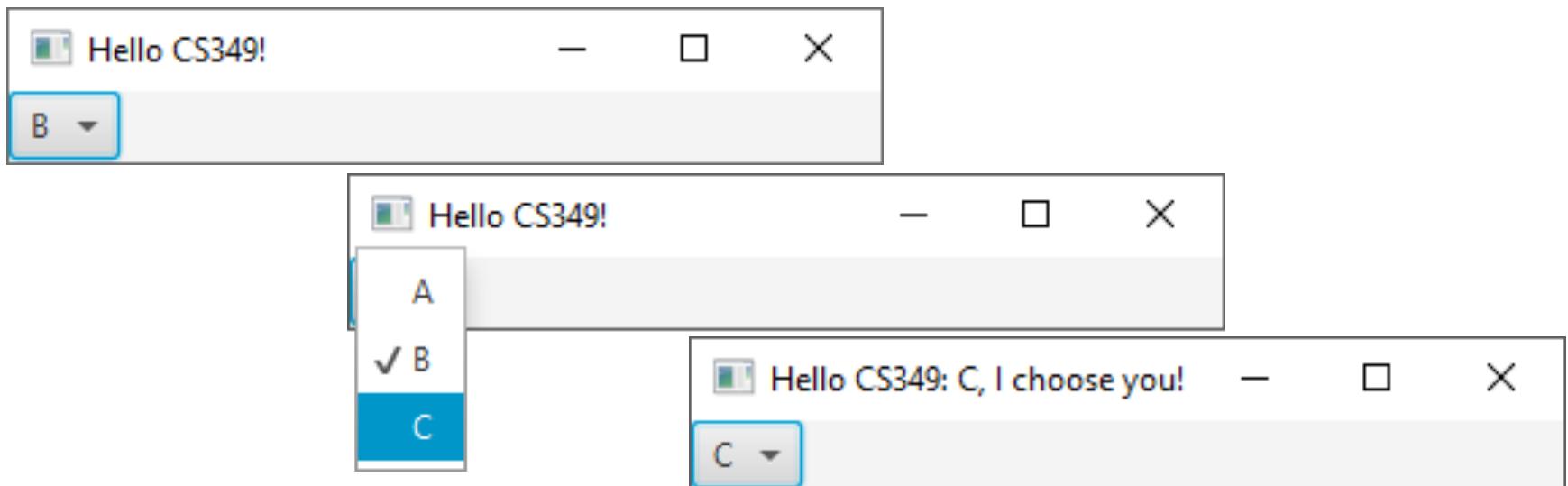
```
val myToggle = ToggleButton("Happy?").apply {
    minWidth = 150.0
    selectedProperty().addListener { _, _, newValue -> // Boolean
        text = "${text.dropLast(1)}${if (newValue) "!" else "?}"
        font = Font.font(null,
            if(newValue) FontWeight.BOLD
            else FontWeight.NORMAL,
            -1.0)
    }
}
```



# Discrete Selection Widgets

Discrete selection widgets allow users to select **one** of an arbitrary number of entries. These widgets include ChoiceBox and RadioButton.

```
val myChoice = ChoiceBox<String>().apply {
    items.addAll("A", "B", "C")
    value = items[1]
    maxWidth = 150.0
    valueProperty().addListener { _, _, newValue -> // String
        stage.title = "Hello CS349: $newValue, I choose you!"
    }
}
```

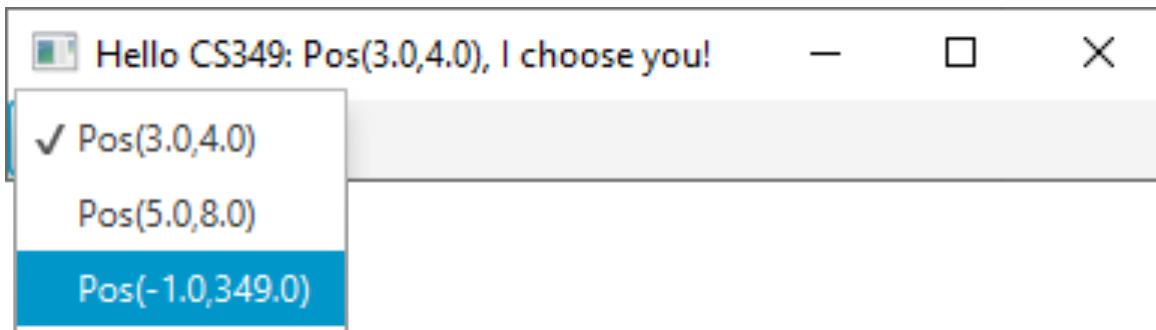


# Discrete Selection Widgets

The following approach uses the custom class Posn and accesses the underlying SelectionModel of the ChoiceBox.

```
class Posn(val x: Double, val y: Double) {
    override fun toString(): String { return "Pos($x,$y)" }
}

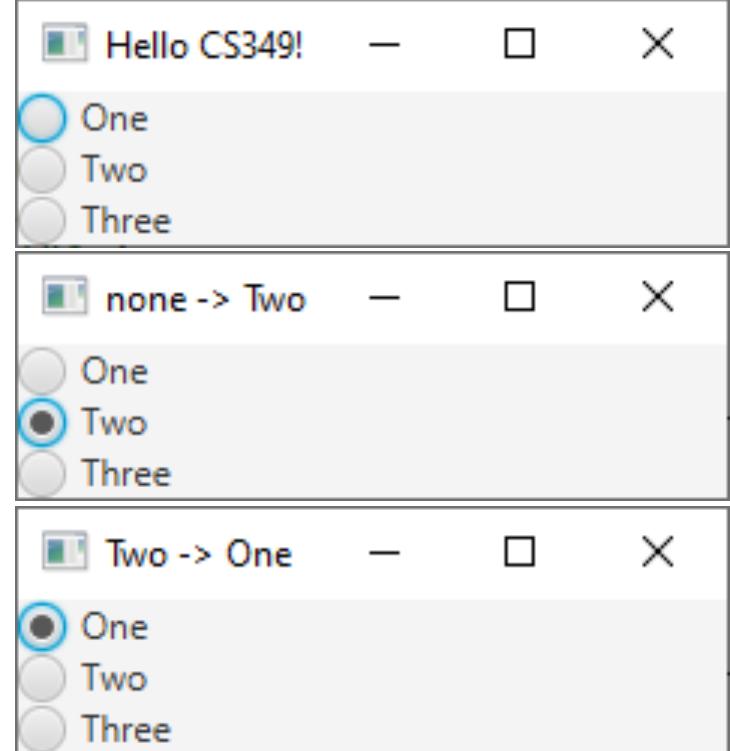
val cbItems = listOf(Posn(3.0, 4.0), Posn(5.0, 8.0), Posn(-1.0, 349.0))
val myDrop = ChoiceBox(FXCollections.observableList(cbItems)).apply {
    maxWidth = 150.0
    selectionModel.select(1)
    selectionModel.selectedItemProperty().addListener
    { _, _, newValue -> // Posn
        stage.title = "Hello CS349: $newValue, I choose you!"
    }
}
```



# Discrete Selection Widgets

Radio buttons only show the 1-of-n-behaviour if they are grouped within a ToggleGroup.

```
val myRadioA = RadioButton("One")
val myRadioB = RadioButton("Two")
val myRadioC = RadioButton("Three")
ToggleGroup().apply {
    myRadioA.toggleGroup = this;
    myRadioB.toggleGroup = this;
    myRadioC.toggleGroup = this;
    selectedToggleProperty().addListener { _, oldValue, newValue ->
        stage.title = "${(oldValue as RadioButton)?.text ?: "none"} ->
            ${(newValue as RadioButton).text}"
    }
}
```

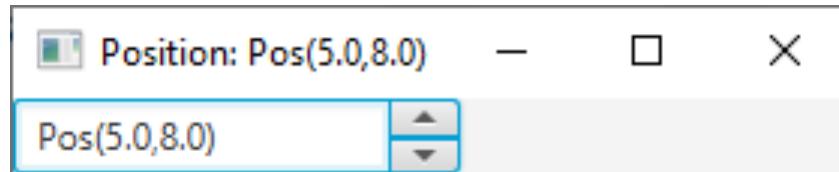


# Discrete Selection Widgets

Spinners can be used both for selection between discrete elements or selection for a continuous range of values.

```
class Posn(val x: Double, val y: Double) {
    override fun toString(): String { return "Pos($x,$y)" }
}

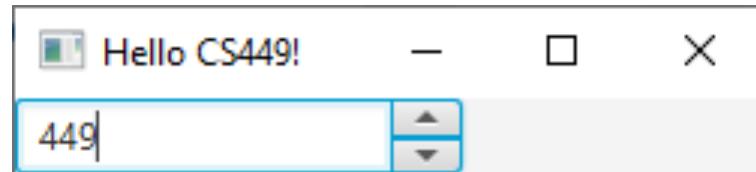
val spItems = listOf(Posn(3.0, 4.0), Posn(5.0, 8.0), Posn(-1.0, 349.0))
val mySpinner = Spinner<Posn>().apply {
    valueFactory =
        ListSpinnerValueFactory(FXCollections.observableList(spItems))
    valueFactory.value = spItems.last()
    maxWidth = 150.0
    valueProperty().addListener { _, _, newValue -> // Posn
        stage.title = "Position: ${newValue}!"
    }
}
```



# Continuous Selection Widgets

Spinners can be used both for selection between discrete elements or selection for a continuous range of values.

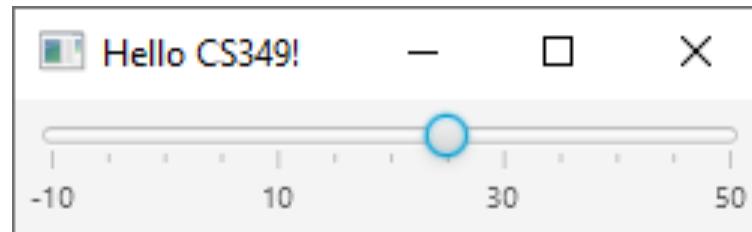
```
val mySpinner = Spinner<Double>(100.0, 499.0, 349.0).apply {  
    maxWidth = 150.0  
    isEditable = true  
    valueProperty().addListener { _, _, newValue -> // Double  
        stage.title = "Hello CS${newValue.toInt()}!"  
    }  
}
```



# Continuous Selection Widgets

Continuous selection widgets allow users to select a value from within a continuous range of values, prominently, integers or real numbers.

```
val mySlider = Slider(-10.0, 50.0, 25.0).apply {
    padding = Insets(5.0)
    isShowTickLabels = true
    isShowTickMarks = true
    isSnapToTicks = true
    majorTickUnit = 20.0
    minorTickCount = 3
    minWidth = 250.0
    valueProperty().addListener { _, _, newValue -> // Double
        stage.title = "${newValue.toInt()}"
    }
}
```



-



X

# Properties & Property Binding

U

CS 349

# Properties

In JavaFX, a **property is a special type of class member**, that

- stores a value that controls the appearance or behaviour of a widget
- can be set manually or programmatically
- can have a **listener attached**
- **can be bound to a property of another class**, so that when one changes, the other is changed automatically

# “Binding” Properties via Listener

We can react to state changes by listening to the corresponding property (here: `valueProperty`) and updating another field (here: `stage.title`) manually.

```
override fun start(stage: Stage) {  
  
    val myChoice = ChoiceBox<String>().apply {  
        items.addAll("A", "BB", "CCC")  
        value = items[1]  
        valueProperty().addListener { _, _, newValue ->  
            stage.title = newValue"  
        }  
    }  
  
    stage.apply {  
        scene = Scene(Pane(myChoice), 300.0, 200.0)  
        title = "Hello CS349!"  
    }.show()  
}
```



# Binding Properties Directly

Alternatively, we can bind two properties together (here: `myChoice.valueProperty` to `stage.titleProperty`). If the first one changes, the other is automatically updated.

```
override fun start(stage: Stage) {  
  
    val myChoice = ChoiceBox<String>().apply {  
        items.addAll("A", "BB", "CCC")  
        value = items[1]  
        stage.titleProperty().bind(valueProperty())  
    }  
  
    stage.apply {  
        scene = Scene(Pane(myChoice), 300.0, 200.0)  
    }.show()  
}
```

If the data does not need to modified, we can use 1:1 binding.



# Binding Properties via Custom Binding

If the data needs to modified, we have to create bindings manually (here: a binding that concatenates two strings).

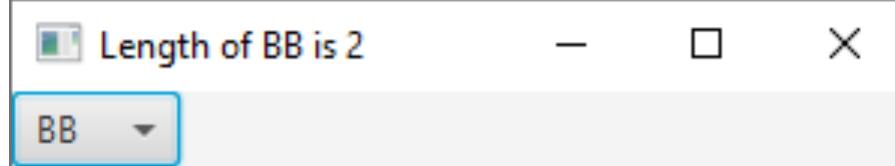
```
override fun start(stage: Stage) {  
  
    val myChoice = ChoiceBox<String>().apply {  
        items.addAll("A", "BB", "CCC")  
        value = items[1]  
        stage.titleProperty().bind(Bindings.concat("Selected string: ",  
                                         valueProperty()))  
    }  
  
    stage.apply {  
        scene = Scene(Pane(myChoice), 300.0, 200.0)  
    }.show()  
}
```



# Binding Properties via Custom Binding

If the data needs to modified, we have to create bindings manually (here: a binding that concatenates multiple strings and converts `valueProperty` into an `Int`).

```
override fun start(stage: Stage) {  
  
    val myChoice = ChoiceBox<String>().apply {  
        items.addAll("A", "BB", "CCC")  
        value = items[1]  
        stage.titleProperty().bind(Bindings.concat(  
            "Length of ",  
            valueProperty(),  
            " is ",  
            Bindings.createIntegerBinding({ valueProperty().value.length },  
                valueProperty())))  
    }  
  
    stage.apply {  
        scene = Scene(Pane(myChoice), 300.0, 200.0)  
    }.show()  
}
```



# End of the Chapter



Please make sure to

- Be aware of the difference between logic inputs and widgets
- Have a rough understanding about the differences of widgets that implement the same logic input
- Remember which widgets are available
- Properties exist, they can be bound together



# Layouts

Types of Layouts

Layouts in JavaFX

Designing UI Components using Layouts

U

CS 349

January 25

-



X

U

CS 349

# Types of Layouts

# Dynamic Layout

Applications need to be able to adjust the presentation of our interfaces. We need to dynamically reposition and resize our content in response to:

- Change in screen **resolution** (e.g., different computers or devices).
- **Resizing** the application window (e.g., user adjustments).

The image displays three side-by-side screenshots of a CBC news website, illustrating how the layout adapts to different screen widths.

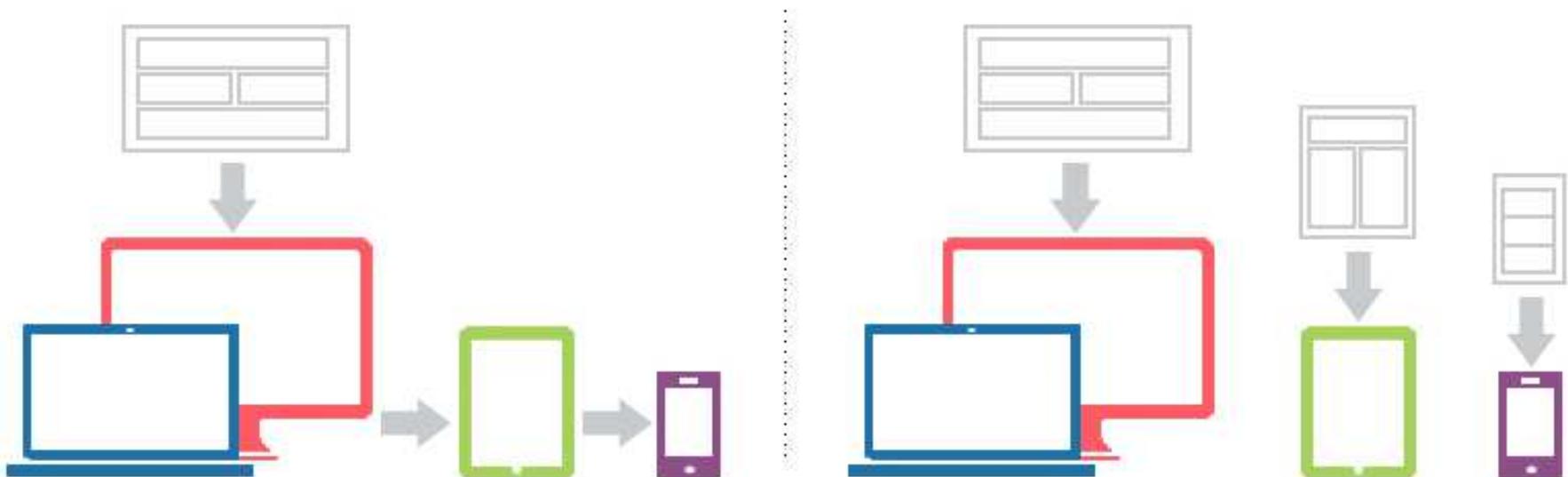
- Left Screenshot (Desktop View):** Shows a detailed news article about Fiona's impact in Port aux Basques. The main headline is "Help arrives in Port aux Basques, and residents displaced by Fiona are 'just trying to keep things together'". Below it is a large image of damaged houses. To the left, there's a sidebar with a "Popular Now" section and several smaller news items. At the bottom, there are video thumbnails and a "LIVE" stream banner.
- Middle Screenshot (Tablet View):** Shows the same news article, but the layout is more compact. The "Popular Now" section and sidebar items are reduced in size and moved to the right. The main image remains prominent.
- Right Screenshot (Mobile View):** Shows the news article in a very compact form. The main headline and image are at the top. Below them, there's a single "Popular Now" item, and the rest of the content is significantly reduced in size and complexity.

In all three views, the top navigation bar remains consistent with links for NEWS, Top Stories, Local, Climate, World, Canada, Politics, Indigenous, Opinion, The National, and More. A search bar and sign-in link are also present in the top right.

# Responsive vs. Adaptive Layouts

Two strategies for layout:

- **Responsive**: support a universal design that **reflows spatial layout** to fit the dimensions of the current view (device or window).
- **Adaptive**: design **optimized spatial layouts** for each of your devices, and dynamically switch to fit devices



# Responsive Layout

We are going to focus on responsive layout: adapting to changes **dynamically**.

To dynamically adjust content to a window, we want to:

- maximize use of available space for displaying widgets,
- while maintaining consistency with spatial layout, and
- preserving the visual quality of spatial layout

This requires that our application can dynamically adjust elements:

- re-allocate space for widgets
- adjust location and size of widgets
- perhaps change visibility, look, and / or feel of widgets

# Compositing Interfaces

Container nodes (or layouts) describe how their children should be placed.

Different (layout) containers have different strategies for handling layout. For example, Group lets the designer position children directly, while StackPane tries to re-position its children.

In a responsive layout, the container is responsible for setting the size and location of its children according to its internal rules.

-



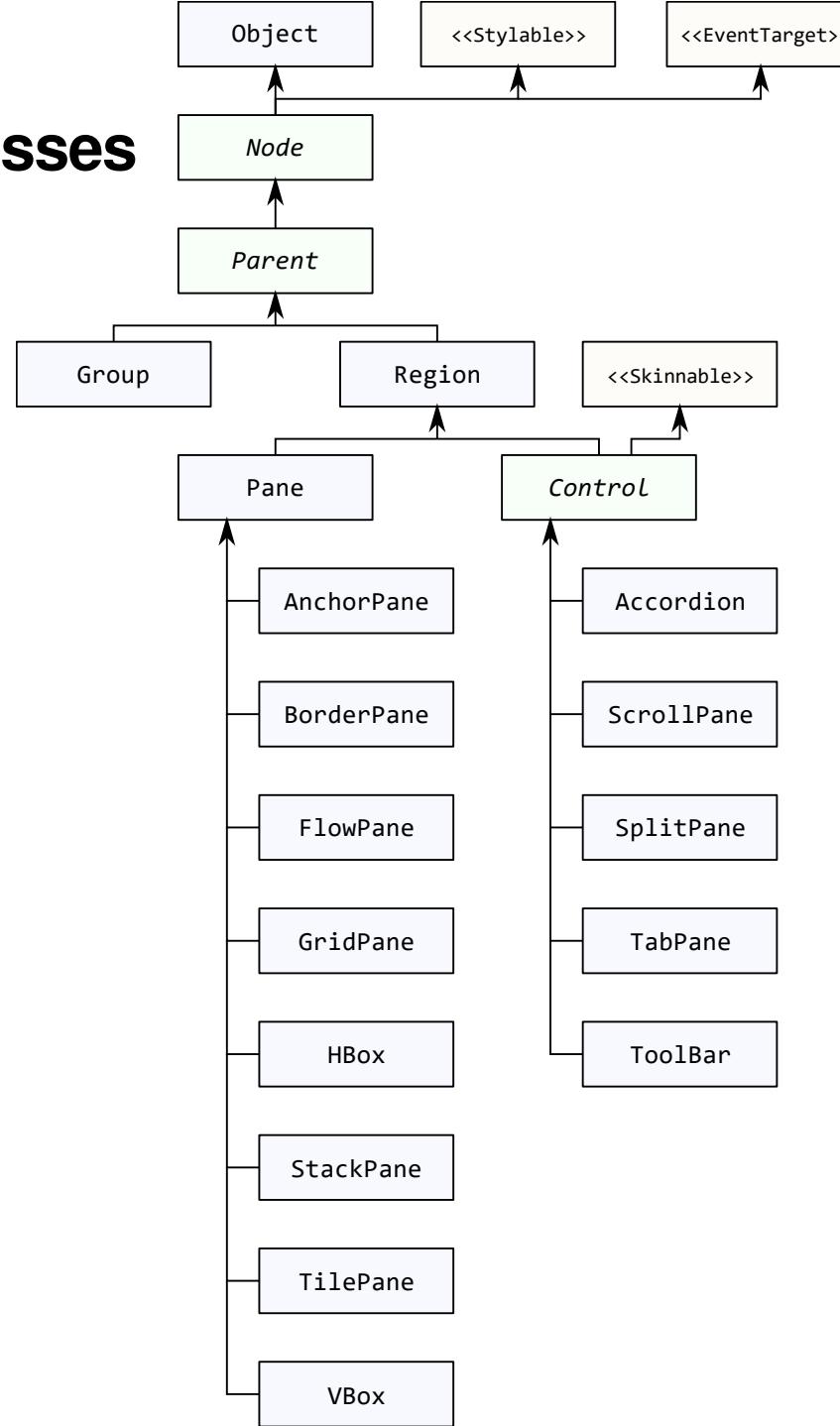
X

# Layouts in JavaFX

U

CS 349

# JavaFX Container Classes



# Layout Strategies

- Fixed Layout: non-resizable
- Variable Intrinsic Layout: adjusting widget size and position
- Relative Layout: positioning components relative to one another
- Custom Layout: define your own!

# Fixed Layouts

The layout does not move or resize nodes by itself. You need to manually specify location and position of all nodes within the layout.

This is most suitable for cases when you have a fixed-size window.

Containers that support fixed layout:

- Group
- Pane

## Container – Group

A Group contains children that are rendered in order whenever this node is rendered. It will take on **the collective bounds of its children and is not directly resizable**. Any transform, effect, or state applied to a Group will be applied to all children of that group.



# Container – Group

```
override fun start(stage: Stage) {  
    val bt1 = Button("I am a button!").apply {  
        minWidth = 300.0  
        rotate = 45.0  
    }  
    val bt2 = Button("I am a button, too!").apply {  
        minWidth = 300.0  
        translateX = 349.0  
        translateY = 42.0  
    }  
    val root = Group(bt1, bt2).apply { rotate = -10.0 }  
  
    stage.apply {  
        title = "Hello, CS349!"  
        scene = Scene(root, 800.0, 100.0).apply { fill = Color.PINK }  
    }.show()  
}
```



## Container – Pane

A Pane can be used directly in cases where absolute positioning of children is required since it does not perform layout beyond resizing resizable children to their preferred sizes. It is the application's responsibility to position the children since the pane leaves the positions alone during layout.



# Container – Pane

```
override fun start(stage: Stage) {  
    val bt1 = Button("I am a button!").apply {  
        minWidth = 300.0  
        rotate = 45.0  
    }  
    val bt2 = Button("I am a button, too!").apply {  
        minWidth = 300.0  
        translateX = 349.0  
        translateY = 42.0  
    }  
    val root = Pane(bt1, bt2).apply { rotate = -10.0 }  
  
    stage.apply {  
        title = "Hello, CS349!"  
        scene = Scene(root, 800.0, 100.0).apply { fill = Color.PINK }  
    }.show()  
}
```



# Variable Intrinsic Layouts

The layout attempts to use the widget's preferred sizes, but queries all widgets first, and allocates space to them as a group.

Layout determined in two-passes (bottom-up, top-down):

- Get each child widget's preferred size (includes recursively asking all of its children for their preferred size...)
- Decide on a layout that satisfies everyone's preferences, then iterate through each child, and set its layout (size / position)

Containers that support variable intrinsic layout:

- `VBox`
- `HBox`
- `FlowPane`

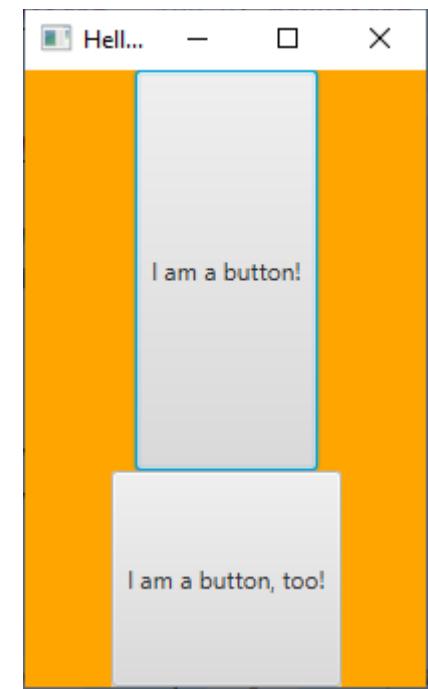
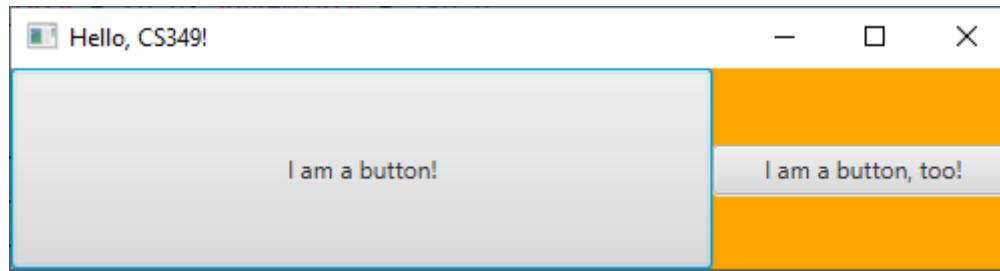
# Container – HBox, VBox

{HBox|VBox} lays out its children in a single {row|column}.

They will resize children (if resizable) to their preferred {widths|heights} and use its {fillHeight|fillWidth} property to determine whether to resize their {heights|widths} to fill its own.

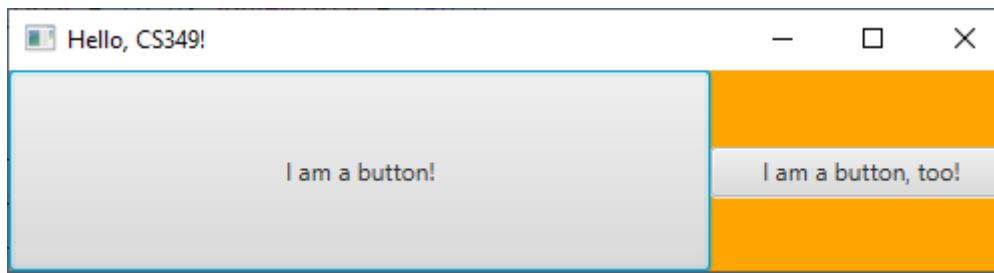
The alignment of the content is controlled by the alignment property.

If an {HBox|VBox} is resized larger than its preferred {width|height}, it will by default leave the extra space unused. To have one or more children be allocated that extra space it may **optionally set an {hgrow|vgrow}** constraint on the child.



# Container – HBox, VBox

```
override fun start(stage: Stage) {  
    val bt1 = Button("I am a button!").apply {  
        minWidth = 60.0; prefWidth = 100.0;  
        maxWidth = Double.MAX_VALUE; maxHeight = Double.MAX_VALUE }  
    val bt2 = Button("I am a button, too!").apply {  
        minWidth = 40.0; prefWidth = 150.0 }  
    val root = HBox(bt1, bt2).apply {  
        background = Background(BackgroundFill(Color.ORANGE, null, null))  
        alignment = Pos.CENTER  
        isFillHeight = true }  
  
    HBox.setHgrow(bt1, Priority.ALWAYS)  
    HBox.setHgrow(bt2, Priority.NEVER)  
  
    stage.apply {  
        title = "Hello, CS349!"  
        scene = Scene(root, 500.0, 100.0)  
    }.show()  
}
```

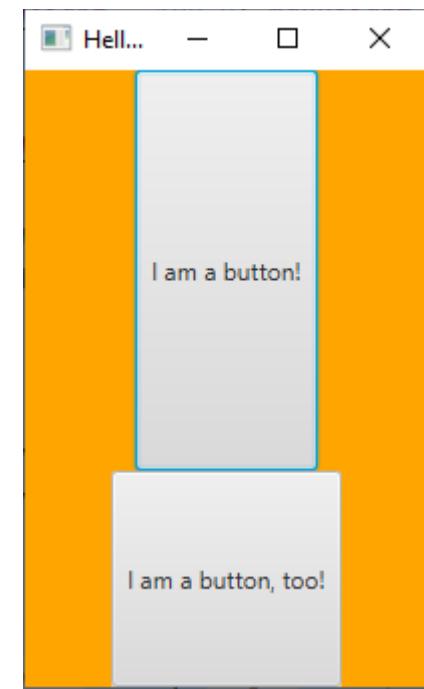


# Container – HBox, VBox

```
override fun start(stage: Stage) {  
    val bt1 = Button("I am a button!").apply {  
        minHeight = 60.0; prefHeight = 100.0; maxHeight = 200.0 }  
  
    val bt2 = Button("I am a button, too!").apply {  
        minHeight = 40.0; prefHeight = 100.0; maxHeight = 200.0 }  
    val root = VBox(bt1, bt2).apply {  
        background = Background(BackgroundFill(Color.ORANGE, null, null))  
        alignment = Pos.CENTER }
```

```
VBox.setVgrow(bt1, Priority.ALWAYS)  
VBox.setVgrow(bt2, Priority.SOMETIMES)
```

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



# Widget Dimensions

Widgets need to be flexible in size and position

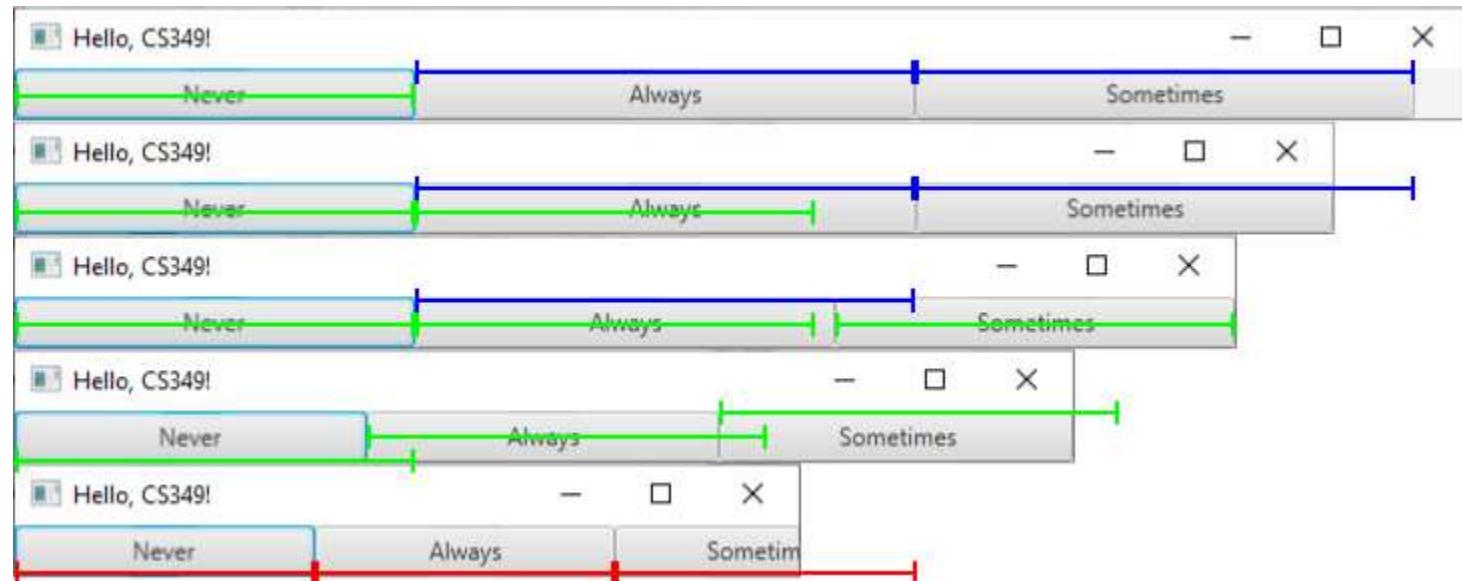
- Widgets store their own position and width / height, but containers have the ability to change these properties.
- Other properties may also be changed by containers (e.g., reducing font size for a caption)

Widgets give the layout algorithm a range of preferred values as “**hints**”, and containers consider the size hints of nodes in determining layout.

- **min**: parent should not resize node's {width|height} smaller than this value
- **pref**: parent should treat this value as the node's ideal {width|height}
- **max**: parent should not resize node's {width|height} larger than this value

# Widget Dimensions

```
val root = HBox(Button("Never").apply {
    minWidth = 150.0; prefWidth = 200.0; maxWidth = 250.0
    HBox.setHgrow(this, Priority.NEVER)
}, Button("Always").apply {
    minWidth = 150.0; prefWidth = 200.0; maxWidth = 250.0
    HBox.setHgrow(this, Priority.ALWAYS)
}, Button("Sometimes").apply {
    minWidth = 150.0; prefWidth = 200.0; maxWidth = 250.0
    HBox.setHgrow(this, Priority.SOMETIMES)
})
```



# Container – FlowPane

FlowPane lays out its children in a flow that wraps at the flowpane's boundary. A {horizontal|vertical} flowpane will layout nodes in {rows|columns}, wrapping at the flowpane's {width|height}.

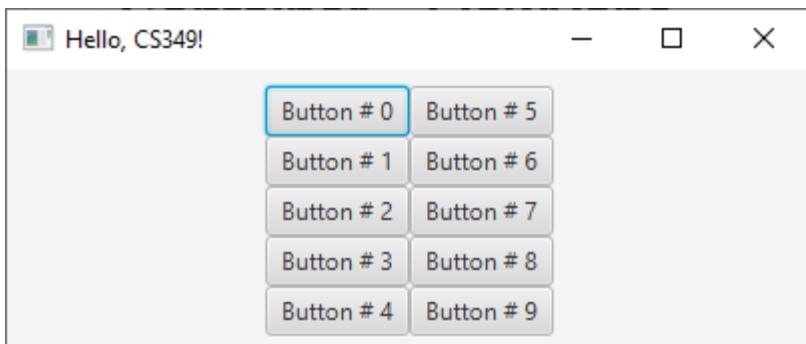
The `prefWrapLength` property establishes its preferred {width| height}.

The `alignment` property controls how the {rows|columns} are aligned within the bounds of the flowpane.



# Container – FlowPane

```
override fun start(stage: Stage) {  
    val root = FlowPane(Orientation.VERTICAL).apply {  
        (0..9).forEach() { children.add(Button("Button # $it")) }  
        alignment = Pos.CENTER  
    }  
  
    stage.apply {  
        title = "Hello, CS349!"  
        scene = Scene(root, 400.0, 200.0)  
    }.show()  
}
```



# Relative Layouts

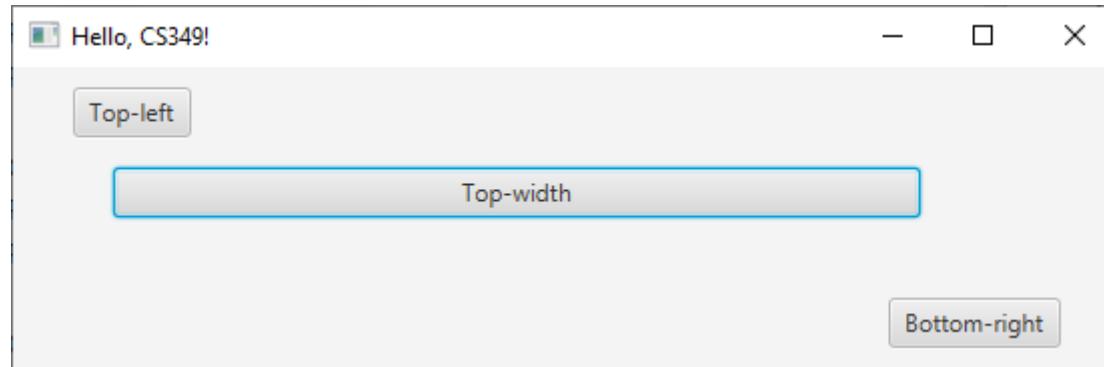
The layout constrains child positions into a **specific layout**.

Containers that support relative layout:

- AnchorPane
- BorderPane
- GridPane
- TilePane

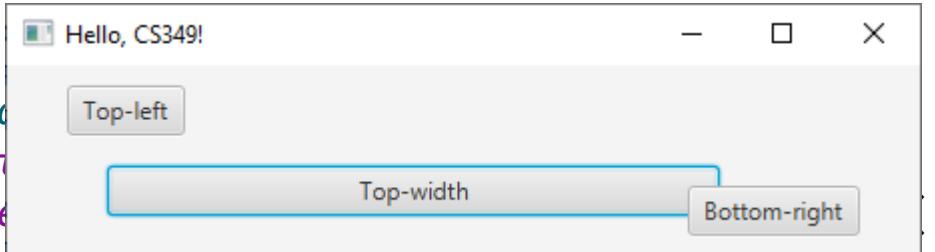
# Container – AnchorPane

AnchorPane allows the edges of child nodes to be anchored to an offset from the anchor pane's edges.



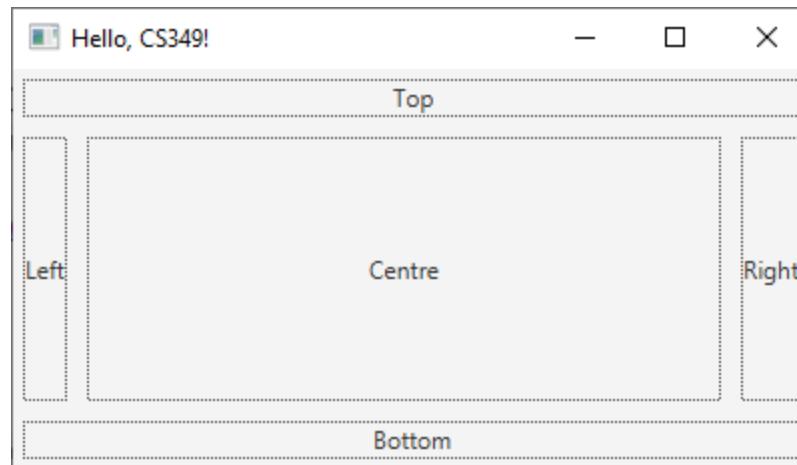
# Container – AnchorPane

```
override fun start(stage: Stage) {  
    val root = AnchorPane().apply {  
        children.add(0, Button("Top-left").apply {  
            AnchorPane.setTopAnchor(this, 10.0)  
            AnchorPane.setLeftAnchor(this, 30.0)  
        })  
        children.add(1, Button("Bottom-right").apply {  
            AnchorPane.setBottomAnchor(this, 10.0)  
            AnchorPane.setRightAnchor(this, 30.0)  
        })  
        children.add(0, Button("Top-width").apply {  
            AnchorPane.setTopAnchor(this, 50.0)  
            AnchorPane.setLeftAnchor(this, 50.0)  
            AnchorPane.setRightAnchor(this, 100.0)  
        })  
    }  
    stage.c  
    ti  
    sce  
}
```



# Container – BorderPane

BorderPane lays out children in top, left, right, bottom, and center positions. The top and bottom children will be resized to their preferred heights and extend the width of the border pane. The left and right children will be resized to their preferred widths and extend the length between the top and bottom nodes. And the center node will be resized to fill the available space in the middle. Any of the positions may be null.



# Container – BorderPane

```
override fun start(stage: Stage) {
    val lc = Label("Centre").apply { maxHeight = Double.MAX_VALUE;
                                         maxWidth = Double.MAX_VALUE }
    val lt = Label("Top").apply { maxWidth = Double.MAX_VALUE }
    val lb = Label("Bottom").apply { maxWidth = Double.MAX_VALUE }
    val ll = Label("Left").apply { maxHeight = Double.MAX_VALUE }
    val lr = Label("Right").apply { maxHeight = Double.MAX_VALUE }

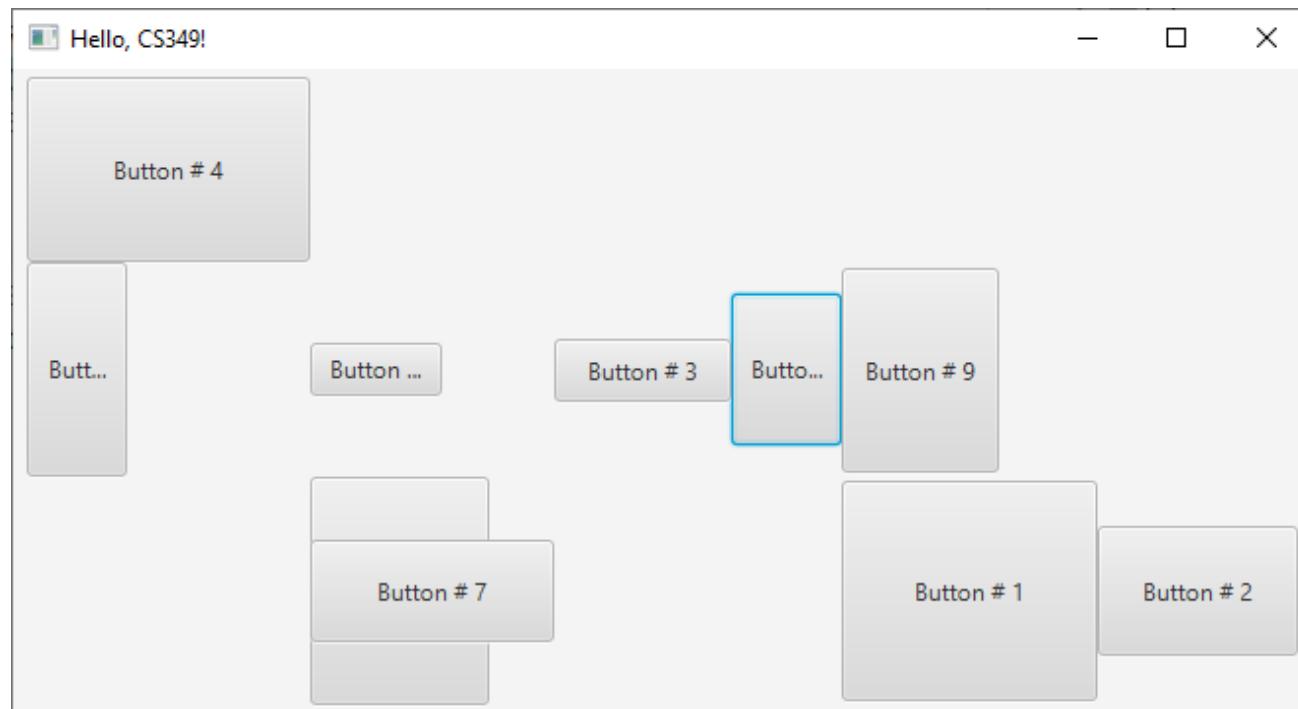
    val root = BorderPane(lc, lt, lr, lb, ll)

    root.children.forEach {
        (it as Label)
        BorderPane.setAlignment(it, Pos.CENTER)
        BorderPane.setMargin(it, Insets(5.0))
        it.alignment = Pos.CENTER;
        it.border= Border(BorderStroke(Color.BLACK,
                                         BorderStrokeStyle.DOTTED, null, BorderStroke.THIN))
    }

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(root , 400.0, 200.0).apply { fill = Color.PINK }
    }.show()
}
```

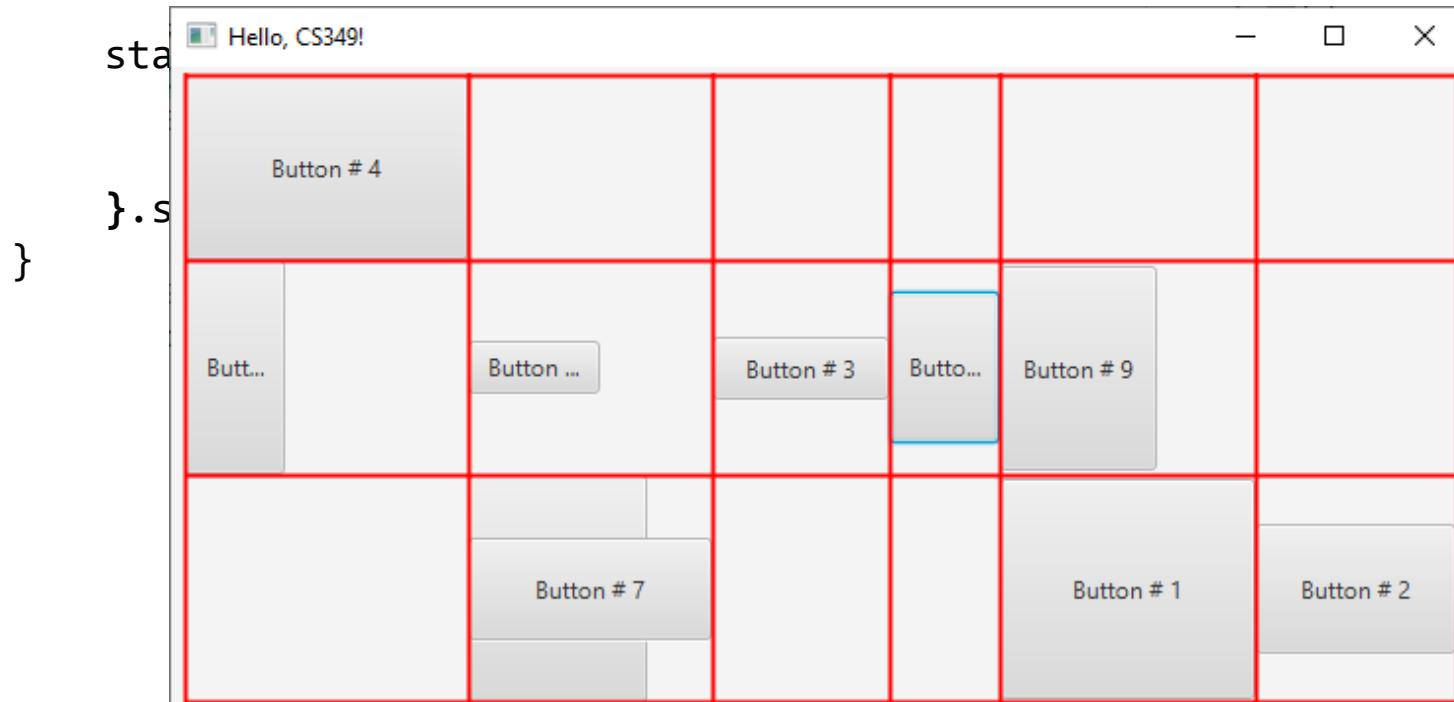
# Container – GridPane

GridPane lays out its children within a flexible grid of rows and columns. A child may be placed anywhere within the grid and may span multiple rows / columns. Children may freely overlap within rows / columns and their stacking order will be defined by the order of the gridpane's children list.



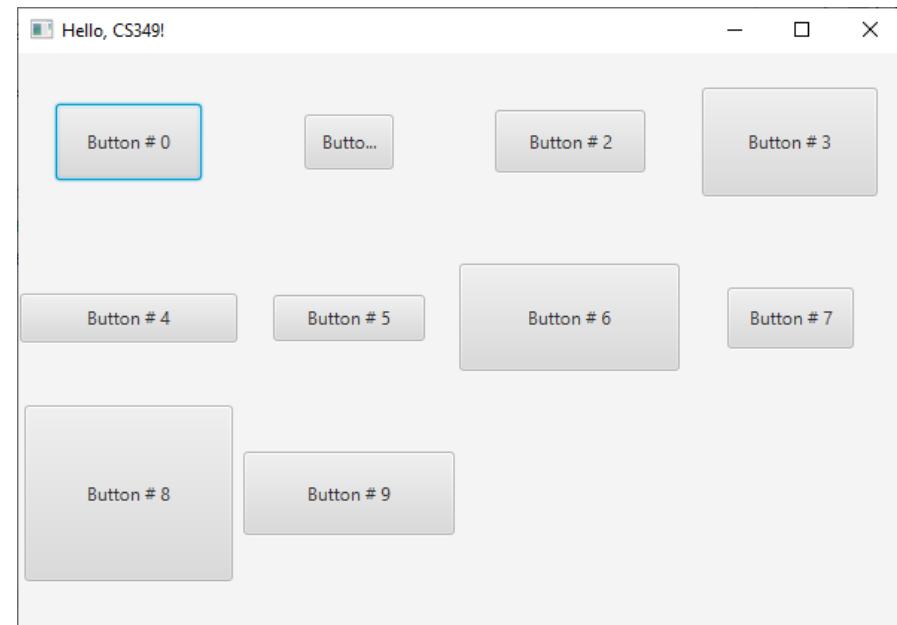
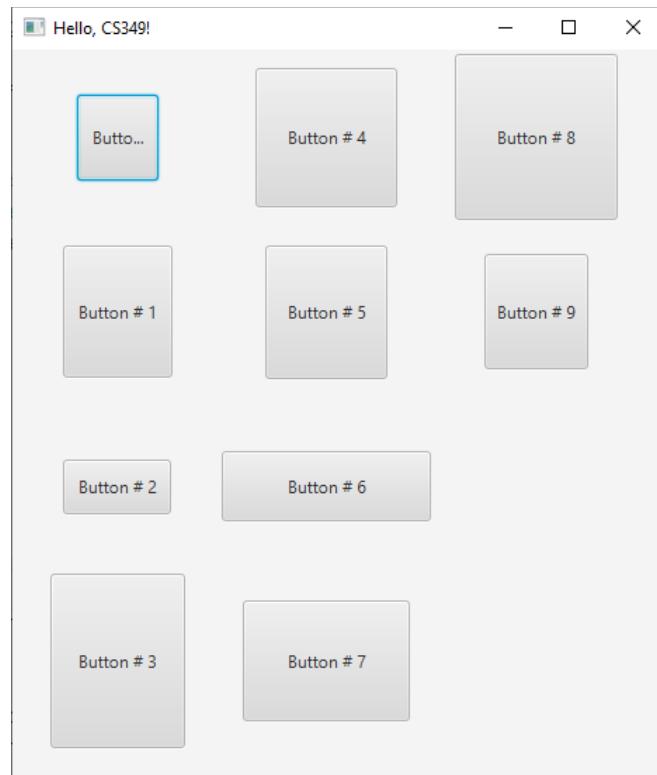
# Container – GridPane

```
override fun start(stage: Stage) {  
    val root = GridPane().apply {  
        (0..9).forEach() { add(Button("Button # $it")).apply {  
            prefWidth = Random.nextDouble() * 100 + 50  
            prefHeight = Random.nextDouble() * 100 + 25  
        }, Random.nextInt(0, 6), Random.nextInt(0, 3)) }  
        alignment = Pos.CENTER  
    }  
}
```



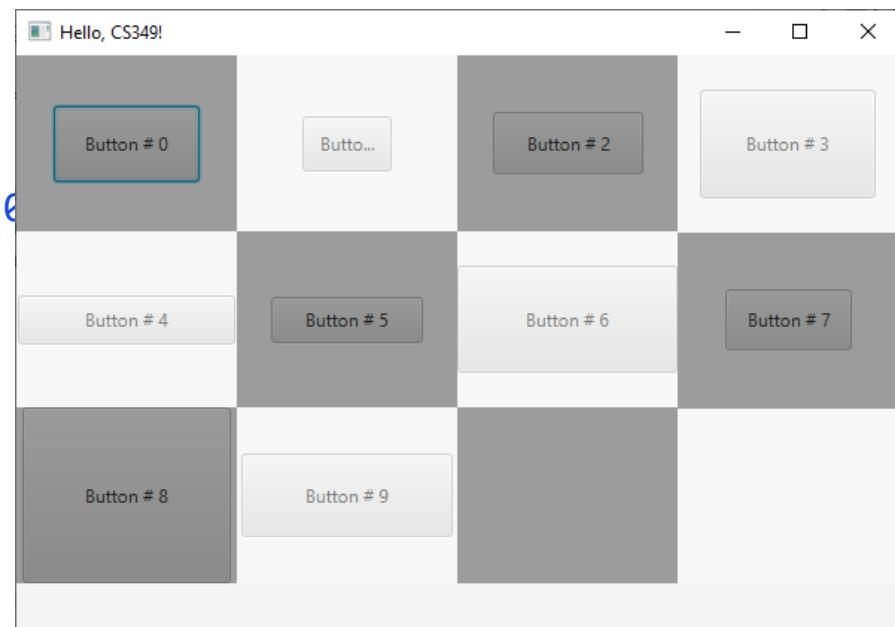
# Container – TilePane

TilePane lays out its children in a grid of uniformly sized "tiles". A {horizontal|vertical} tilepane will tile nodes in {rows|columns}, wrapping at the tilepane's {width|height}.



# Container – TilePane

```
override fun start(stage: Stage) {  
    val root = TilePane(Orientation.HORIZONTAL).apply {  
        (0..9).forEach() { children.add(Button("Button # $it")).apply {  
            prefWidth = Random.nextDouble() * 100 + 50  
            prefHeight = Random.nextDouble() * 100 + 25  
        })}  
        alignment = Pos.TOP_LEFT  
        prefRows = 3  
    }  
  
    stage.apply {  
        title = "Hello, CS349!"  
        scene = Scene(root, 400.0)  
    }.show()  
}
```

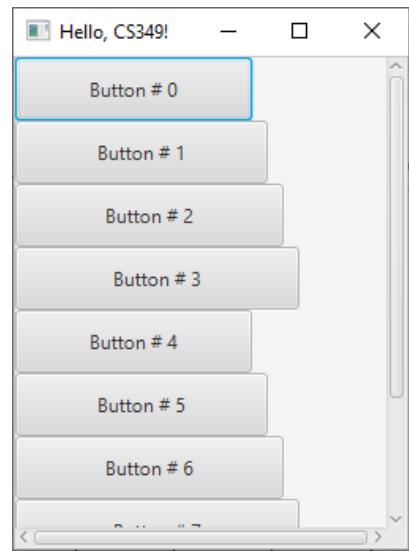


## **Container – ScrollPane**

ScrollPane provides a scrolled, clipped viewport of its contents. It allows the user to scroll the content around either directly (panning) or by using scroll bars. It allows specification of the scroll bar policy, which determines when scroll bars are displayed: always, never, or only when they are needed.

# Container – ScrollPane

```
override fun start(stage: Stage) {  
    val root = VBox().apply {  
        (0..9).forEach() { children.add(Button("Button # $it")  
            .prefWidth = 150.0 + it * 10 % 40  
            .prefHeight = 40.0  
            .maxHeight = Double.MAX_VALUE)  
        }  
        VBox.setVgrow(this, Priority.ALWAYS)  
    })}  
}  
  
val scroll = ScrollPane(root).apply {  
    hbarPolicy = ScrollPane.ScrollBarPolicy.ALWAYS  
    isFitToWidth = true  
}  
  
stage.apply {  
    title = "Hello, CS349!"  
    scene = Scene(scroll, 250.0, 500.0)  
}.show()  
}
```



# Container – ScrollPane

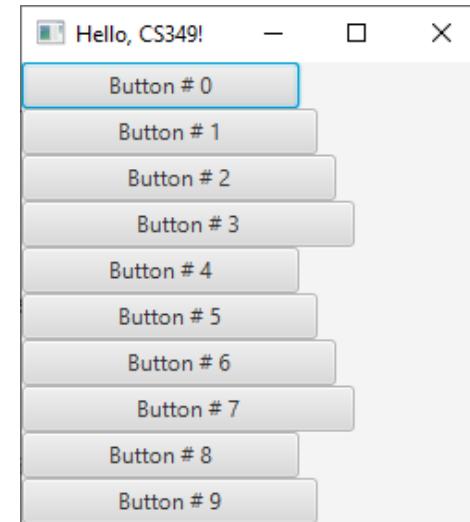
VBox only  
enough vertical space  
Vgrow NEVER



VBox only  
enough vertical space  
Vgrow ALWAYS



VBox only  
not enough vertical space



# Container – ScrollPane

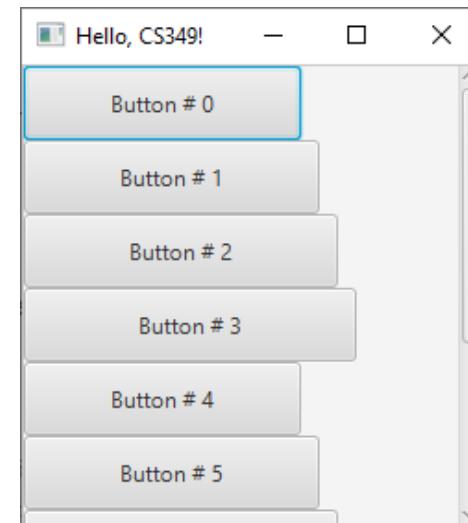
VBox in ScrollPane  
enough vertical space  
Vgrow NEVER



VBox in ScrollPane  
enough vertical space  
Vgrow ALWAYS



VBox in ScrollPane  
not enough vertical space



## **Container – TabPane**

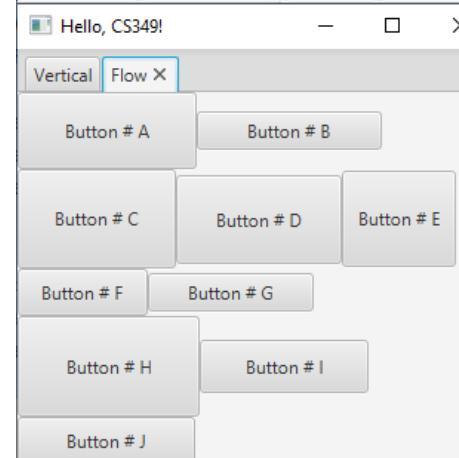
TabPane allows switching between a group of tabs, with only one tab visible at a time. Its tabs can be positioned at any of its four sides. Its default {height|width} will be determined by the largest content {height|width}.

# Container – TabPane

```
override fun start(stage: Stage) {
    val vboxTab = VBox().apply {
        (0..9).forEach() { children.add(Button("Button # $it")).apply {
            prefWidth = Random.nextDouble() * 75 + 100
            prefHeight = Random.nextDouble() * 50 + 25
        })}
    }
    val flowTab = FlowPane().apply {
        ('A'..'J').forEach() { children.add(Button("Button # $it")).apply {
            prefWidth = Random.nextDouble() * 50 + 75
            prefHeight = Random.nextDouble() * 50 + 25
        })}
    }

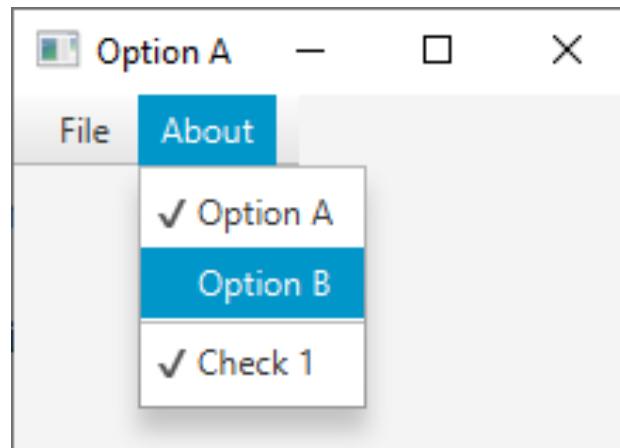
    val tab = TabPane().apply {
        tabs.add(Tab("Vertical", vboxTab))
        tabs.add(Tab("Flow", flowTab))
        tabsClosingPolicy = TabPane.TabClosingPolicy.ALL_TABS
    }

    stage.apply {
        title = "Hello, CS349!"
        scene = Scene(tab, 400.0, 400.0).apply { fill = Color.PINK }
    }.show()
}
```



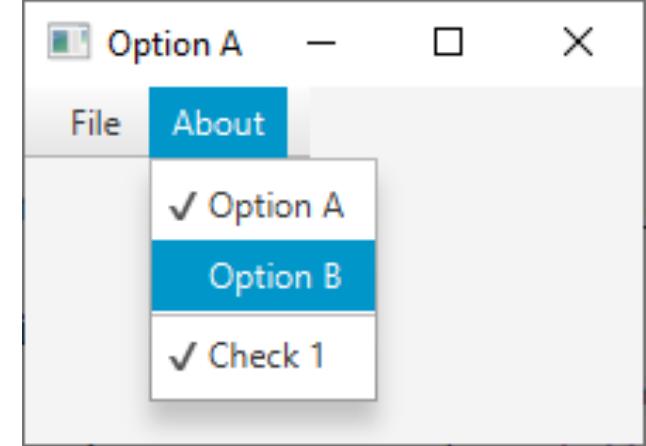
## Container – MenuBar

MenuBar is traditionally placed at the very top of the user interface and embedded within it are Menus. By default, for each menu added to the menu bar, it will be represented as a button with the Menu text value displayed.



# Container – MenuBars

```
val menuBar = MenuBar()
menuBar.menus.addAll(
    Menu("File").apply {
        items.add(MenuItem("Quit").apply {
            onAction = EventHandler { Platform.exit() }
        })
    },
    Menu("About").apply {
        val rm1 = RadioMenuItem("Option A")
        val rm2 = RadioMenuItem("Option B")
        val cm = CheckMenuItem("Check 1").apply {
            selectedProperty().addListener { _, _, new ->
                stage.title = "$text ${if (new) "on" else "off"}"
            }
        }
        items.addAll(rm1, rm2, SeparatorMenuItem(), cm)
        ToggleGroup().apply {
            rm1.toggleGroup = this; rm2.toggleGroup = this
            selectToggle(rm1)
            selectedToggleProperty().addListener { _, _, new ->
                stage.title = (new as RadioMenuItem).text
            }
        }
    }
)
```



# End of the Chapter



Please make sure to

- Remember which widgets are available



Any further questions?

-



# CS349: User Interfaces

Architectural Patterns for User Interfaces

Implementing MVC

Model–View–ViewModel

U

CS 349

January 30

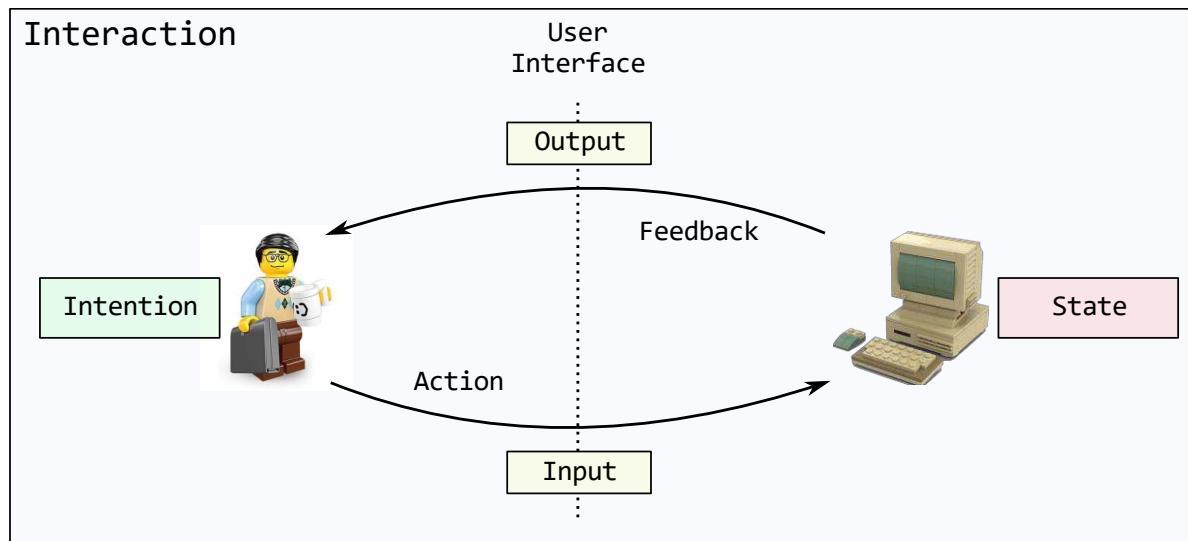
# Architectural Patterns for User Interfaces

U | CS 349

# Recall: User Interaction

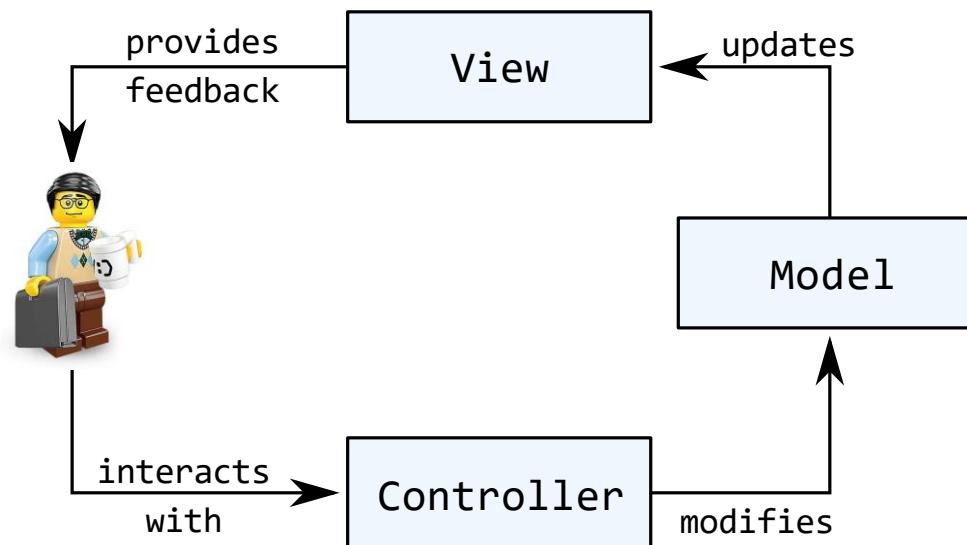
People typically interact with technology to reach a goal.

They determine what they need to do, and perform actions to the UI, which gives feedback in return.



# User Interaction

We often design our systems to directly support this style of interaction. This architectural pattern is called *Model–View–Controller (MVC)*.



# Model-View-Controller (MVC)

Developed at Xerox PARC in 1979 by Trygve Reenskaug for Smalltalk-80, the precursor to Java.

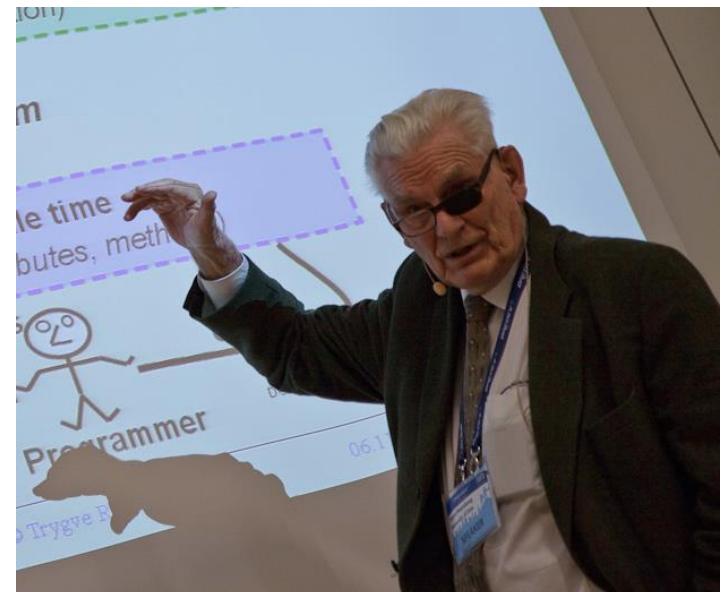
MVC used to be the standard design pattern for GUIs.

Used at many levels

- Overall application design
- Individual components

Many variations of MVC:

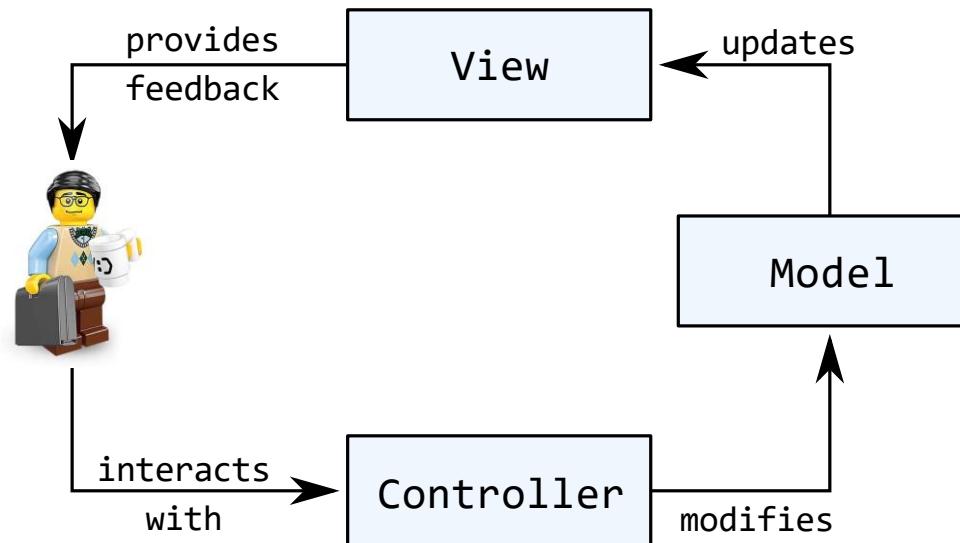
- Model–View–Presenter
- Model–View–Adapter
- Hierarchical Model–View–Controller



# Why use MVC?

Separation of “business logic” and the user-interface logic.

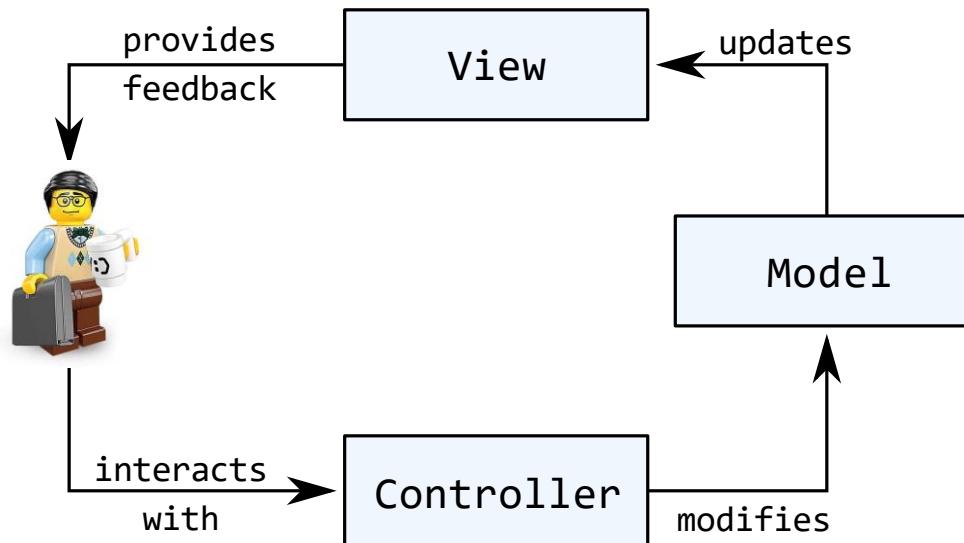
- View and Controller can be changed without changing the underlying Model.
- This is useful when
  - adding support for a new input device (e.g., voice control, touchscreen)
  - adding support for a new type of output device (e.g., different size screen)



# Why use MVC?

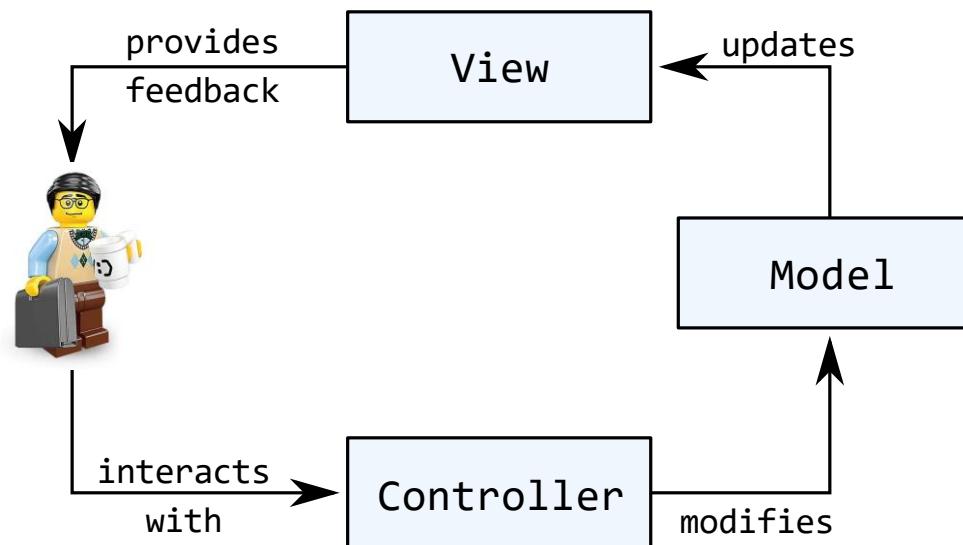
Supports multiple views of the underlying data / model. This is useful when

- viewing numeric data as a table, a line graph, a pie chart, etc.
- displaying simultaneous “overview” and “detail” views
- enabling “edit” and “preview” views



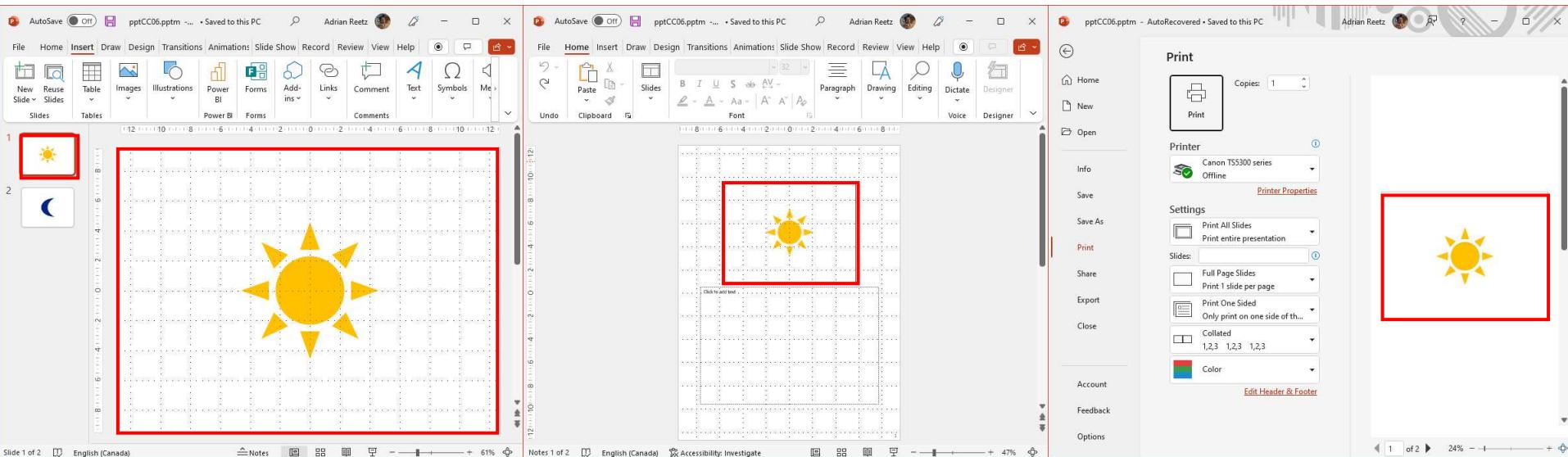
# Why use MVC?

Separation of concerns in code (code reuse, ease of testing)



# Why use MVC?

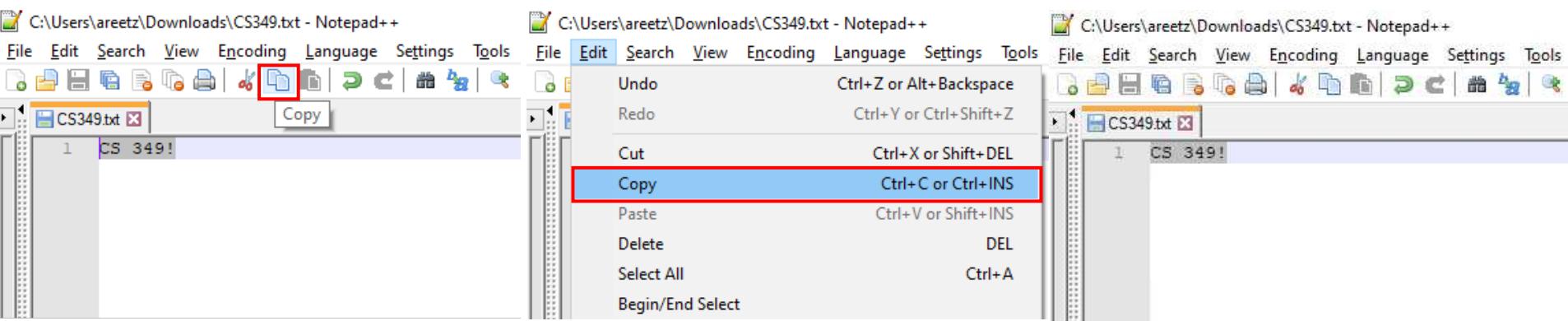
Four views of the same data:



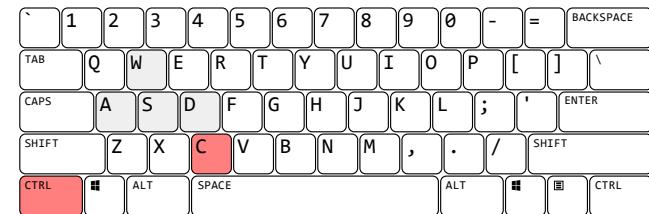
When one view changes, the others should change as well.

# Why use MVC?

Three controllers for the same action:



All controllers should use the same functionality.



U

CS 349

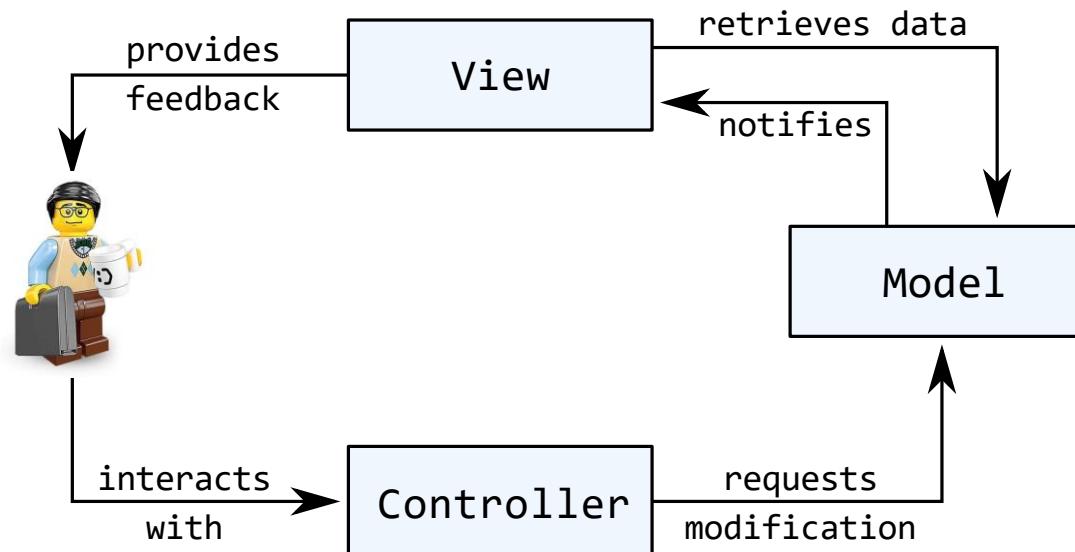
# Implementing MVC

# MVC Implementation

MVC implementations typically uses the Observer design pattern.

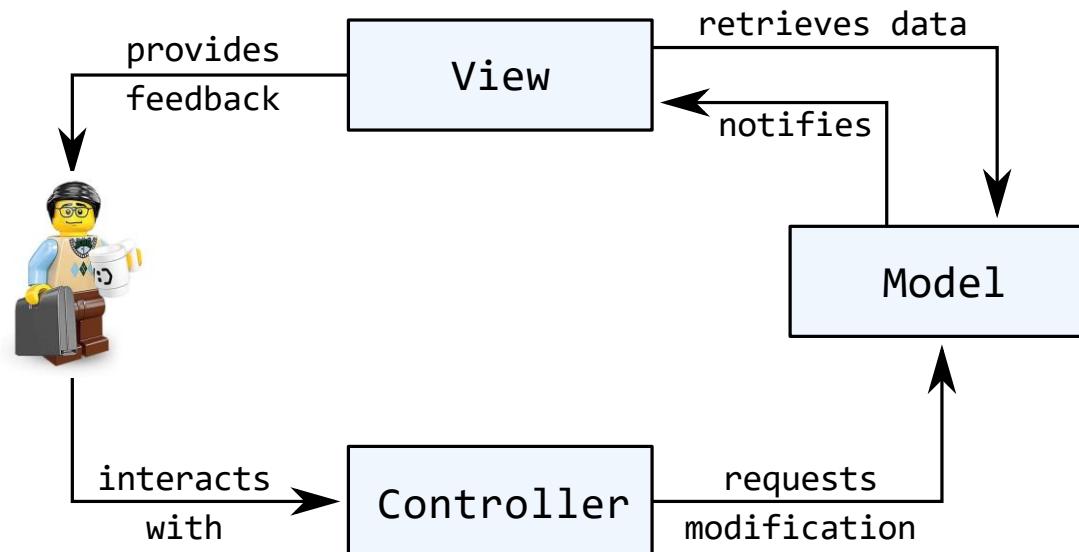
Interface architecture is decomposed into three parts:

- **Model**: manages system state and its modification
- **View**: manages interface to provide feedback
- **Controller**: manages interaction to request system state modification



# MVC Implementation

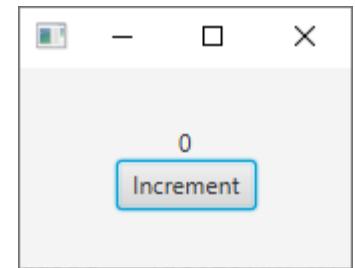
1. User performs action on Controller.
2. Controller asks Model to act upon input event.
3. Model might change state and notify View that state change occurred.
4. View retrieves updated state from Model and visualizes it.
5. User analyzes new state of the system based on feedback from View.



# No MVC

No separation between Model, View, and Controller

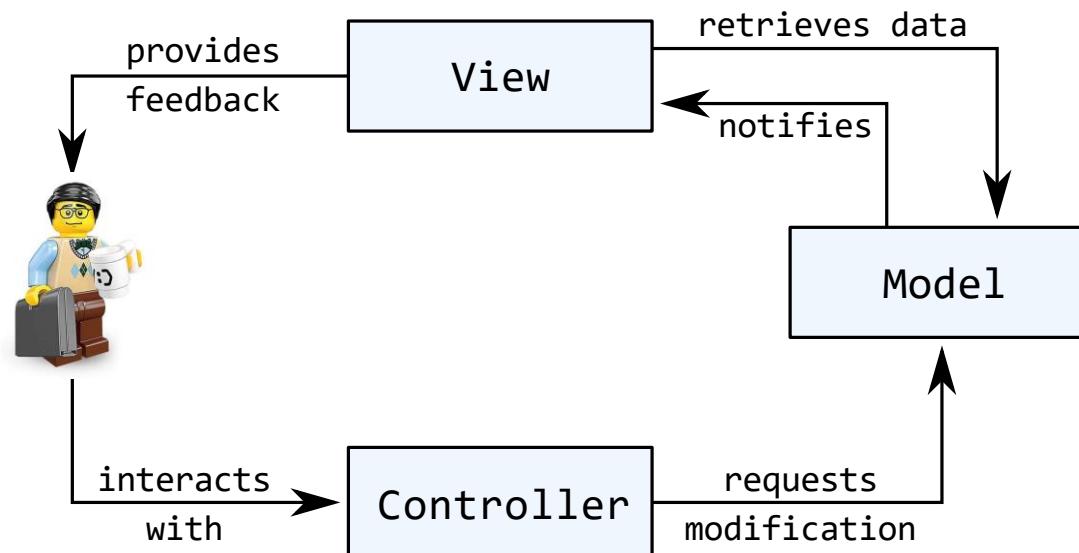
```
class NoMVC: VBox() {  
    init {  
        val countLbl = Label(0.toString())  
        children.addAll(  
            countLbl,  
            Button("Increment").apply {  
                onAction = EventHandler {  
                    countLbl.text = (countLbl.text.toInt() + 1).toString()  
                }  
            })  
        alignment = Pos.CENTER  
    }  
}
```



# MVC Implementation

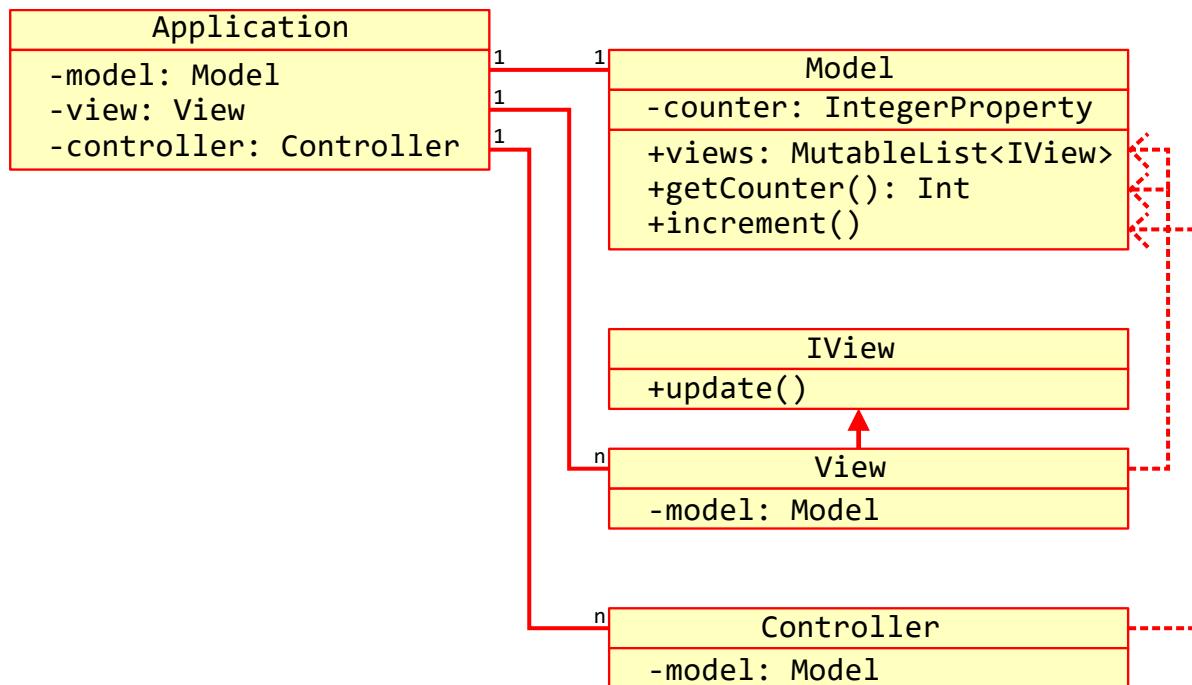
Interface architecture consisting of three parts:

- **Model**: stores system state and manages its modification
- **View**: presents system state and feedback
- **Controller**: allows for input to manipulate system state



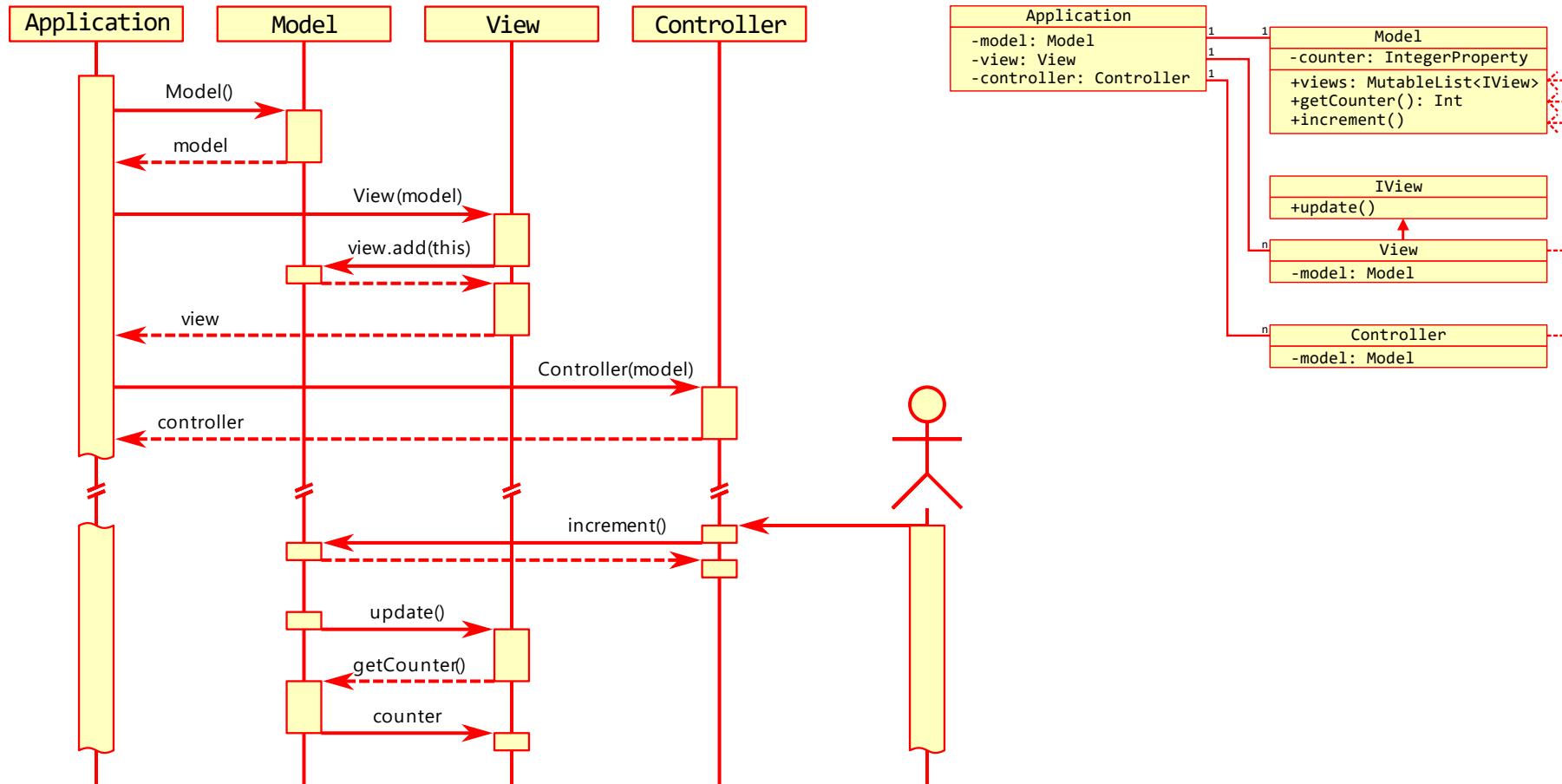
# With MVC – Conceptually

Maximal separation between Model, View, and Controller



# With MVC – Conceptually

Maximal separation between Model, View, and Controller

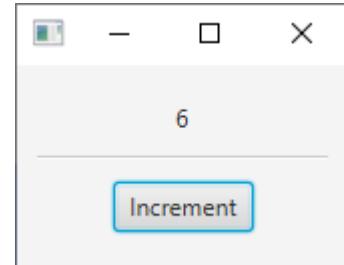


# With MVC – Conceptually

```
class Model {  
  
    private var counter = 0  
  
    val views = mutableListOf<IView>()  
  
    fun increment() {  
        ++counter  
        views.forEach { it.update() }  
    }  
  
    fun getCounter(): Int {  
        return counter  
    }  
  
}  
  
class Controller(model: Model): Button("Increment") {  
    init {  
        onAction = EventHandler {  
            model.increment()  
        }  
    }  
}
```

```
interface IVView {  
    fun update()  
}  
  
class View(private val model: Model): Label(), IVView {  
  
    init {  
        model.views.add(this)  
        update()  
    }  
  
    override fun update() {  
        text = model.getCounter().toString()  
    }  
}
```

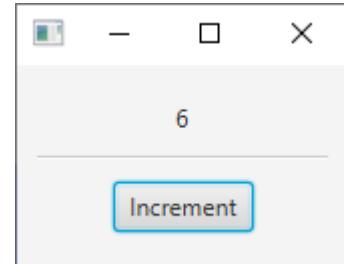
```
override fun start(stage: Stage) {  
  
    val model = Model()  
    val view = View(model)  
    val ctrl = Controller(model)  
    stage.apply {  
        title = "Hello, CS349!"  
        scene = Scene(Vbox(view, Separator(), ctrl),  
                     165.0, 100.0)  
    }.show()  
}
```



# With MVC – Conceptually

```
class Model {  
  
    private var counter = 0 // data, here just a single integer  
  
    val views = mutableListOf<IVView>() // list of all views that listen to a change in data  
  
    fun increment() { // function that allows for a controller to manipulate the model  
        ++counter  
        views.forEach { it.update() } // notifies all listeners that the state of the model has changed  
    }  
  
    fun getCounter(): Int { // function that allows for views to retrieve the data  
        return counter  
    }  
  
}  
  
class Controller(model: Model): Button("Increment") {  
    init {  
        onAction = EventHandler {  
            model.increment()  
        }  
    }  
}
```

```
interface IVView {  
    fun update()  
}  
  
class View(private val model: Model): Label(), IVView {  
  
    model.views.add(this)  
}  
  
    override fun update() {  
        text = model.counter.toString()  
    }  
  
}  
  
    override fun start(stage: Stage) {  
        val model = Model()  
        val view = View(model)  
        val ctrl = Controller(model)  
        stage.apply {  
            title = "Hello, CS349!"  
            scene = Scene(Vbox(view, Separator(), ctrl),  
                         165.0, 100.0)  
        }.show()  
    }  
}
```

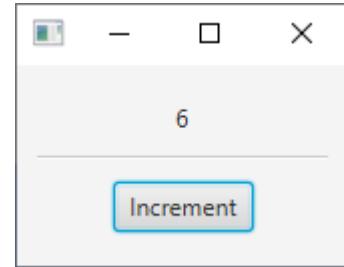


# With MVC – Conceptually

```
class Model {  
  
    private var counter = 0  
  
    val views = mutableListOf<IView>()  
  
    fun increment() {  
        ++counter  
        views.forEach { it.update() }  
    }  
  
    fun getCounter(): Int {  
        return counter  
    }  
  
}  
  
class Controller(model: Model): Button("Increment") {  
    init {  
        onAction = EventHandler {  
            model.increment()  
        }  
    }  
}
```

```
interface IVView { // interface for all views  
    fun update() // function that the model calls  
} // every time its state changes  
  
class View(private val model: Model): Label(), IVView {  
  
    init {  
        model.views.add(this) // adds this view as  
        update() // listener to the model  
    }  
  
    override fun update() {  
        text = model.getCounter().toString()  
    }  
}
```

```
override fun start(stage: Stage) {  
  
    val model = Model()  
    val view = View(model)  
    val ctrl = Controller(model)  
    stage.apply {  
        title = "Hello, CS349!"  
        scene = Scene(Vbox(view, Separator(), ctrl),  
                     165.0, 100.0)  
    }.show()  
}
```

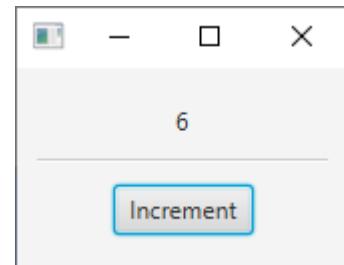


# With MVC – Conceptually

```
class Model {  
  
    private var counter = 0  
  
    val views = mutableListOf<IView>()  
  
    fun increment() {  
        ++counter  
        views.forEach { it.update() }  
    }  
  
    fun getCounter(): Int {  
        return counter  
    }  
  
}  
  
class Controller(model: Model): Button("Increment") {  
    init {  
        onAction = EventHandler {  
            model.increment() // call to the model after a user has performed an action  
        }  
    }  
}
```

```
interface IVView {  
    fun update()  
}  
  
class View(private val model: Model): Label(), IVView {  
  
    init {  
        model.views.add(this)  
        update()  
    }  
  
    override fun update() {  
        text = model.getCounter().toString()  
    }  
}
```

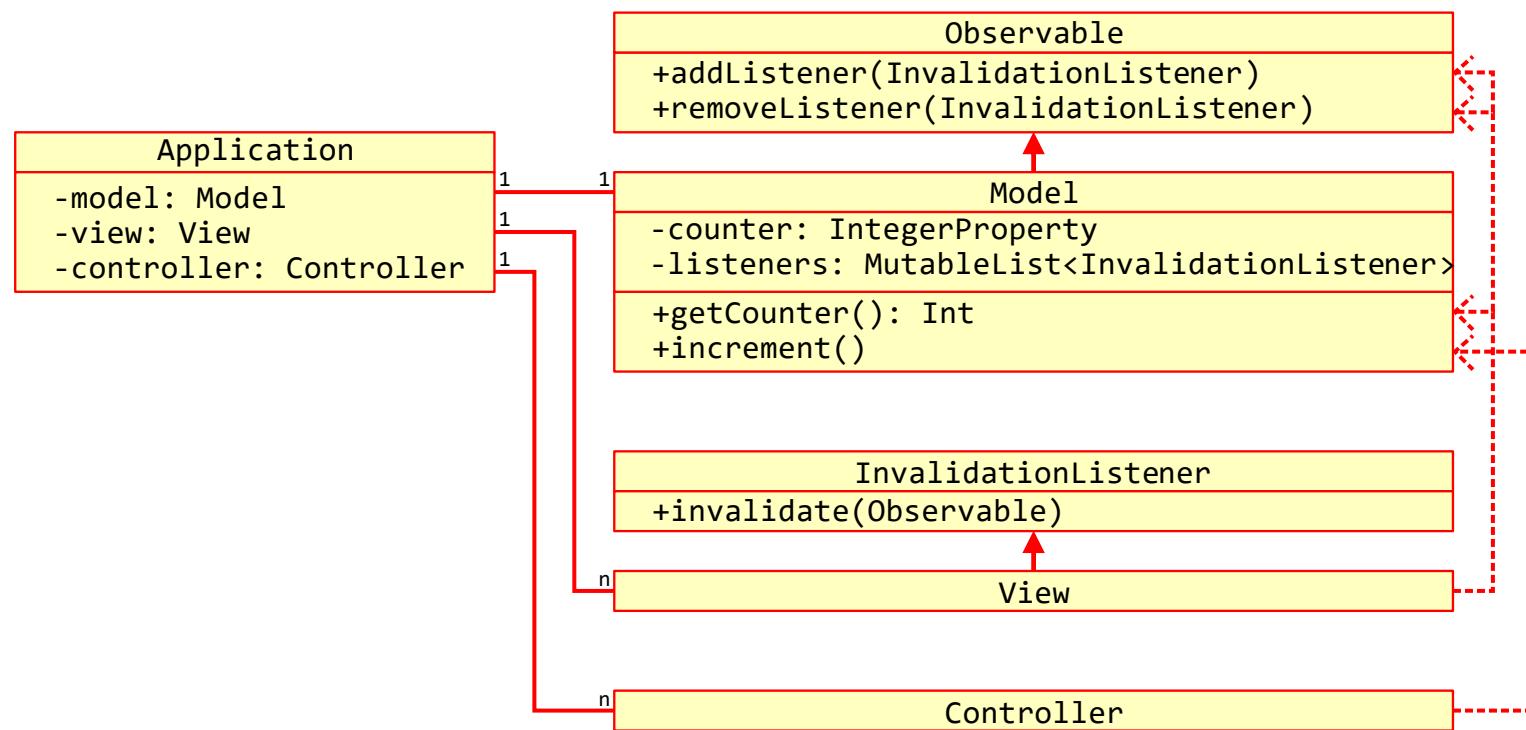
```
override fun start(stage: Stage) {  
  
    val model = Model()  
    val view = View(model)  
    val ctrl = Controller(model)  
    stage.apply {  
        title = "Hello, CS349!"  
        width = 165.0, height = 100.0  
    }.show()  
}
```



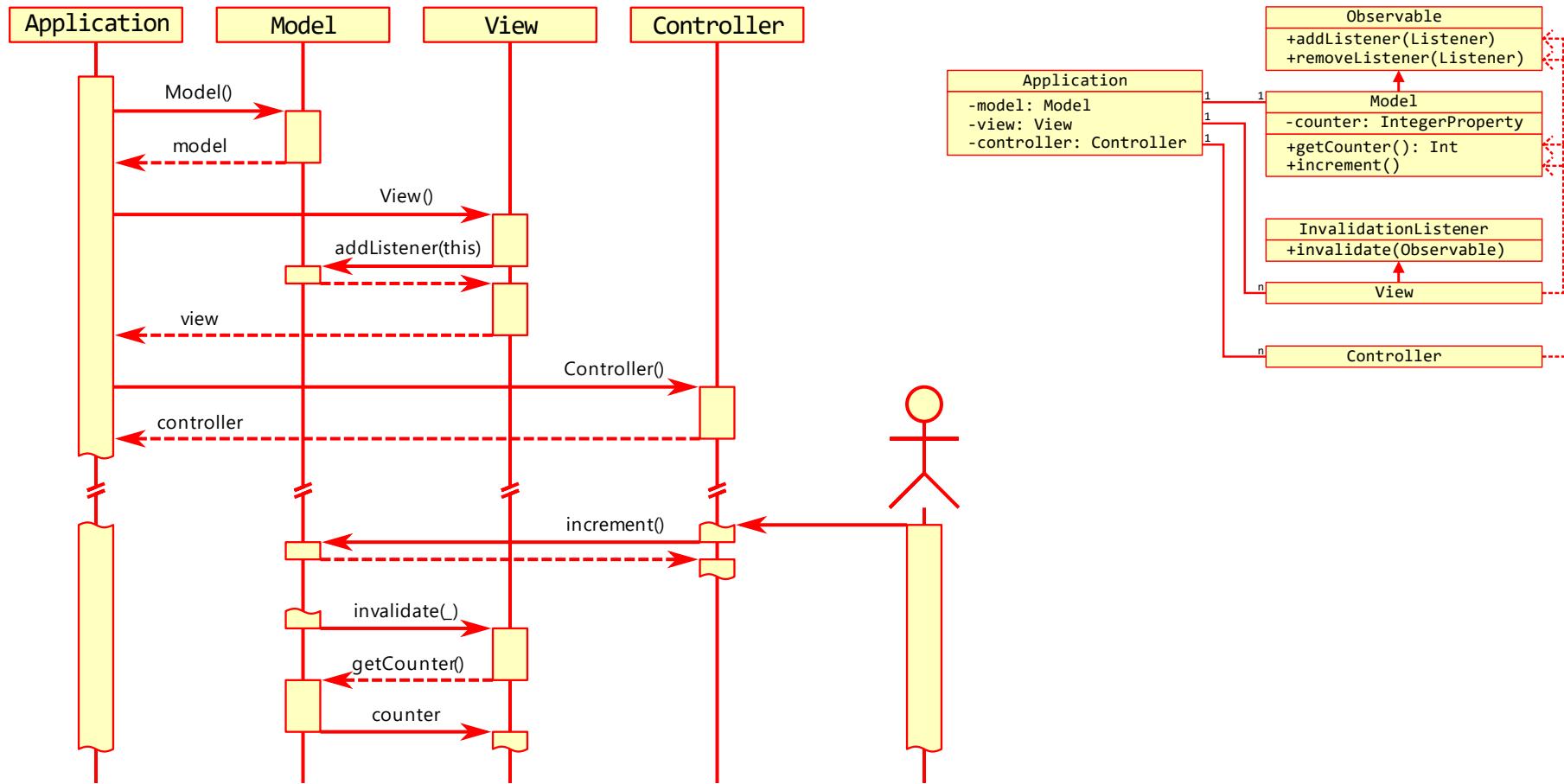
# With MVC – JavaFX

Same approach as before, just using built-in functionality:

- Model is object instead of class: makes Model singleton
- Model is using existing Observable
- Views are using existing InvalidationListener



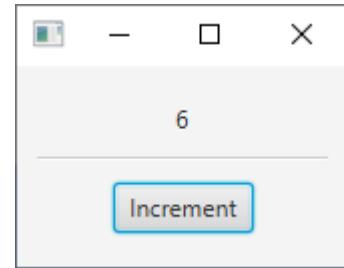
# With MVC – JavaFX



# With MVC – JavaFX

```
object Model: Observable {  
  
    private var counter = 0  
  
    fun increment() {  
        ++counter  
        listeners.forEach { it?.invalidated(this) }  
    }  
    fun getCounter(): Int {  
        return counter  
    }  
  
    private val listeners = mutableListOf<InvalidationListener?>()  
  
    override fun addListener(listener: InvalidationListener?) {  
        listeners.add(listener)  
    }  
    override fun removeListener(listener: InvalidationListener?) {  
        listeners.remove(listener)  
    }  
  
}  
  
class Controller: Button("Increment") {  
    init {  
        onAction = EventHandler {  
            Model.increment()  
        }  
    }  
}
```

```
class View() : Label(), InvalidationListener {  
  
    init {  
        Model.addListener(this)  
        invalidate(null)  
    }  
  
    override fun invalidated(observable: Observable?) {  
        text = Model.getCounter().toString()  
    }  
  
    override fun start(stage: Stage) {  
  
        val view = View()  
        val ctrl = Controller()  
        stage.apply {  
            title = "Hello, CS349!"  
            scene = Scene(Vbox(view, ctrl), 165.0, 100.0)  
        }.show()  
    }  
}
```



# Optimizing View Updates

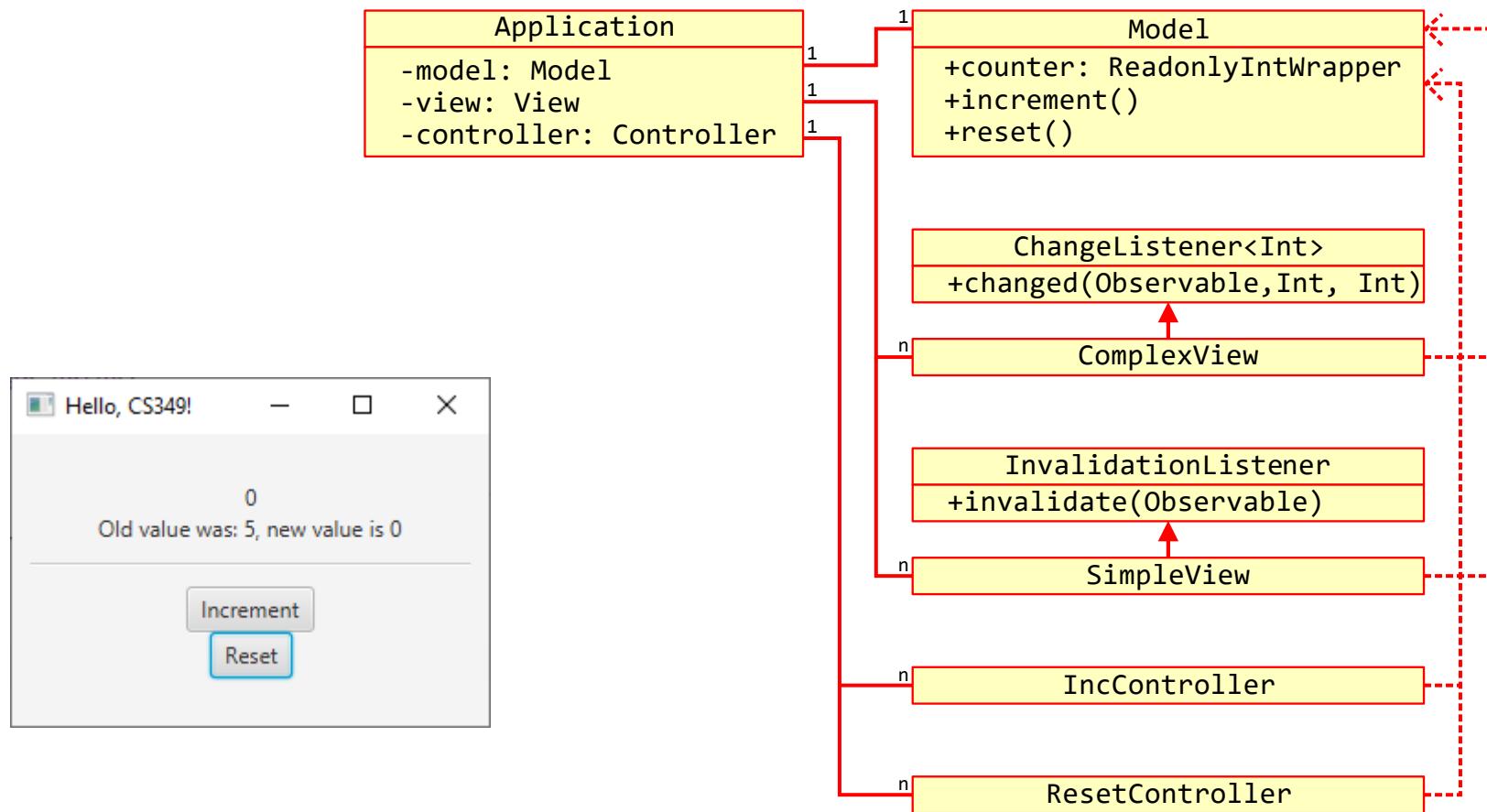
With each notification, *everything* in *every view* is refreshed from the model. To avoid this inefficiency, one could

- Add parameters to view notifications to indicate *what* changed
- Be more granular in what views can listen to

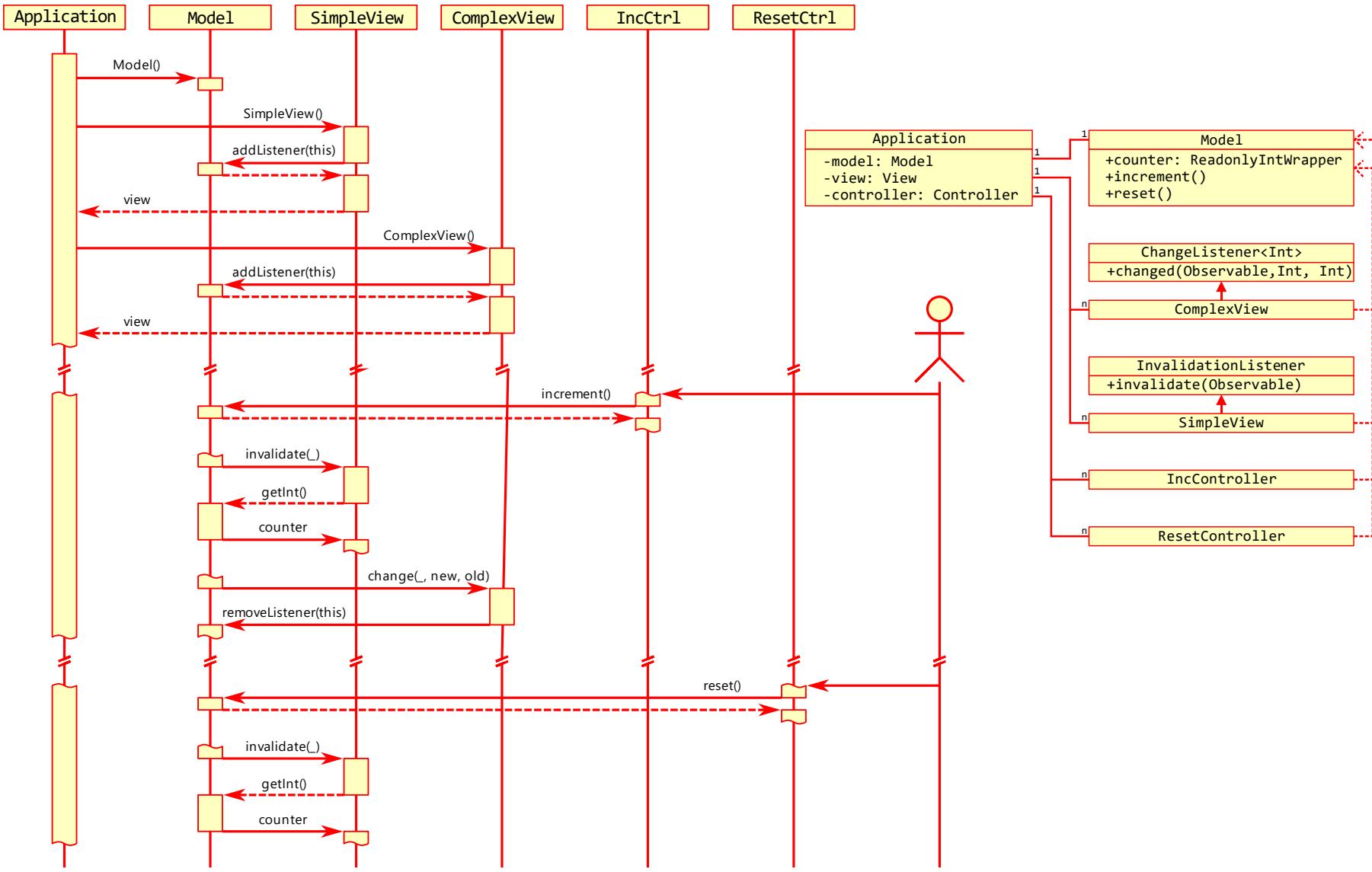
In CS349 we do not worry about efficiency for now: it is okay to just update the entire interface

# With MVC – JavaFX / Read-only Properties

Listening to properties with InvalidationListeners or ChangeListeners.



# With MVC – JavaFX / Read-only Properties



# With MVC – JavaFX / Read-only Properties

```
object Model {  
  
    private val counter = ReadOnlyIntegerWrapper(0)  
    val Counter = counter.readOnlyProperty  
  
    fun incrementCounter() {  
        ++counter.value  
    }  
  
    fun resetCounter() {  
        counter.value = 0  
    }  
}
```

```
class IncController : Button("Increment") {  
    init {  
        onAction = EventHandler {  
            Model.incrementCounter()  
        }  
    }  
}
```

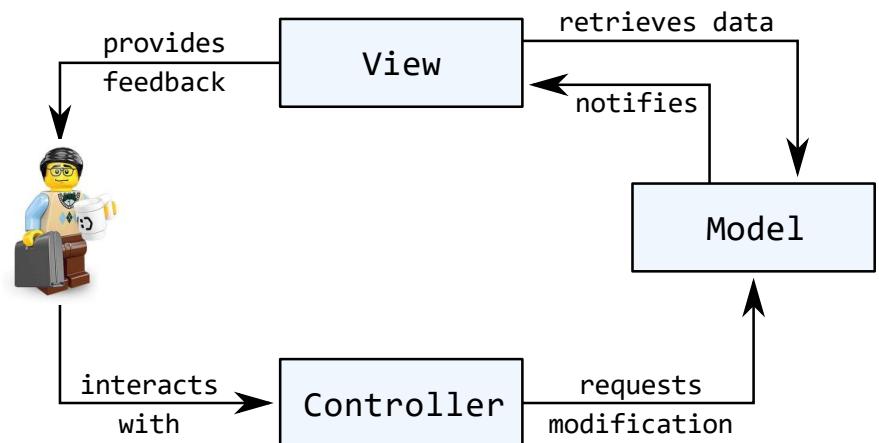
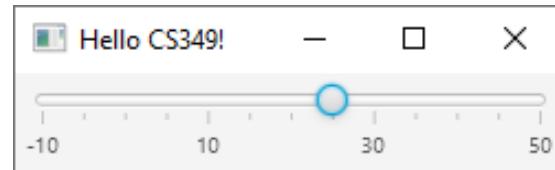
```
class ResetController : Button("Reset") {  
    init {  
        onAction = EventHandler {  
            Model.resetCounter()  
        }  
    }  
}
```

```
class ComplexView : Label(), ChangeListener<Number> {  
  
    init {  
        Model.Counter.addListener(this)  
        changed(null, null, Model.Counter.value)  
    }  
  
    override fun changed(  
        observable: ObservableValue<out Number>?,  
        oldValue: Number?,  
        newValue: Number?) {  
        text = "Old: $oldValue, new: $newValue"  
    }  
}
```

```
class SimpleView : Label(), InvalidationListener {  
  
    init {  
        Model.Counter.addListener(this)  
        invalidated(null)  
    }  
  
    override fun invalidated(observable: Observable?) {  
        text = Model.Counter.value.toString()  
    }  
}
```

# MVC Separation – Theory and Practice

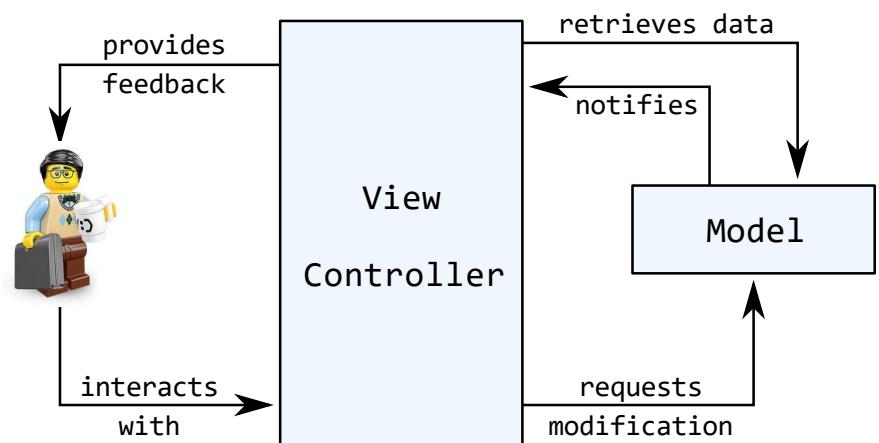
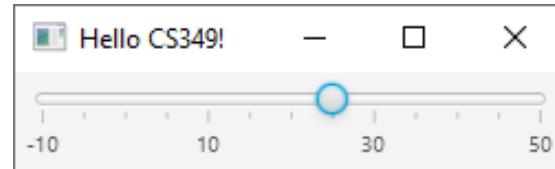
In theory, View and Controller are separate and uncoupled. In practice, View and Controller can be tightly coupled. In this case, separating between view and controller makes little sense.



# MVC Separation – Theory and Practice

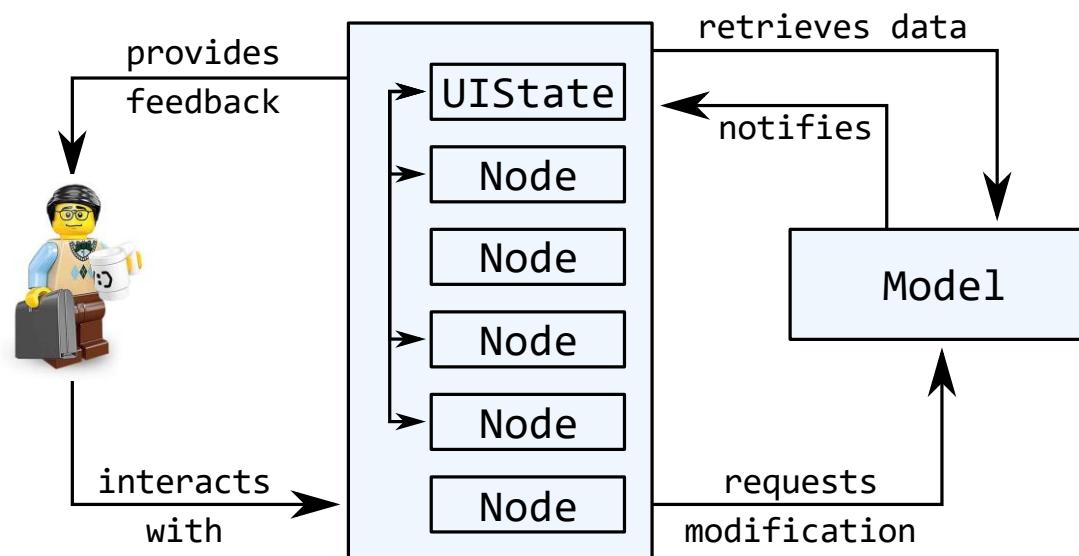
Instead, the view and the controller are oftentimes combined into a “ViewController”.

```
class MySlider(private val model: Model) : Slider(), IView {  
  
    init {  
        min = model.minValue  
        max = model maxValue  
        model.addView(this)  
        valueProperty().addListener { _ ->  
            model.setValue(value)  
        }  
        update()  
    }  
  
    override fun update() {  
        value = model.getValue()  
    }  
}
```



# MVC – Models and States

The Model holds the *system state*, but the View itself can have different *UI states*, e.g., showing a status bar or not, showing English or French labels, etc.



U  
CS 349

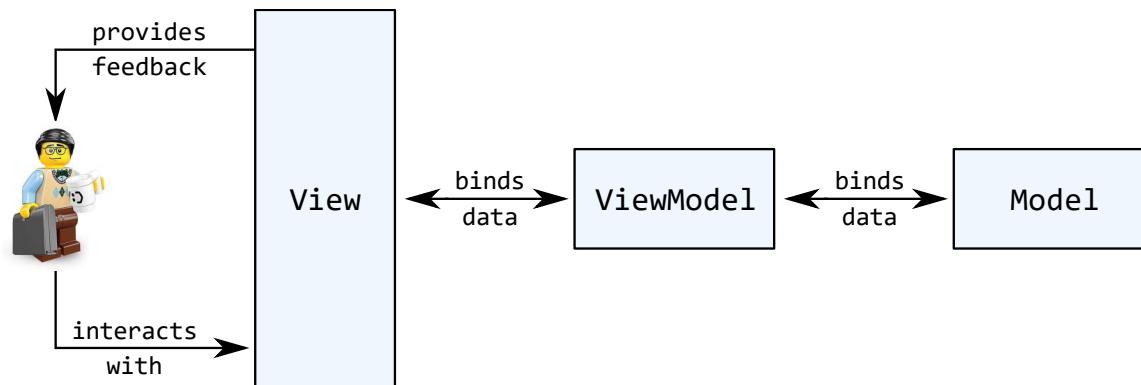
Model–View–ViewModel

# Model–View–ViewModel

The ViewModel mediates between the Model and View.

- Manages the view's display logic
- Display-independent logic is relegated to the Model.

Useful in scenarios where you may have view-dependent state, e.g., a localized UI that uses time and date formats specific to your region; universal data would be in the model, and the data format conversion would be managed by the ViewModel.



# With MVVM

```
object Model {  
  
    private val counter = ReadOnlyIntegerWrapper(0)  
    val Counter = counter.readOnlyProperty  
  
    fun incrementCounter() {  
        ++counter.value  
    }  
  
    fun resetCounter() {  
        counter.value = 0  
    }  
}  
  
class ViewModel {  
  
    val countProperty = SimpleStringProperty()  
  
    init {  
        Model.Counter.addListener { _, _, new ->  
            new as Int  
            countProperty.value =  
                "$ ${new / 100}.${new % 100 / 10}${new % 10}"  
        }  
    }  
    fun incrementCounter() {  
        Model.incrementCounter()  
    }  
    fun resetCounter() {  
        Model.resetCounter()  
    }  
}
```

```
class View(viewModel: ViewModel) : VBox() {  
  
    init {  
        children.addAll(  
            Label().apply {  
                textProperty().bind(viewModel.countProperty)  
            },  
            Button("Increment").apply {  
                onAction = EventHandler {  
                    viewModel.incrementCounter() }  
            },  
            Button("Reset").apply {  
                onAction = EventHandler {  
                    viewModel.resetCounter() }  
            })  
        alignment = Pos.CENTER  
    }  
}
```



# End of the Chapter



Please make sure to

- Understand how the MVC architecture (in theory) works
- Understand the shortcomings of the “pure” MCS architecture
- Understand the different implementation variations of M(VC) in JavaFX.



Any further questions?

-



X

# CS349 – User Interfaces

Events

Basic Event Handling

Event Dispatch

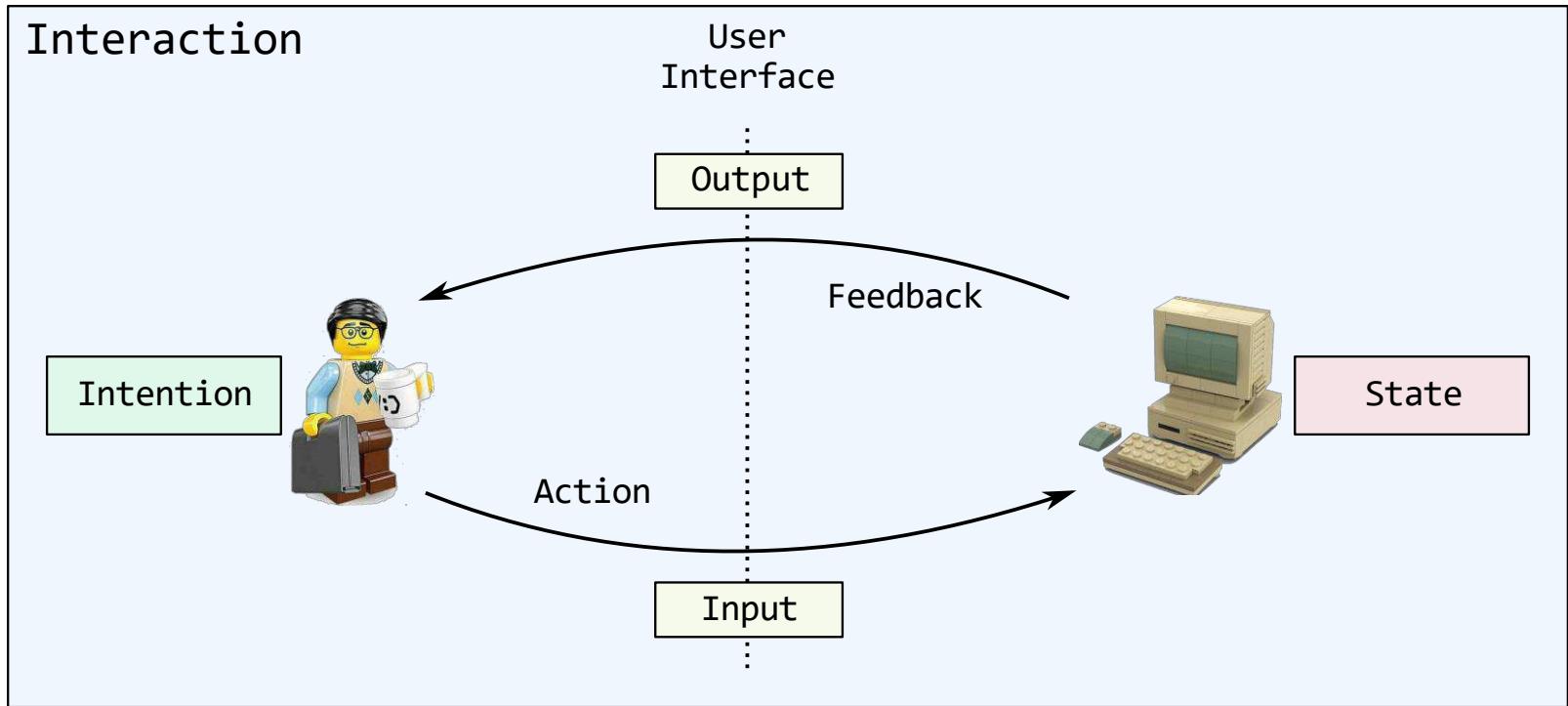
U  
CS 349

February 1

U  
CS 349

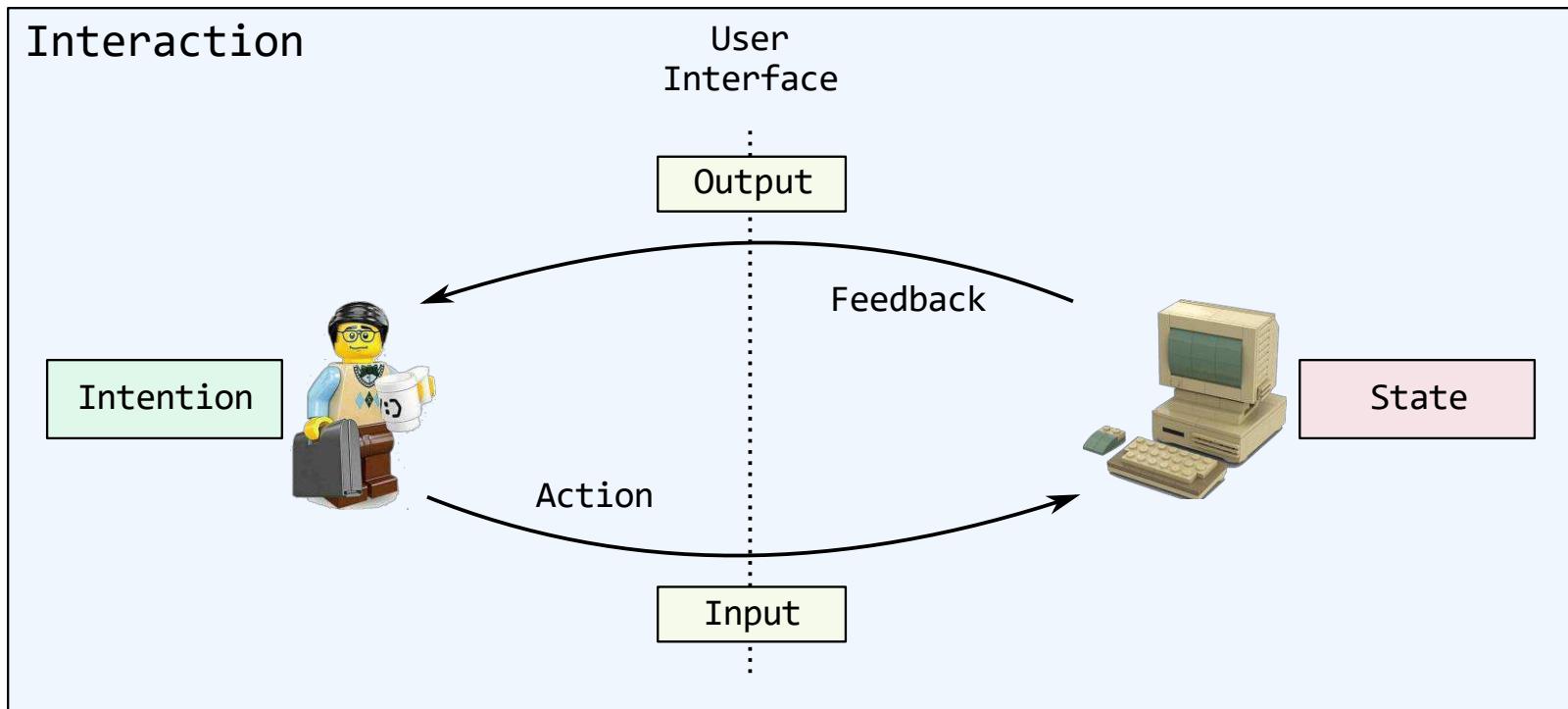
# Events

# Event-Driven Programming



In GUI, users expect immediate feedback from the system to their actions.

# Event-Driven Programming



Event-driven programming is a programming paradigm that bases program execution flow on events. These can represent user actions or other triggers in the system. By using multiple threads and prioritizing certain events, we can create GUIs that retain a responsive feel to users.

# **Why Event-Driven Programming?**

The system is designed to be responsive to events.

This allows us to prioritize user-initiated actions, and carefully prioritize work while remaining responsive to a user.

## **Foreground Events**

- Events initiated by the user
- Created as a result of the user interacting with the user interface

## **Background Events**

- Events generated by the system
- May be received and processed by the user interface.

# Types of Events

An event is a message to notify an application that something happened; a message of interest to an interface.

Events can be initiated by a user, e.g., as mouse input, or by the system, e.g., through a timer.

Examples:

- Properties changes (text, checked, selected)
- Keyboard events (key press, key release)
- Pointer events (button press, button release, mouse move, mouse enter, mouse leave)
- Input focus changes (focus gained, focus lost)
- Window events (window resized, window minimize)
- Timer events (tick)

# Event

Events are messages representing something of interest.

JavaFX has an event class (`javafx.event.Event`), with several specific event types.

All event classes include the following fields:

**EventType** Type of the event that occurred, e.g., `KEY_EVENT`

**source** node from where the event originated

**isConsumed** Whether this event has been processed

All user interface classes also include:

**target** node from where the event started

Events also include fields specific to their subtype; e.g., any mouse event will contain the cursor coordinates.

# Event Subclasses

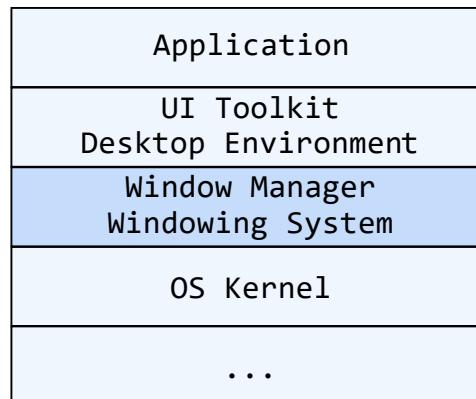
The following Event subclasses are commonly used:

- An **ActionEvent** occurs, for example, when a button is activated.
- A **MouseEvent** occurs when a mouse is used. This includes actions, such as, clicking, pressing, releasing, moving, entering, and exiting.
- A **DragEvent** replaces mouse events during drag-and-drop gesture. This includes actions, such as, drag entered, drag over, drag dropped, and drag exited.
- A **KeyEvent** indicates the key stroke occurred on a node. This includes actions, such as, key pressed, key released, and key typed.
- A **WindowEvent** is related to window showing / hiding actions. This includes actions like window hiding and window showing.

# Event Propagation – System Event Loop

The Window Manager receives notification about an event, packages the event, and pass it to the appropriate UI toolkit. It performs these steps:

1. Collecting event information from the underlying OS Kernel
2. Storing relevant information in an event structure, and storing events in a queue
3. Dispatching events to the correct UI toolkit / Application.<sup>†</sup>



<sup>†</sup> This is typically the application window that triggered the event but can be another window that is intercepting events.

# Event Propagation – JVM Event Loop

Kotlin applications rely on the JVM to queue, manage, and dispatch events for each running application.

The JVM has an event-handling thread that performs these steps:

1. Pulling events from the JVM event queue
2. Formatting them as Kotlin events
3. Dispatching them to the Application

## **Event Propagation – Application Event Loop**

An application might have its own event loop and secondary event queue that accumulate events until it is ready to handle them.

U  
CS 349

# Basic Event Handling

# Basic Event Handlers

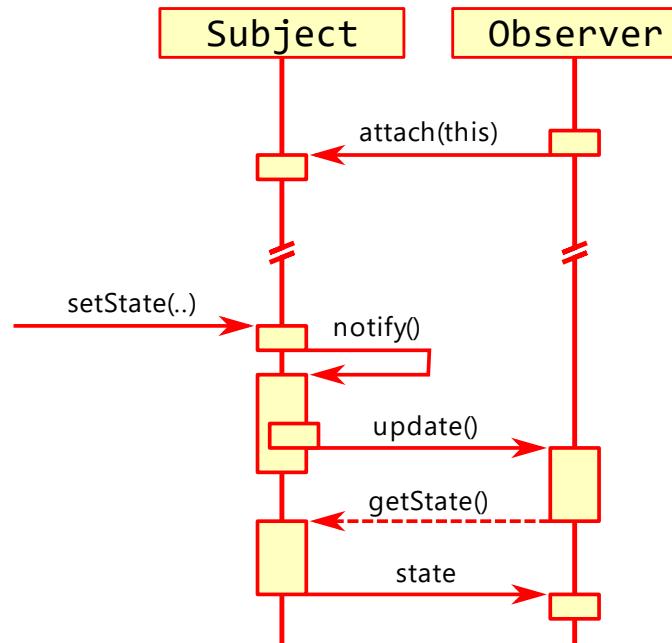
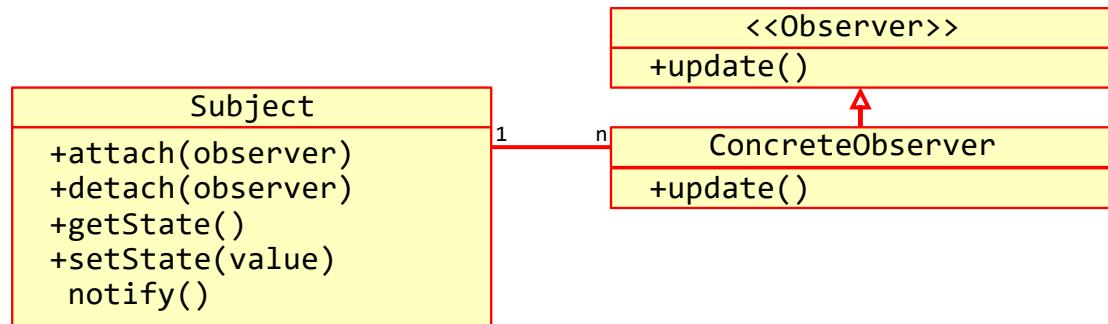
JavaFX defines interfaces for *specific event types* (or *device types*), e.g., MouseEvent, KeyEvent, TouchEvent, etc.

1. Attach a listener function to these events that process them.
2. When an event is dispatched, the relevant listener function is called for that widget.

This approach follows the Observer-pattern:

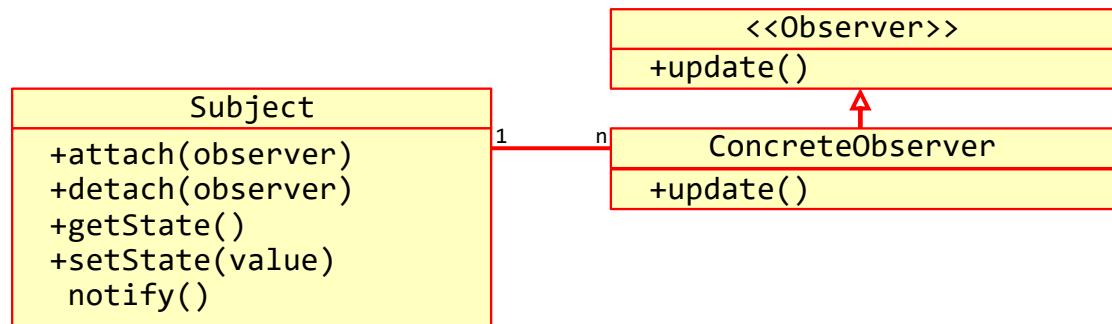
# Observer Pattern

With the observer pattern, a subject, maintains a list of observers, and notifies them of any state changes, usually by calling one of their methods.



# Basic Event Handlers

In JavaFX, observers are implemented as EventHandlers. It is a generic class that captures and processes a specific event. They typically are associated with the widget that generates the event.



```
scene.onMouseClicked = // subject / attach
    EventHandler { // observer
        stage.title =
            "Click ${it.sceneX}/${it.sceneY}" } // getState
```

U  
CS 349

# Event Dispatch

# Event Dispatch in Java FX

The event dispatch process contains the following steps:

1. **Target selection** Which node should receive the event?
2. **Route construction** What is the path to the node through the scene graph?
3. **Event capturing** Traverse path downwards from Root to the node
4. **Event bubbling** Traverse path upwards from the node back to Root

# Target Selection & Route Construction

**Target Selection** is determined by the type of Event:

- For **key events**, the target is the node that has focus
- For **mouse events**, the target is the node at the location of the cursor
- For **touch screen events**, target selection may be more complex, e.g., a **continuous gesture** (like pinch-to-zoom) might select the target node at the center point of all touches at gesture start, whereas a **swipe** (like swipe right) might select the target node at the center of the entire path of all fingers

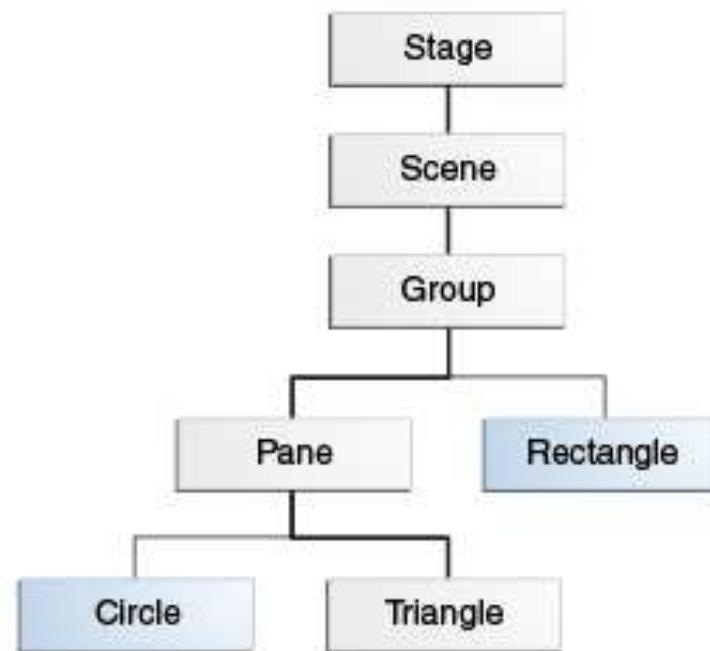
**Route construction** is the path to a particular Node through the tree:

- Stored as chain, using the Source and Target fields of the event.

# Event Capturing and Bubbling

JavaFX supports both top-down and bottom-up processing. Events propagate from the Root to the Target (“Capture phase”), then back up to the Root (“Bubble phase”). Any Node in the path can intercept (“consume”) the Event, on either pass.

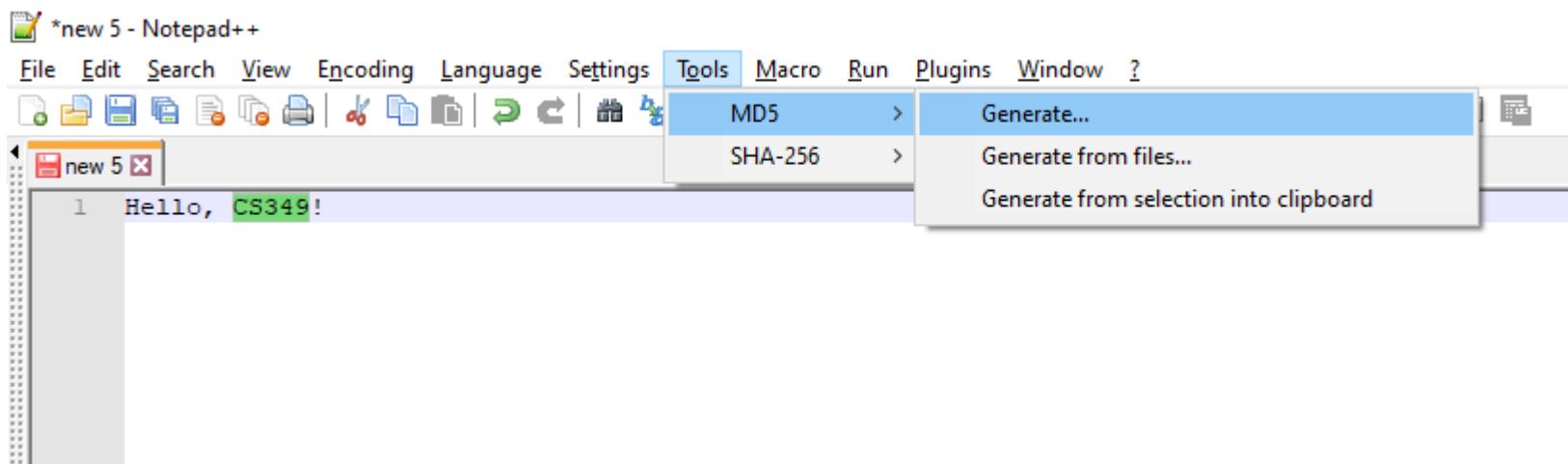
The **Capture phase** walks **down** the tree from the Stage (root) through each Node until it reaches the Triangle.



The **Bubble phase** walks **up** the tree from the Triangle, through each Node until it reaches the Stage (Root)

# Positional Dispatch

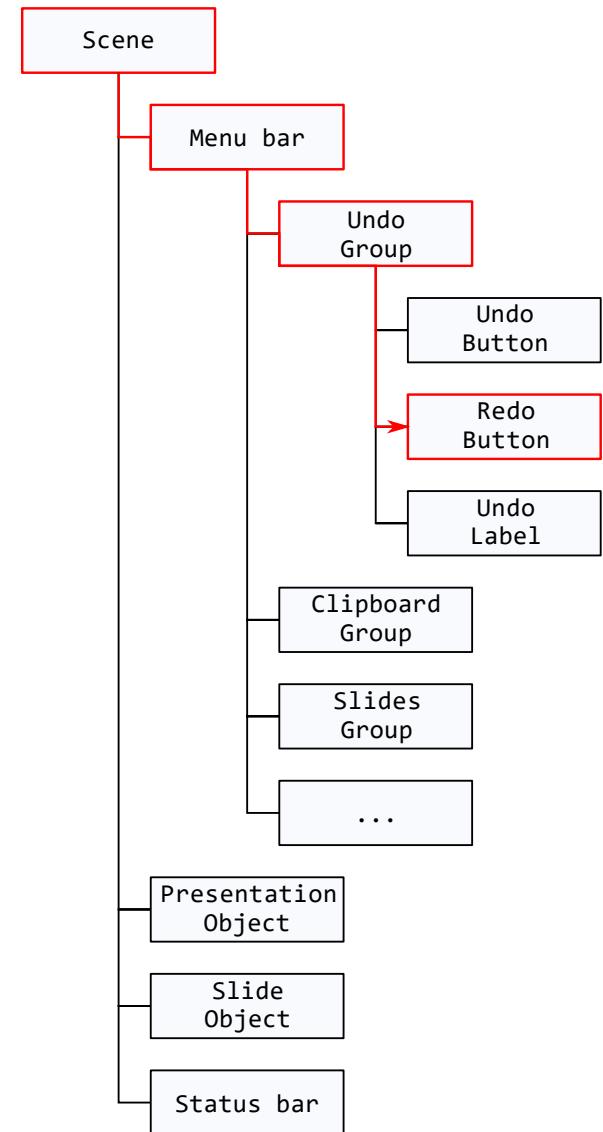
Sending events to the Node under the cursor is called **positional dispatch**.



# Top-down Positional Dispatch (using Capture phase)

Event is dispatched to the root node of the scene graph first, and then travels through the scene graph to the target node. This means that the target node receives the event first.

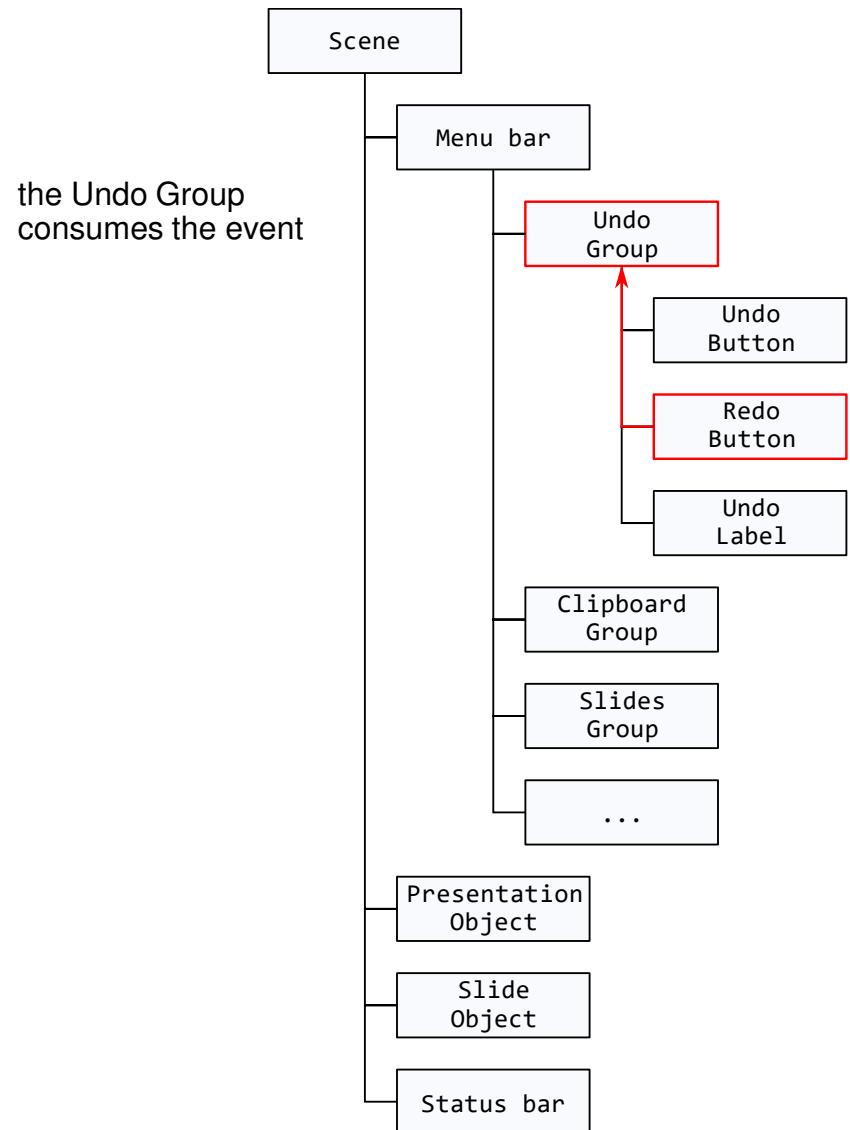
Any intermediate node can decide to consume the event, in which case it will not further propagate.



# Bottom-up Positional Dispatch (using Bubble phase)

Event is dispatched to the target node first, and than travels through the scene graph towards the root node.

Event is dispatched to leaf node widget in the UI tree that contains the mouse cursor (using a Handler, registered at the Node for that event)

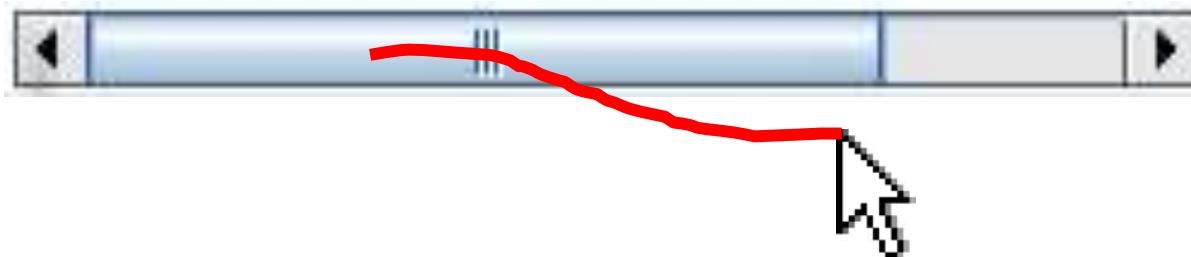


# Positional Dispatch Limitations

Positional dispatch can lead to odd behaviour:

- Mouse drag starts in a scrollbar, but then moves outside the scrollbar: send the events to the adjacent widget?
- Mouse press event in one button widget but release is in another: each button gets one of the events?

Sometimes position is not enough, also need to consider which widget is “in focus”

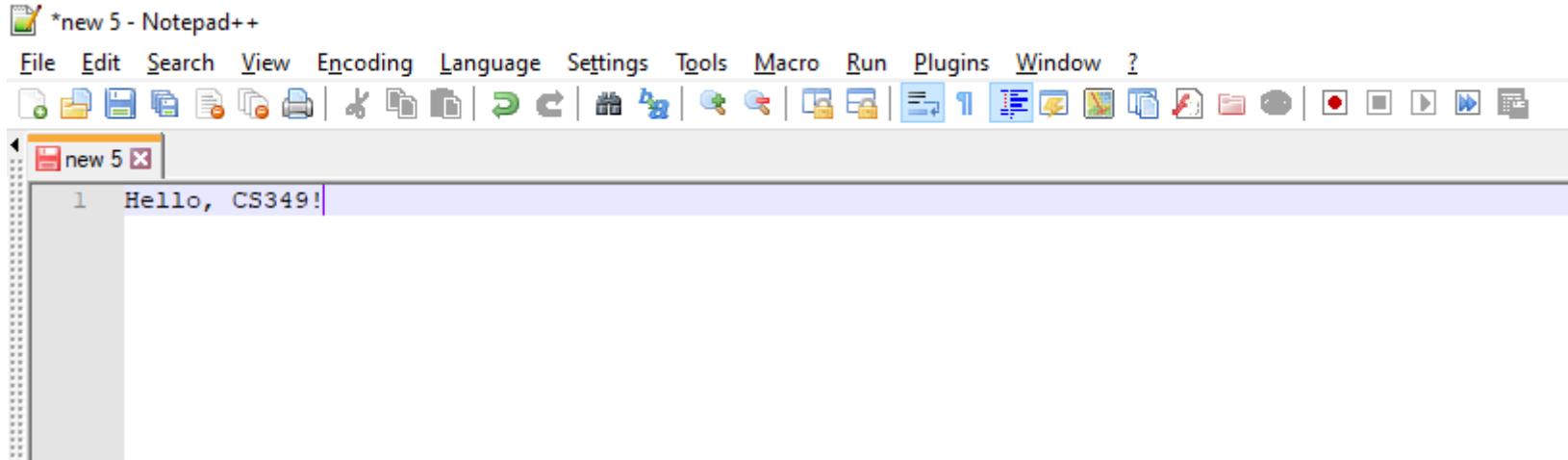


# Focus Dispatch

Events dispatched to widget regardless of mouse cursor position

Needed for all keyboard and some mouse events:

- Keyboard focus: Click on text field, move cursor off, start typing
- Mouse focus: Mouse down on button, move off, mouse up ... also called “mouse capture”

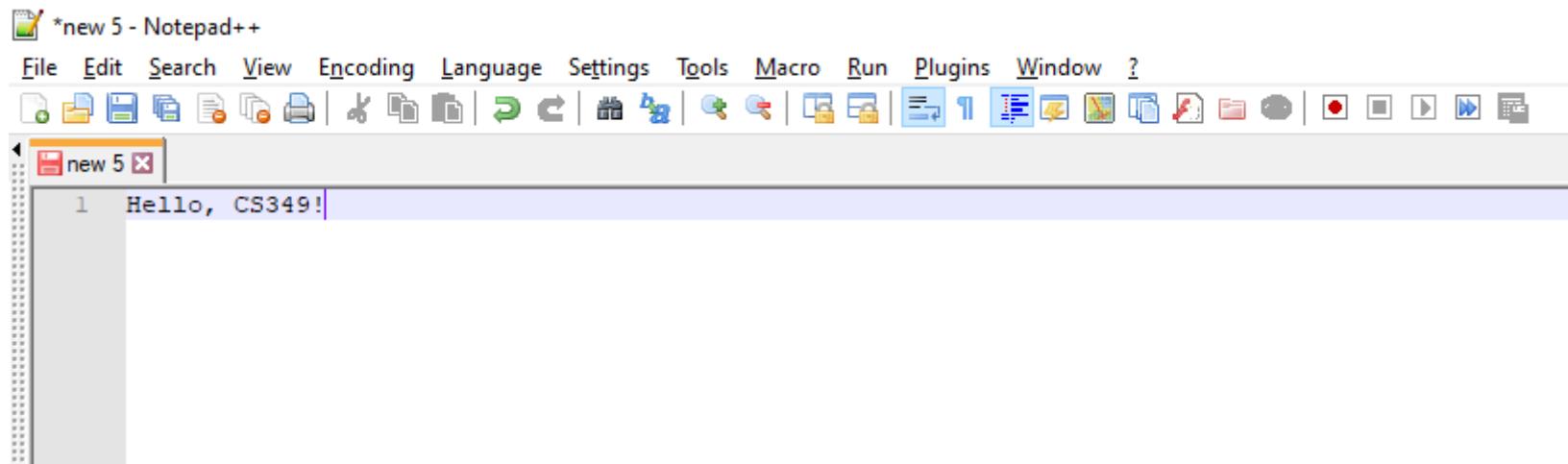


# Focus Dispatch

Maximum one keyboard focus and one mouse focus

Need to gain and lose focus at appropriate times

- Transfer focus on mouse down (“capture”)
- Transfer focus when TAB key is pressed



# Setting up Positional Dispatch Handlers

1. Define an event handler to capture a specific type of device, e.g., Mouse, Keyboard, or Touch.
2. Register the handler with a node
  - as a **filter** to process an event during the **capture** phase
  - as a **handler** to process during the **bubble** phase.

All “interested” nodes that could potentially process an event (i.e., that are in the route for dispatch) can register for an event and will have an opportunity to process it. They can also choose to “consume” an event, in which case it stops being propagated further along the route.

# Setting up Positional Dispatch Handlers

Add to a specific Node either using EventFilter (Capture phase) or EventHandler (Bubble phase):

```
scene.addEventFilter(MouseEvent.MOUSE_CLICKED) {  
    println("Click:Filter ${it.sceneX}/${it.sceneY}") }  
  
scene.addEventHandler(MouseEvent.MOUSE_CLICKED) {  
    println("Click:Handler ${it.sceneX}/${it.sceneY}") }  
  
// Output:  
// Click:Filter 349.0/32.0  
// Click:Handler 349.0/32.0
```

# Setting up Positional Dispatch Handlers

```
override fun start(stage: Stage) {
    val filter = {it: MouseEvent ->
        println("Filter source: ${it.source.javaClass}")
        println("Filter target: ${it.target.javaClass}")
    }
    val handler = {it: MouseEvent ->
        println("Handler source: ${it.source.javaClass}")
        println("Handler target: ${it.target.javaClass}")
    }
    val rect = Rectangle(120.0, 120.0, Color.RED).apply {
        addEventFilter(MouseEvent.MOUSE_CLICKED, filter)
        addEventHandler(MouseEvent.MOUSE_CLICKED, handler)
    }
    val root = Pane().apply {
        addEventFilter(MouseEvent.MOUSE_CLICKED, filter)
        addEventHandler(MouseEvent.MOUSE_CLICKED, handler)
        translateX = 60.0
        background = Background(BackgroundFill(Color.GREEN,
            children.add(rect)
        )
    }
    stage.apply {
        scene = Scene(root, 320.0, 240.0).apply {
            addEventFilter(MouseEvent.MOUSE_CLICKED, filter)
            addEventHandler(MouseEvent.MOUSE_CLICKED, handler)
        }
        title = "Hello CS349!"
    }.show()
}
```



Filter source: Scene  
Filter target: Rectangle  
Filter source: Pane  
Filter target: Rectangle  
Filter source: Rectangle  
Filter target: Rectangle  
Handler source: Rectangle  
Handler target: Rectangle  
Handler source: Pane  
Handler target: Rectangle  
Handler source: Scene  
Handler target: Rectangle

# End of Chapter



Any further questions?

-



X

# Using GUI interfaces

Indirect vs. Direct Manipulation

Instrumental Interaction

U

CS 349

February 6

# Reminder – GUI Interaction

For desktop-based GUI interaction, we can assume the presence of a

- Screen capable of “high-resolution” graphics output
- Text entry (e.g., keyboard), and
- Pointing device (e.g., mouse, touchpad)

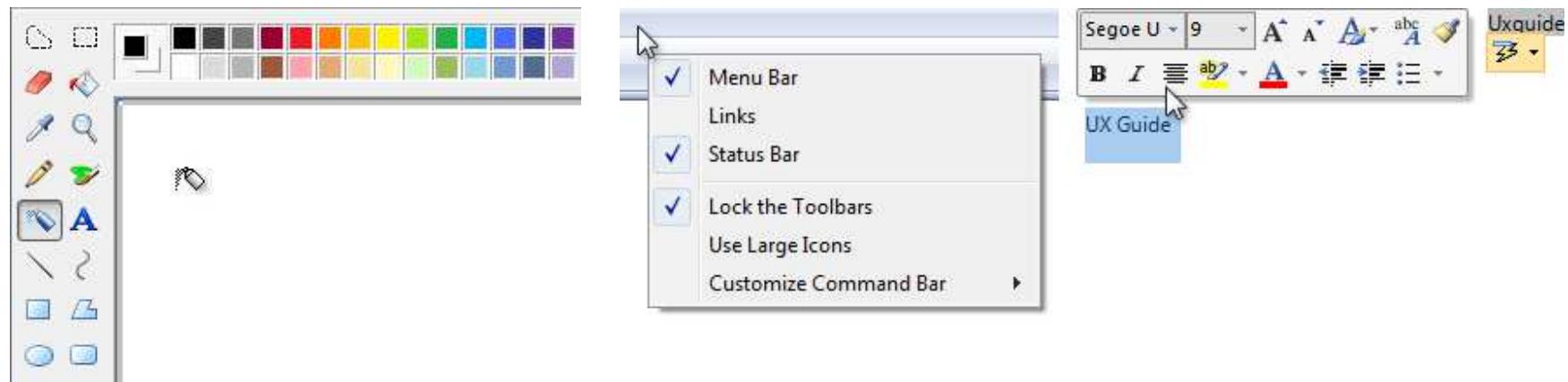


# Reminder – WIMP

Interfaces consists of Windows, Icons, Menus, and Pointers  
It is usually associated with the “desktop” metaphor.

Users interact by pointing + clicking

- Point at the object of interest (e.g., widget, image, text)
- Click to select, Drag to move, Double-click to activate
- “Real-time” interaction and feedback, using on-screen animation



# GUI Interaction – Advantages

The user always remains in control

- The system waits for input, then responds
- Unlike batch where the user waits for the program to complete

The design emphasizes **recognition** of interface features over **recall** of complex commands.

- Utilizes familiar graphical elements across different applications.
- Enables discovery of options and experimentation

## Mania for icons may be mixed blessing

A picture may be worth a thousand words, but is it worth two or three alphanumeric keystrokes?

That is a question I ponder as everybody in micro-computing goes icon-crazy.

What began as the Macintosh revolution, using a "mouse" to pinpoint and call up editorial options which were represented by icons

**Personal  
computers**

**CHRY'S  
GOYENS**



With or without icons, entry to a program is fairly simple in either case. In the latter, the user would have to memorize a small series of DOS commands, and then learn another series of program commands, for the WordStar (which, by the way, are on-screen in directory mode).

Where the desktop environment

ceding the letters or after, or even a backslash (/) preceding the letters, and that any typo will mean no access, you know that Windows will help.

Windows simplifies hard-disk management. Spreadsheet information from a program like Lotus 1-2-3 can be taken from there and transferred to Windows Write, the

# GUI Interaction – Advantages

It uses metaphor to make the interface more familiar

- Graphical objects results in an interaction language that is closer to users' own language, and closer to the task domain
- Examples include: "desktop", "folder", and "drag-and-drop"

## Mania for icons may be mixed blessing

A picture may be worth a thousand words, but is it worth two or three alphanumeric keystrokes?

That is a question I ponder as everybody in micro-computing goes icon-crazy.

What began as the Macintosh revolution, using a "mouse" to pinpoint and call up editorial options which were represented by icons

**Personal  
computers**

**CHRY'S  
GOYENS**



With or without icons, entry to a program is fairly simple in either case. In the latter, the user would have to memorize a small series of DOS commands, and then learn another series of program commands, for the WordStar (which, by the way, are on-screen in directory mode).

Where the desktop environment

ceding the letters or after, or even a backslash (/) preceding the letters, and that any typo will mean no access, you know that Windows will help.

Windows simplifies hard-disk management. Spreadsheet information from a program like Lotus 1-2-3 can be taken from there and transferred to Windows Write. the

**Consistent interface elements** also help to make a GUI explorable and predictable for users.

## Titlebar

02.gui\_interaction

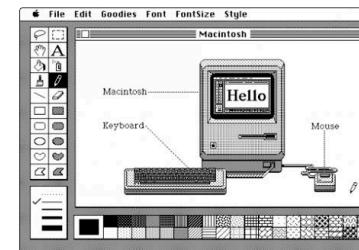
Section

Tooltip

## Graphical User Interface (GUI)

### Hardware

- High resolution, high refresh graphics
- Keyboard e.g. mechanical, touchscreen
- Pointing device e.g. mouse, touchpad



### Capabilities

- Display graphics, animation and text
- Manage text entry
- Point-and-click interaction

### The system needs to

- Handle input devices - keyboard (text) and pointing (mouse/touch)
- Provide output methods - drawing primitives, bitmaps, text
- Wait for user input and respond to it in a timely manner.

Slide (object)

Click to add notes

Slide 2 of 31 English (United States)

Notes

Comments



89%

-



X

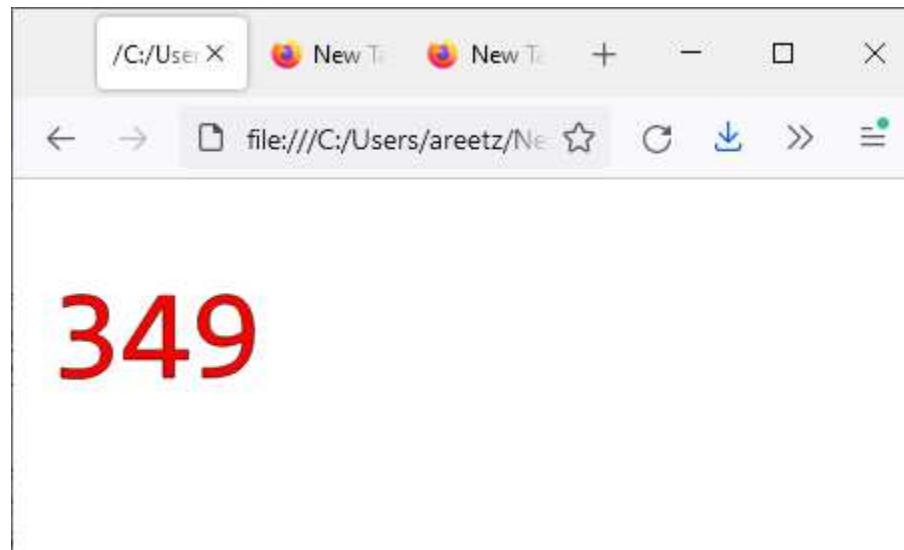
U

CS 349

# Indirect vs. Direct Manipulation

# Indirect vs. Direct Manipulation

Our goal is modifying this SVG-file (Scalable Vector Graphic), here shown in Firefox. We want to move the text “**349**” to the right:



# Indirect Manipulation

One way to achieve this goal is by modifying the SVG-file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Created with Inkscape (http://www.inkscape.org/) -->

<svg xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:cc="http://creativecommons.org/ns#" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
<defs id="defs10903" />
<sodipodi:namedview id="base" pagecolor="#ffffff" bordercolor="#666666" borderopacity="1.0" inkscape:pageopacity="0.0" inkscape:pag-
<metadata id="metadata10906">
<g inkscape:label="Layer 1" inkscape:groupmode="layer" id="layer1" transform="translate(0,-257)">
<text
  xml:space="preserve"
  style="font-style:normal;font-variant:normal;font-weight:normal;font-stretch:normal;font-size:16.93333244px;line-height:1.25;f
  x="18.966774"
  y="283.31937"
  id="text10886"
  inkscape:transform-center-x="-1.4365737"
  inkscape:transform-center-y="1.3363476">
<tspan
  sodipodi:role="line"
  id="tspan10884"
  x="18.962805"
  y="283.31937"
  style="fill:#ff0000;fill-opacity:1;stroke:#000000;stroke-width:0.15000001;stroke-miterlimit:4;stroke-dasharray:none;stroke-o
</tspan>
</text>
</g>
</svg>
```

# Indirect Manipulation

Is this good or bad (in terms of interaction design)?

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Created with Inkscape (http://www.inkscape.org/) -->

<svg xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:cc="http://creativecommons.org/ns#" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
<defs id="defs10903" />
<sodipodi:namedview id="base" pagecolor="#ffffff" bordercolor="#666666" borderopacity="1.0" inkscape:pageopacity="0.0" inkscape:pag-
<metadata id="metadata10906">
<g inkscape:label="Layer 1" inkscape:groupmode="layer" id="layer1" transform="translate(0,-257)">
<text
  xml:space="preserve"
  style="font-style:normal;font-variant:normal;font-weight:normal;font-stretch:normal;font-size:16.93333244px;line-height:1.25;f
  x="18.966774"
  y="283.31937"
  id="text10886"
  inkscape:transform-center-x="-1.4365737"
  inkscape:transform-center-y="1.3363476">
<tspan
  sodipodi:role="line"
  id="tspan10884"
  x="18.962805"
  y="283.31937"
  style="fill:#ff0000;fill-opacity:1;stroke:#000000;stroke-width:0.15000001;stroke-miterlimit:4;stroke-dasharray:none;stroke-o
</tspan>
</text>
</g>
</svg>
```

# Indirect Manipulation

Advantages:

Disadvantages:

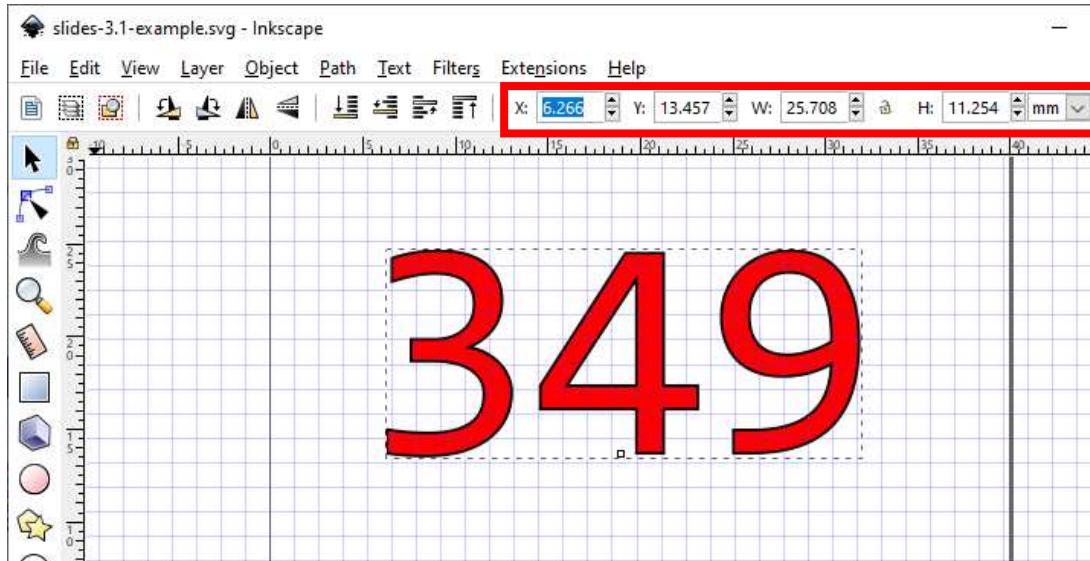
```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Created with Inkscape (http://www.inkscape.org/) -->

<svg xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:cc="http://creativecommons.org/ns#" xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
<defs id="defs10903" />
<sodipodi:namedview id="base" pagecolor="#ffffff" bordercolor="#666666" borderopacity="1.0" inkscape:pageopacity="0.0" inkscape:pag-
<metadata id="metadata10906">
<g inkscape:label="Layer 1" inkscape:groupmode="layer" id="layer1" transform="translate(0,-257)">
<text
  xml:space="preserve"
  style="font-style:normal;font-variant:normal;font-weight:normal;font-stretch:normal;font-size:16.93333244px;line-height:1.25;f
  x="18.966774"
  y="283.31937"
  id="text10886"
  inkscape:transform-center-x="-1.4365737"
  inkscape:transform-center-y="1.3363476">
<tspan
  sodipodi:role="line"
  id="tspan10884"
  x="18.962805"
  y="283.31937"
  style="fill:#ff0000;fill-opacity:1;stroke:#000000;stroke-width:0.15000001;stroke-miterlimit:4;stroke-dasharray:none;stroke-o
</tspan>
</text>
</g>
</svg>
```

- Can be perfectly precise
- Can be automated (e.g., when batch-processing multiple files)

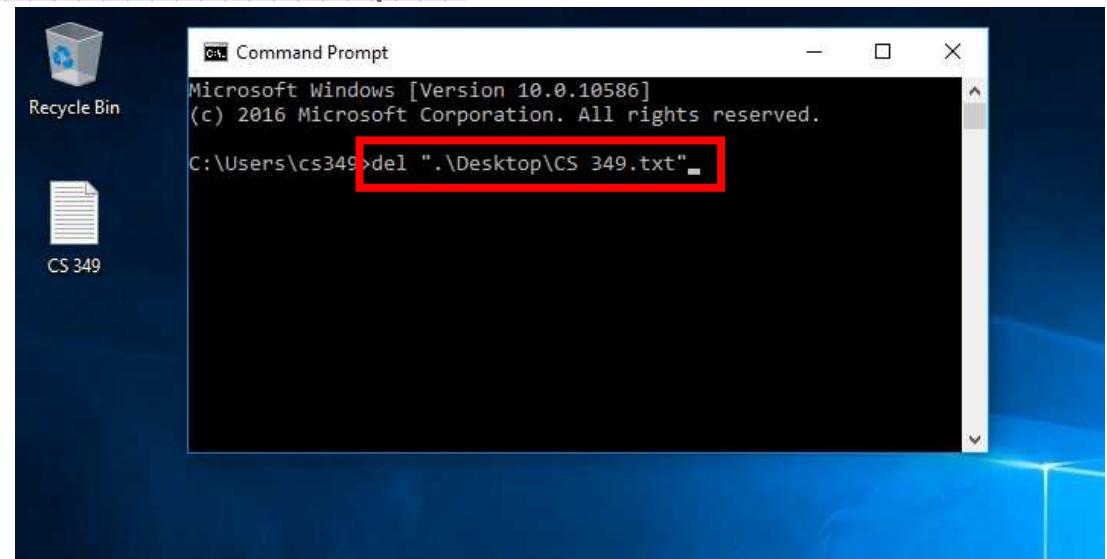
- Unclear which value to modify
- Unclear what the values mean
- Can be time-consuming on complex files
- No visual feedback of the result of the modification

# Indirect Manipulation – Other Examples

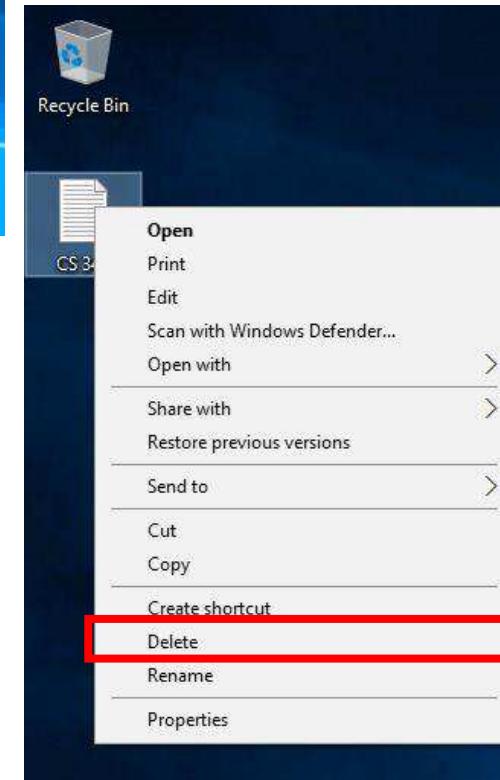
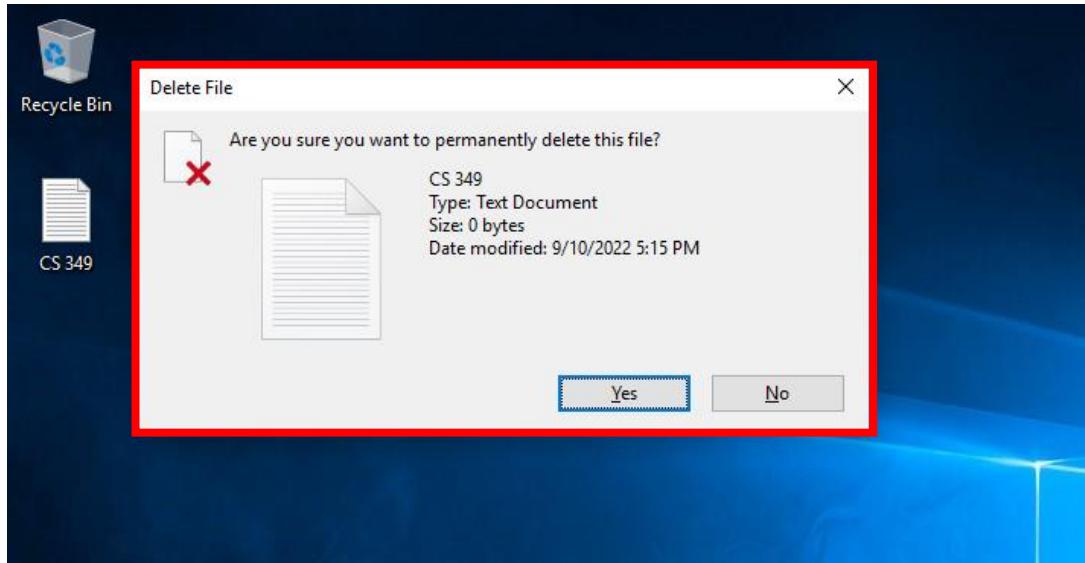


## Other disadvantages

- Oftentimes no “Undo”

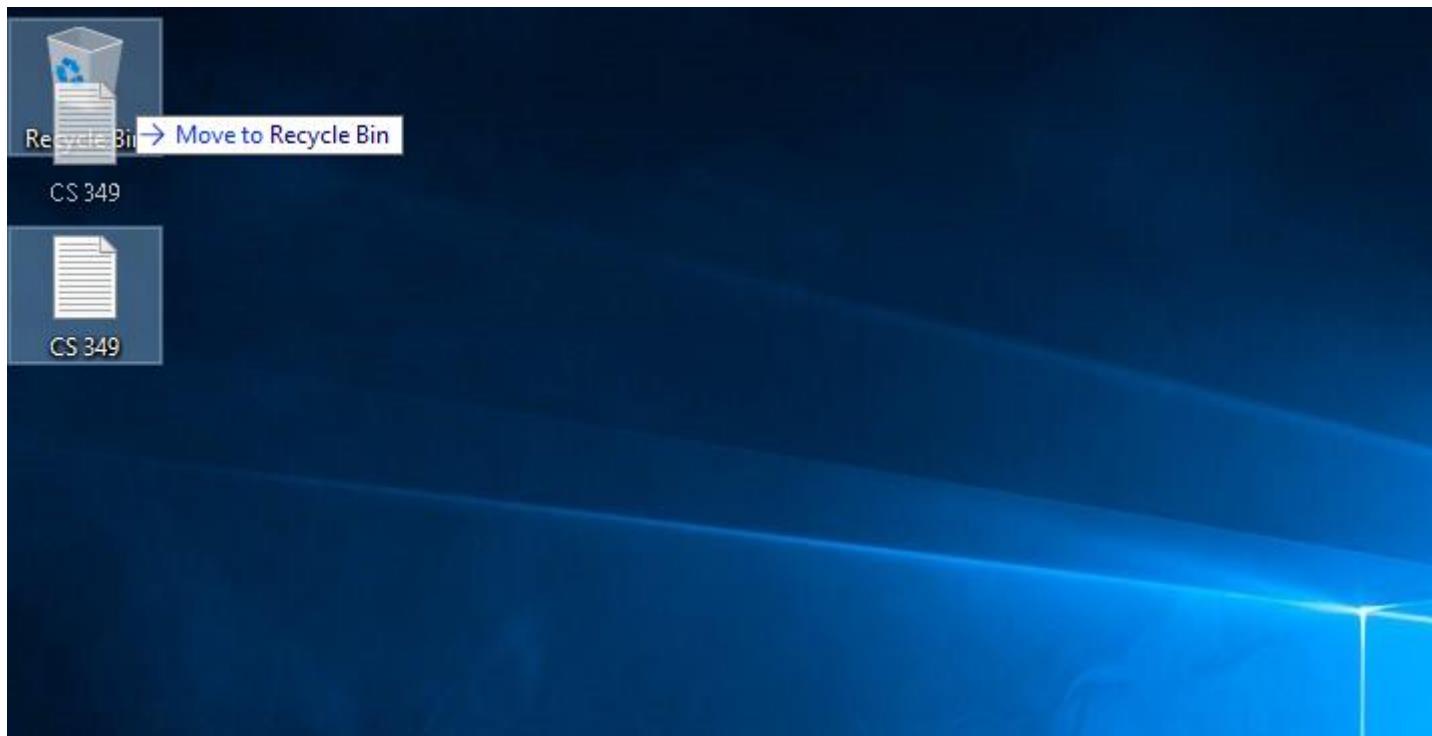


# Indirect Manipulation – Other Examples



# Direct Manipulation

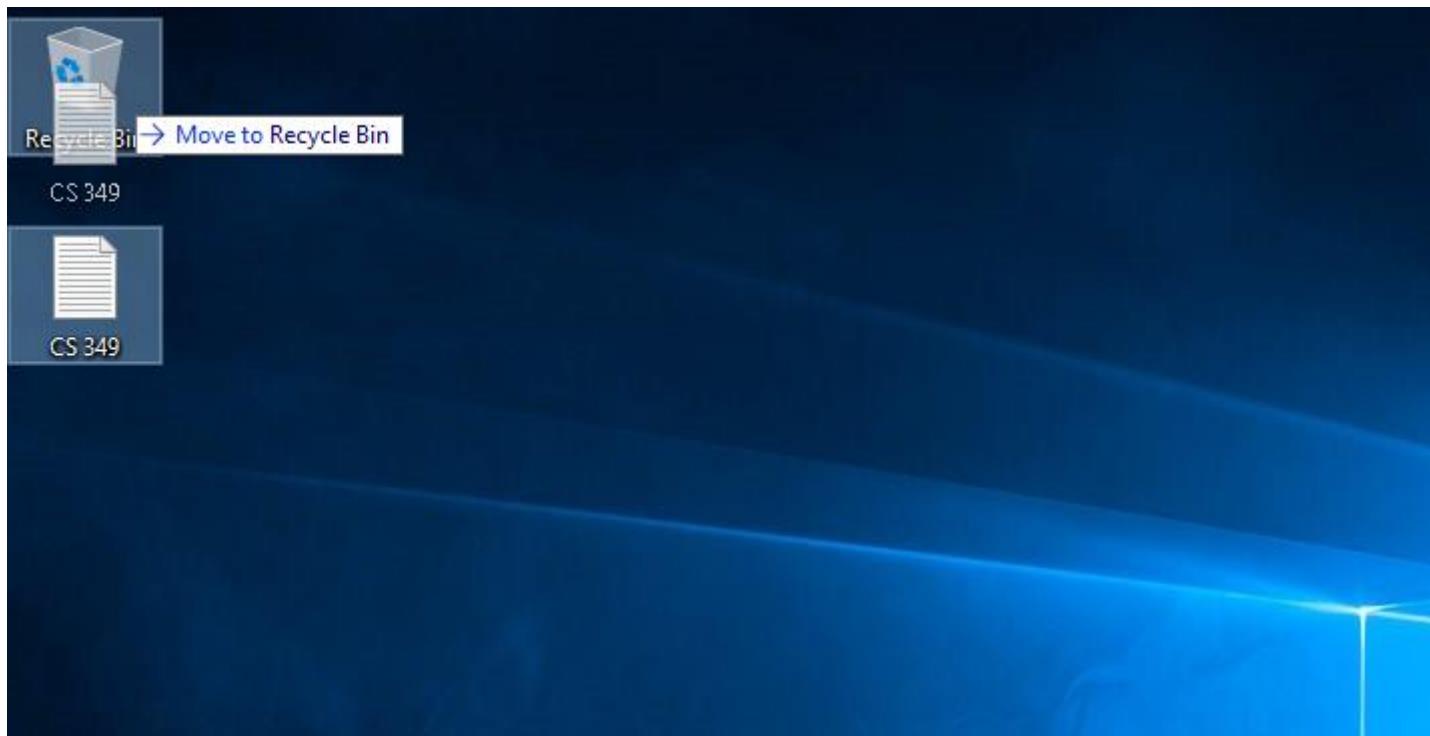
- Continuous representation of the object of interest
- Physical actions instead of complex syntax
- Continuous feedback and reversible, incremental actions
- Rapid, self-revealing approach to learning



# Direct Manipulation

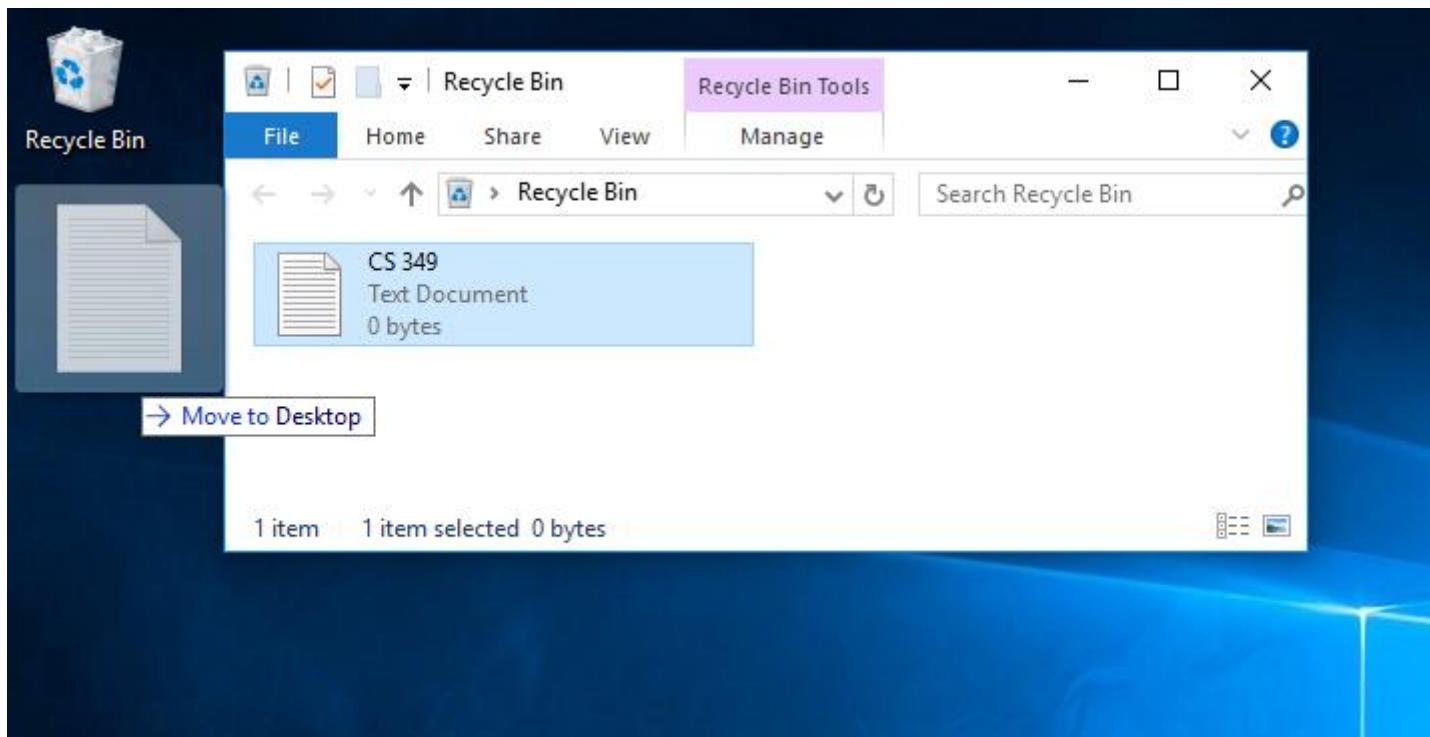
Direct manipulation is when a virtual representation of data (object of interest) is manipulated in a similar way to a real-world object.

Direct manipulation is meant to make the interaction feel as if the user was manipulating a real-world object instead of working through an intermediary.



# Direct Manipulation

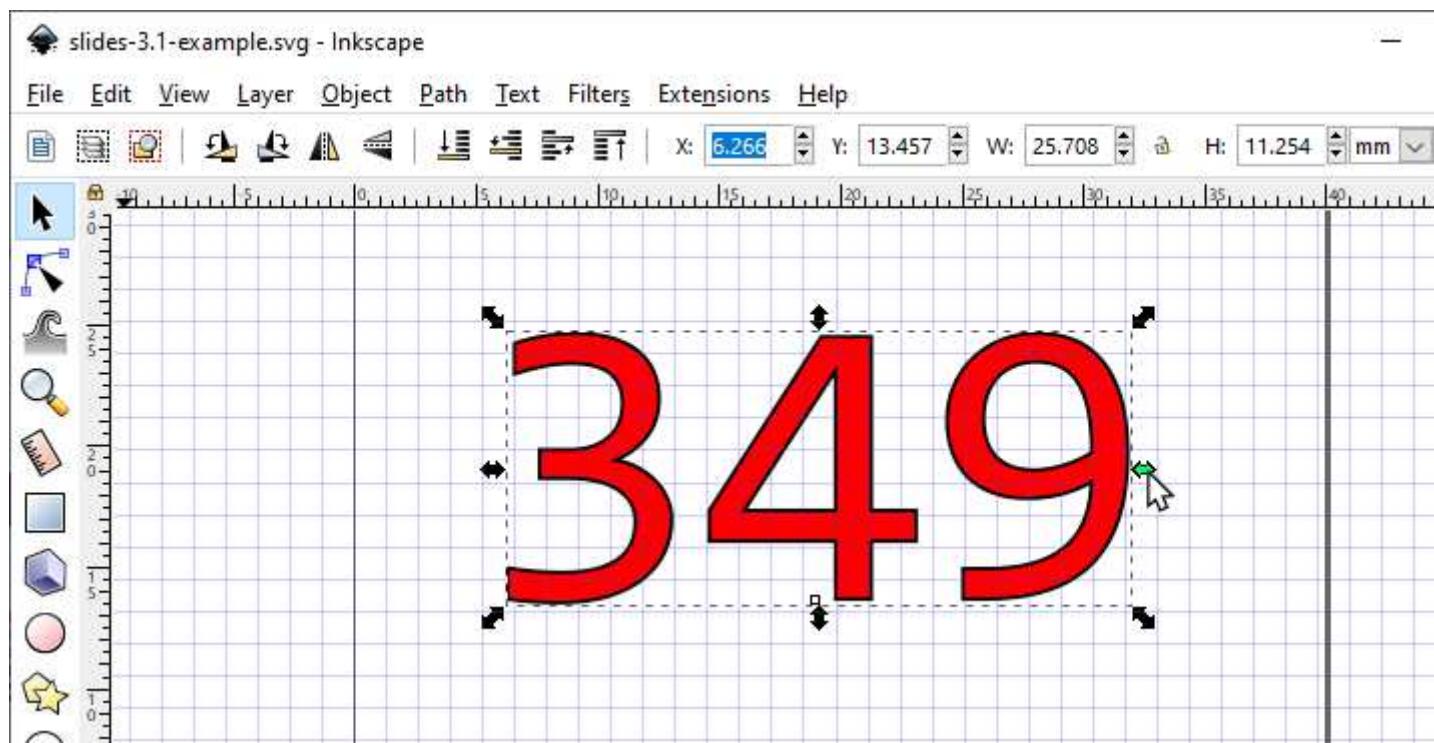
- Continuous representation of the object of interest
- Physical actions instead of complex syntax
- Continuous feedback and reversible, incremental actions
- Rapid, self-revealing approach to learning



# Direct Manipulation – Other Examples

# How do we use this in designing a user interface?

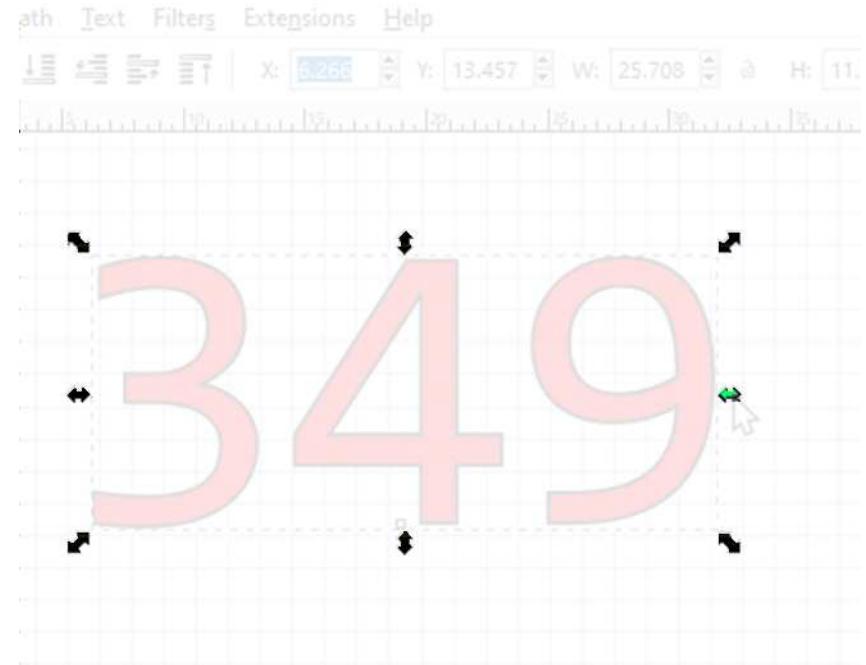
- We divide the interface into **objects of interest** and **supporting tools**.



# Principles of Direct Manipulation

**Objects of interest:** visual representation of the data

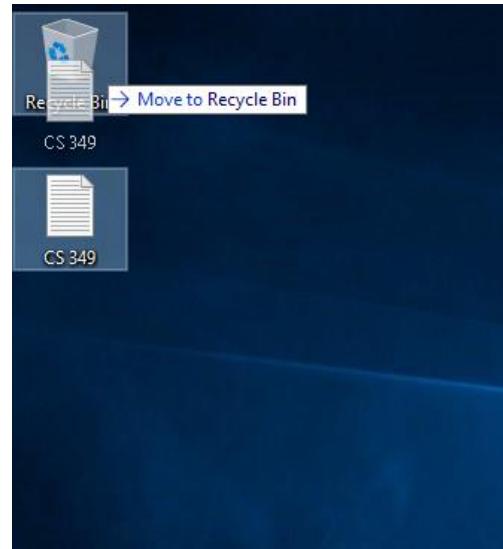
**Supporting tools:** visual widget that enables manipulating objects of interest



# Direct Manipulation – Advantages

While interacting with direct manipulation interfaces, users feel as if they are interacting with the domain rather than with the interface, so they focus on the task rather than on the technology.

There is a feeling of direct involvement with a world of objects of interest rather than communication with an intermediary.

A screenshot of a Microsoft Command Prompt window titled "C:\ Command Prompt". The window shows the following text:

```
Microsoft Windows [Version 10.0.10586]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\cs349>del ".\Desktop\CS 349.txt"
```

# Direct Manipulation – Analogy to the Real World

Real World Interaction:

- Object to be discarded
  - Move hand to object
  - Pick up object with hand
- Waste basket
  - Move hand with object to waste basket
  - Release object from hand

DM Interface

- Icon of object of interest (OoI) to be discarded
  - Move pointer to OoI
  - Click button to select OoI
- Waste basket icon
  - Drag icon of OoI to waste basket icon
  - Release button to deselect OoI

# Direct Manipulation – Affordance



# Direct Manipulation – Affordance

Perceived affordance: what the user believes that they can do with an object, based on its appearance.

Actions that are suggested by the object.



Affordances in the interface are like affordances for analogous actions in the real world: they should build on existing experiences and intuitions to aid learning.

# Direct Manipulation – Adaption to the Desktop

How does this apply to WIMP interfaces?

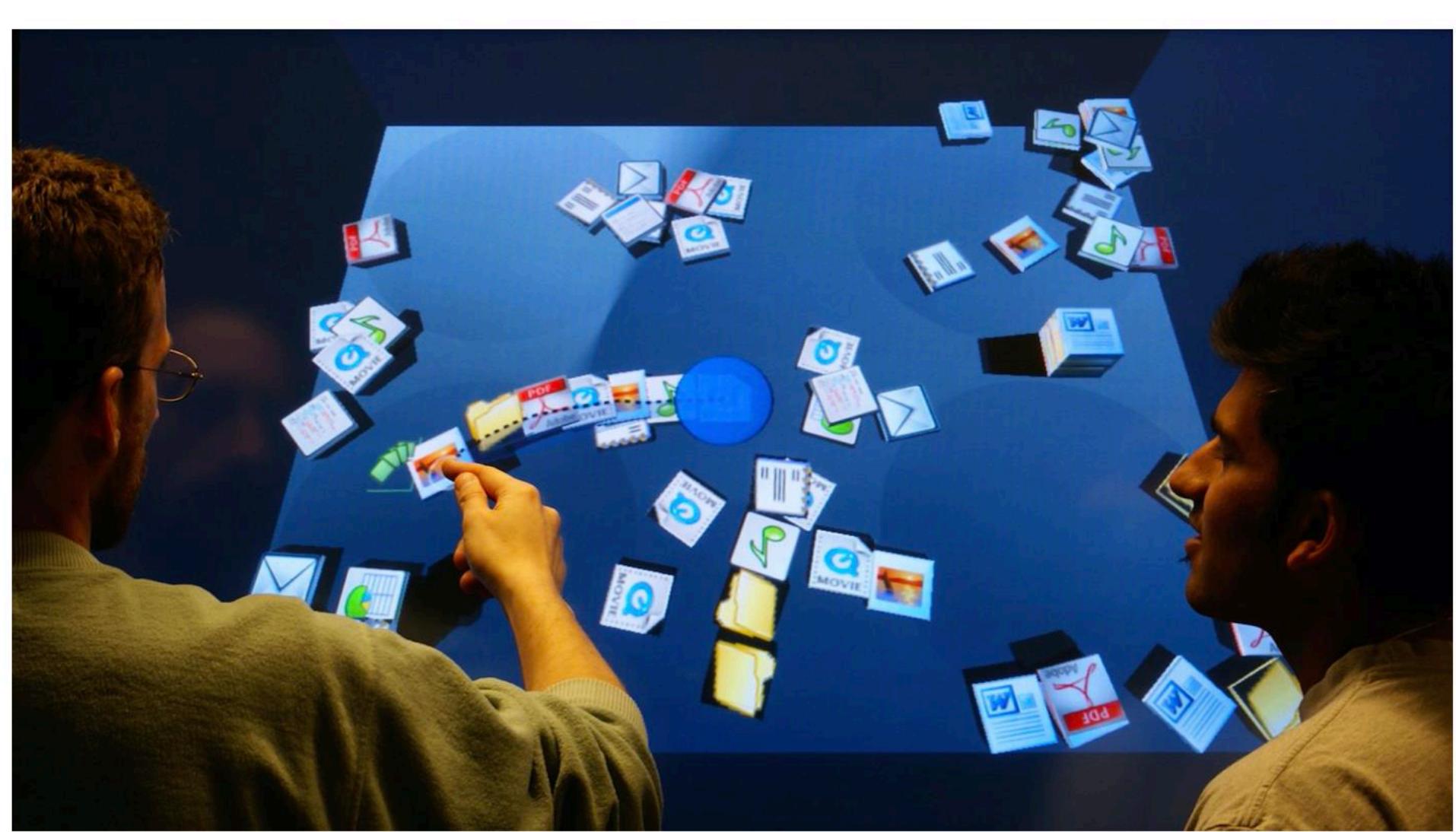
- Create objects of interests and supporting tools
- Allow for physical actions where possible.

Where do we see Direct Manipulation?

- Working surface / Desktop
- Grab and move / Click and drag to move elements (e.g., icons, files)
- Grab to engage / Click to activate (e.g., buttons, toolbars, other objects)

Desktop metaphor is prevalent

- It does not consistently use DM / there is a degree of “DM-ness”



## BumpTop - A Multi Touch 3D Physics Desktop

<https://github.com/bumptop/BumpTop/wiki>

<https://www.youtube.com/watch?v=eqcmPJ-oVL0>

# **Direct Manipulation – Adaption to the Desktop**

Always good? Always bad? It depends...

# Direct Manipulation – Adaption to the Desktop

Modern GUIs do not always use direct manipulation (Desktop interfaces, frankly, do a poor job of modeling DM):

- Due to the complexity / possibilities of the digital world, many objects of interest have properties that cannot be easily represented using DM
- Many commands are invoked indirectly
  - Menus, dialog boxes, toolbars are not direct manipulation ... they are “tools” that pull users away from objects of interest
- Many objects in the interface are not objects of interest
  - Toolbar palettes

# **Direct Manipulation – Adaption to the Desktop**

Are there times to not use Direct Manipulation?

There are also very good reasons to deliberately break away from DM:

- Visually impaired users cannot see the graphics; no linear flow for screen readers; physically impaired may have difficulty with required movements
- Consumes valuable screen space, forcing valuable information off-screen
- Switching between keyboard and pointer is time consuming
- Analogies may not be clear
  - Users need to learn meaning of visual representations
  - Visual representations may be misleading

U

CS 349

# Instrumental Interaction



# Interaction Model

“An interaction model is a set of principles, rules, and properties that guide the design of an interface. It describes **how to combine interaction techniques in a meaningful and consistent way** and defines the look and feel of the interaction from the user's perspective. Properties of the interaction model can be used to evaluate specific interaction designs.”

— Michel Beaudouin-Lafon. 2000. *Instrumental interaction: an interaction model for designing post-WIMP user interfaces*. In *Proc of the CHI '00*, 446–453. DOI: 10.1145/332040.332473

# Instrumental Interaction

“The instrumental interaction model is based on how we naturally use tools (or instruments) to manipulate objects of in the physical world. Objects of interest are called *domain objects* and are manipulated with computer artifacts called *interaction instruments*.”

— Michel Beaudouin-Lafon. 2000. *Instrumental interaction: an interaction model for designing post-WIMP user interfaces*. In *Proc of the CHI '00*, 446–453. DOI: 10.1145/332040.332473

With instrumental interaction, interfaces have *domain objects* and *interaction instruments*

- Domain objects: the thing of interest, data and associated attributes, which is manipulated using an interaction instrument
- Interaction instrument: a necessary mediator between the user and domain objects

# Instrumental Interaction

With instrumental interaction, interfaces have *domain objects* and *interaction instruments*

Goal:



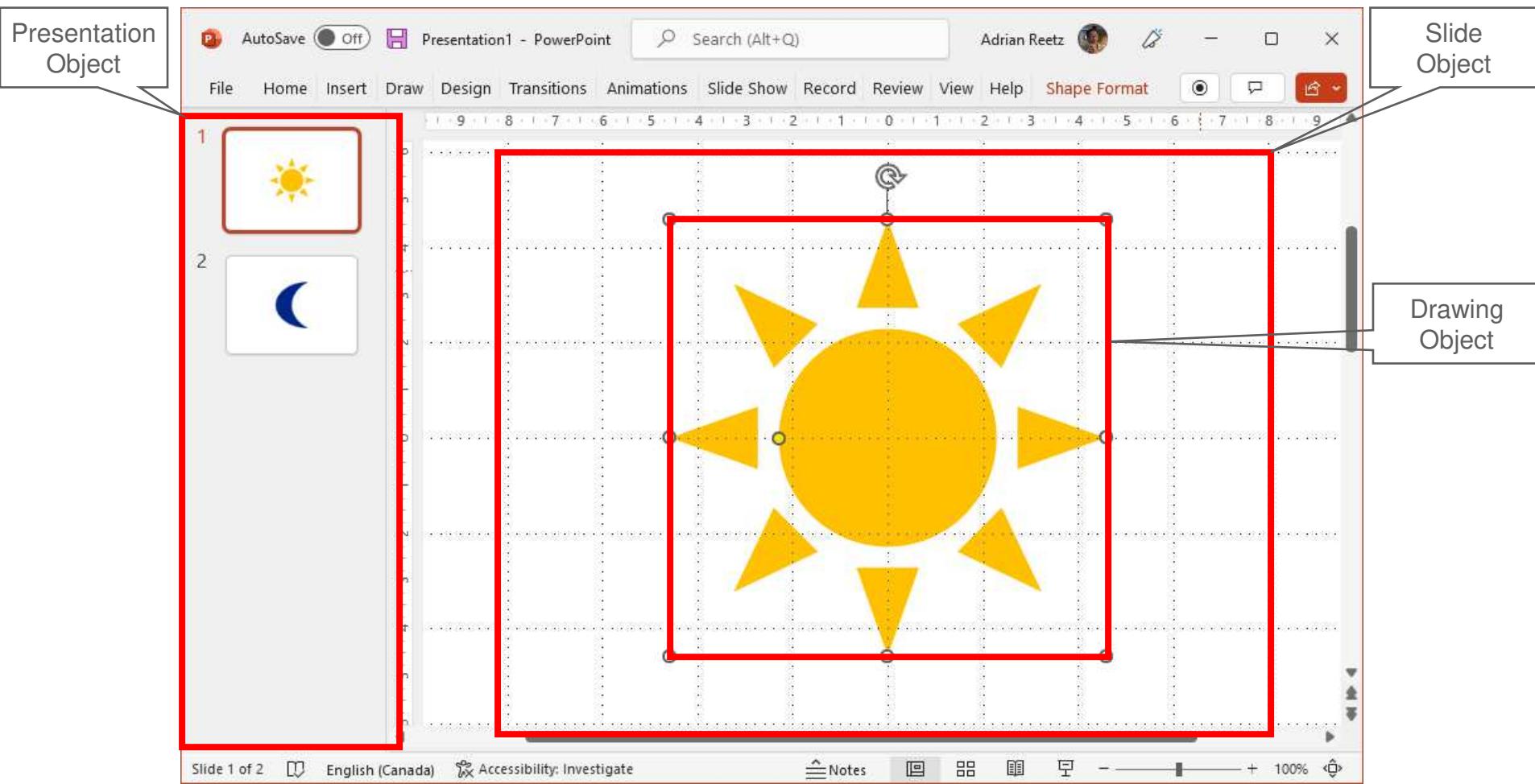
- Domain objects: the thing of interest, data and associated attributes, which is manipulated using an interaction instrument



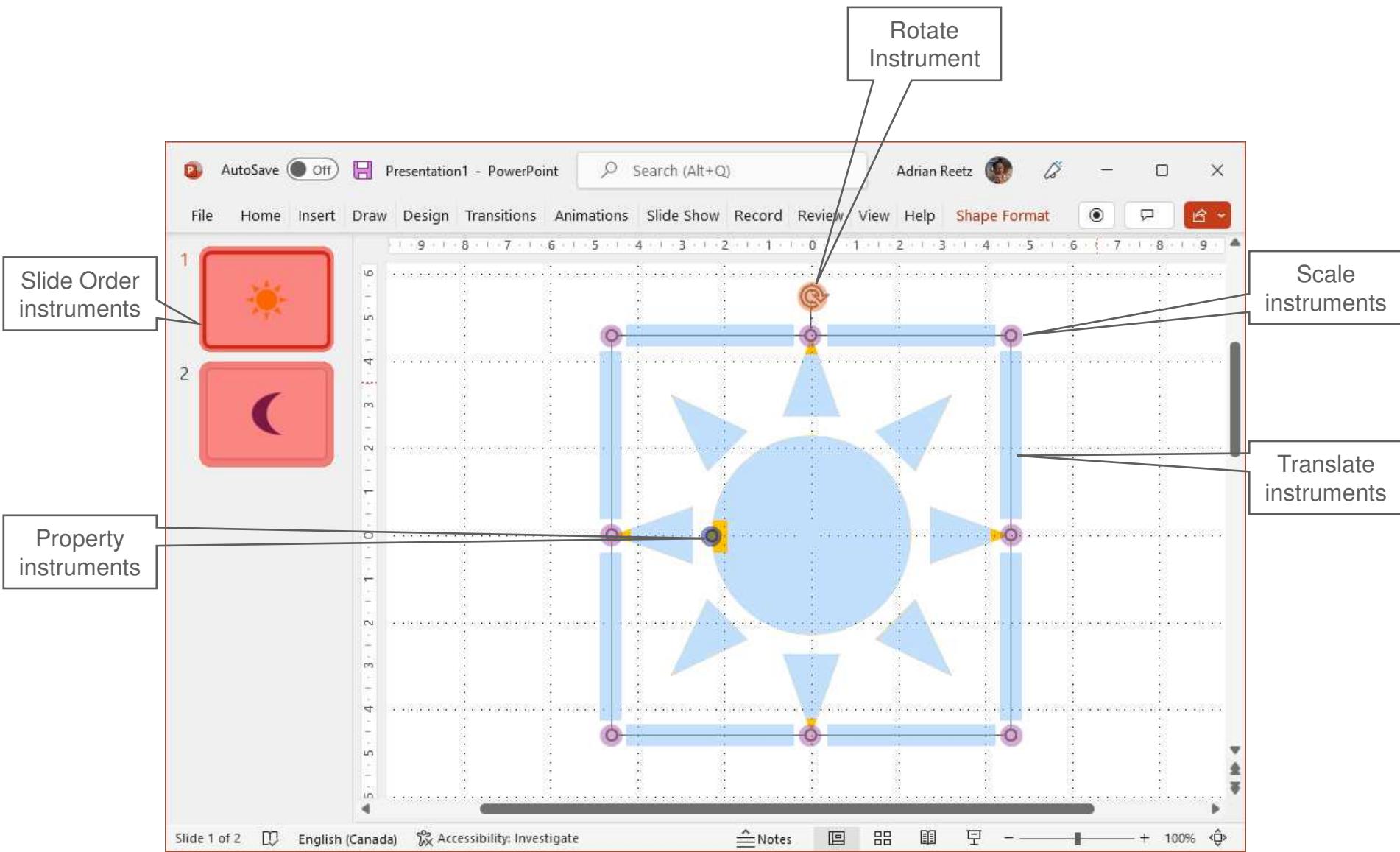
- Interaction instrument: a necessary mediator between the user and domain objects



# Instrumental Interaction – Domain Objects



# Instrumental Interaction – Interaction Instruments



# Instrumental Interaction – Activation

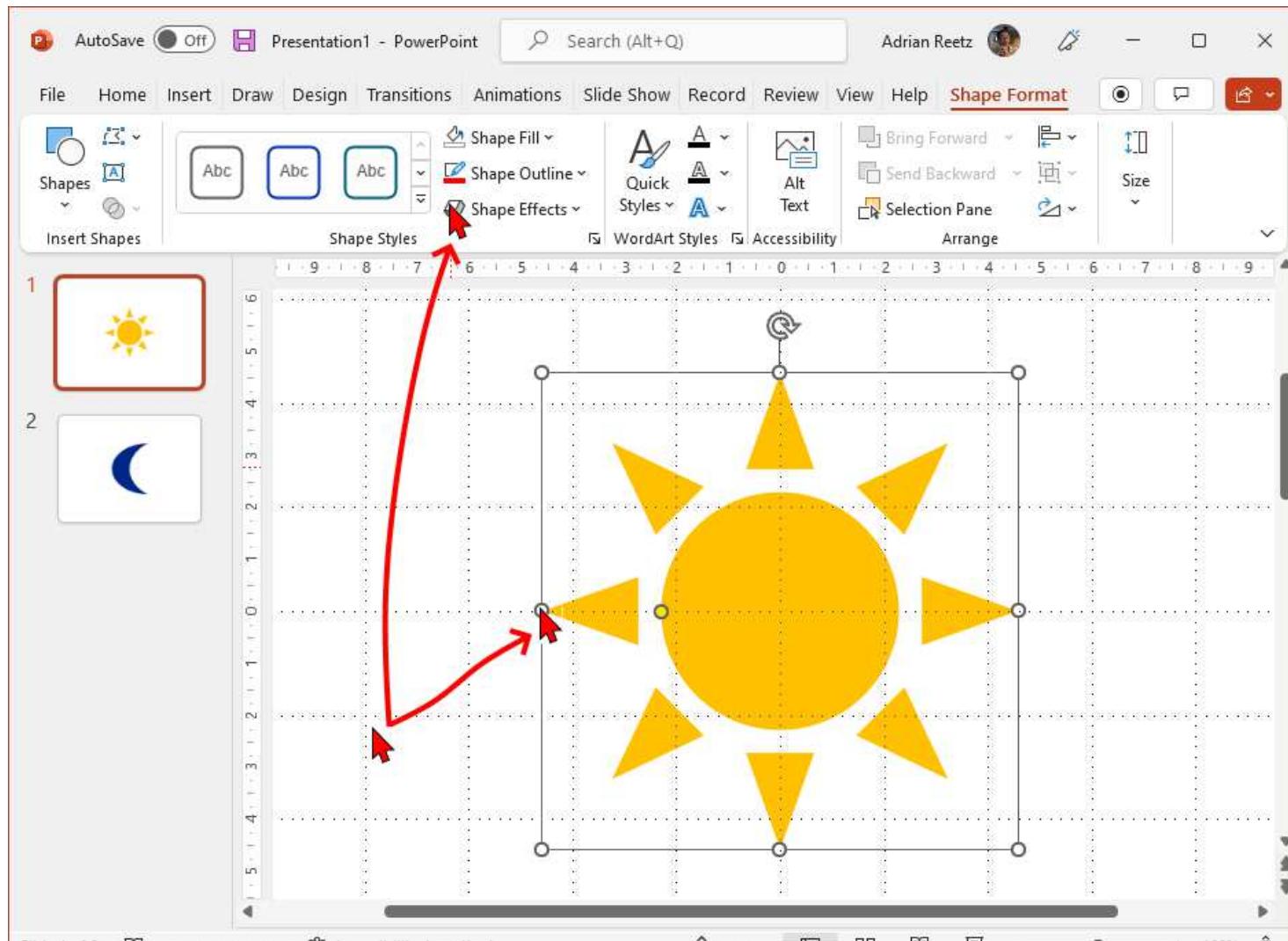
Activation = triggering functionality

Interaction instruments can be activated **spatially** and **temporally**

- Spatial activation has a **movement cost**
- Temporal activation has a **time cost**

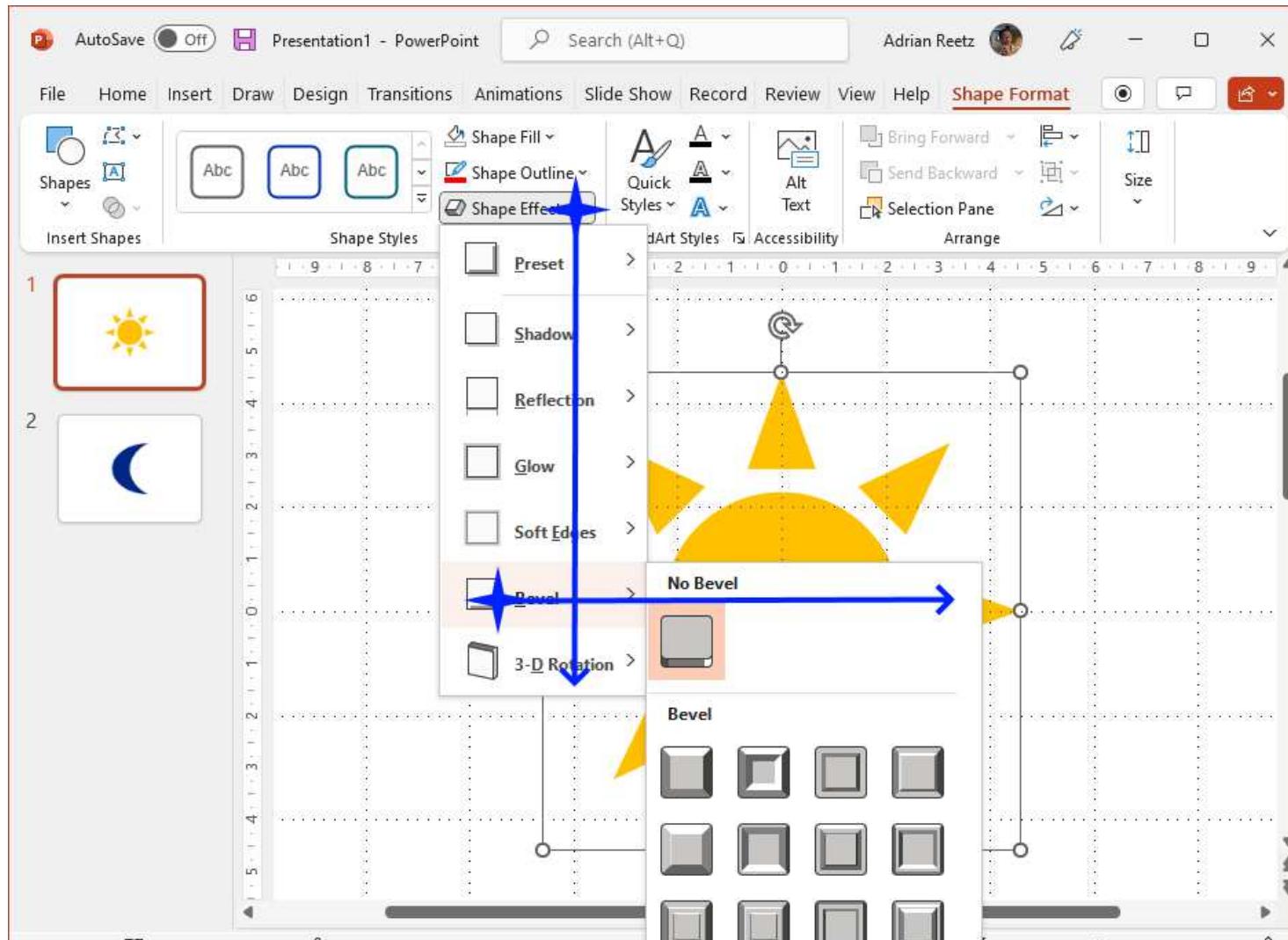
# Instrumental Interaction – Activation

Spatial activation has a **movement cost**.



# Instrumental Interaction – Activation

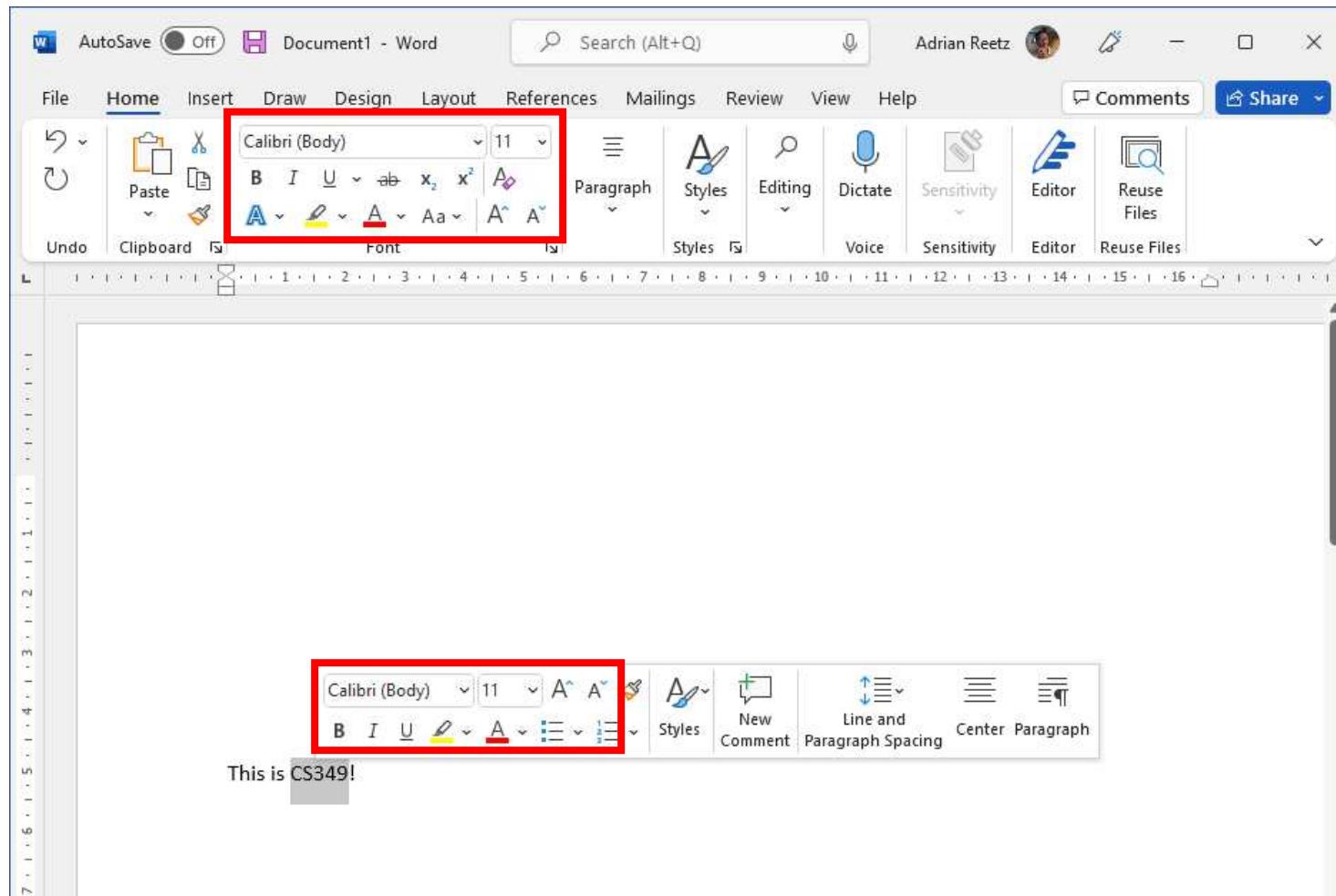
Temporal activation has a **time cost**.



# Instrumental Interaction – Activation

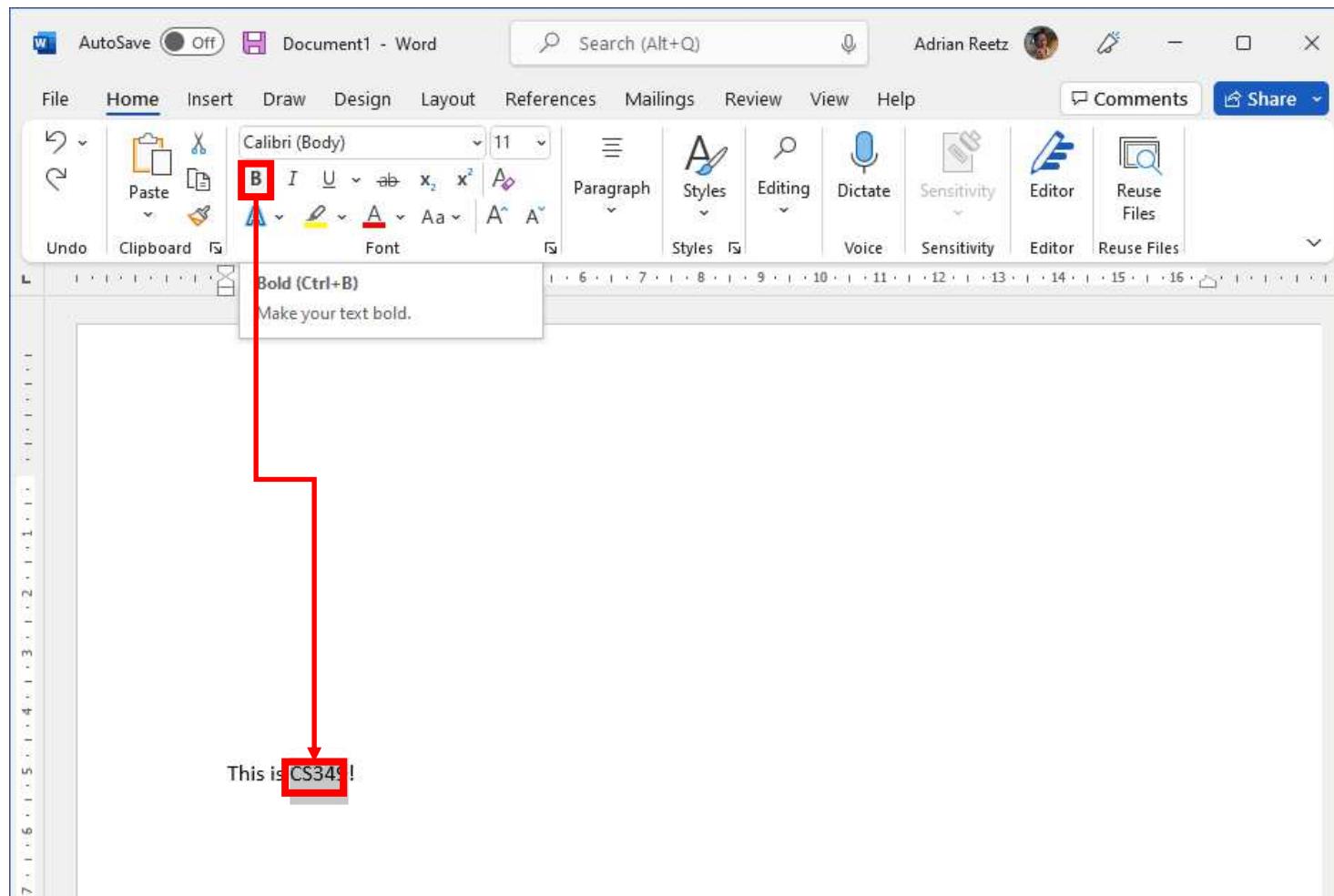
UI layout and design is concerned with the tradeoff of these costs.

The more frequently an instrument is used, the lower its activation cost should be.



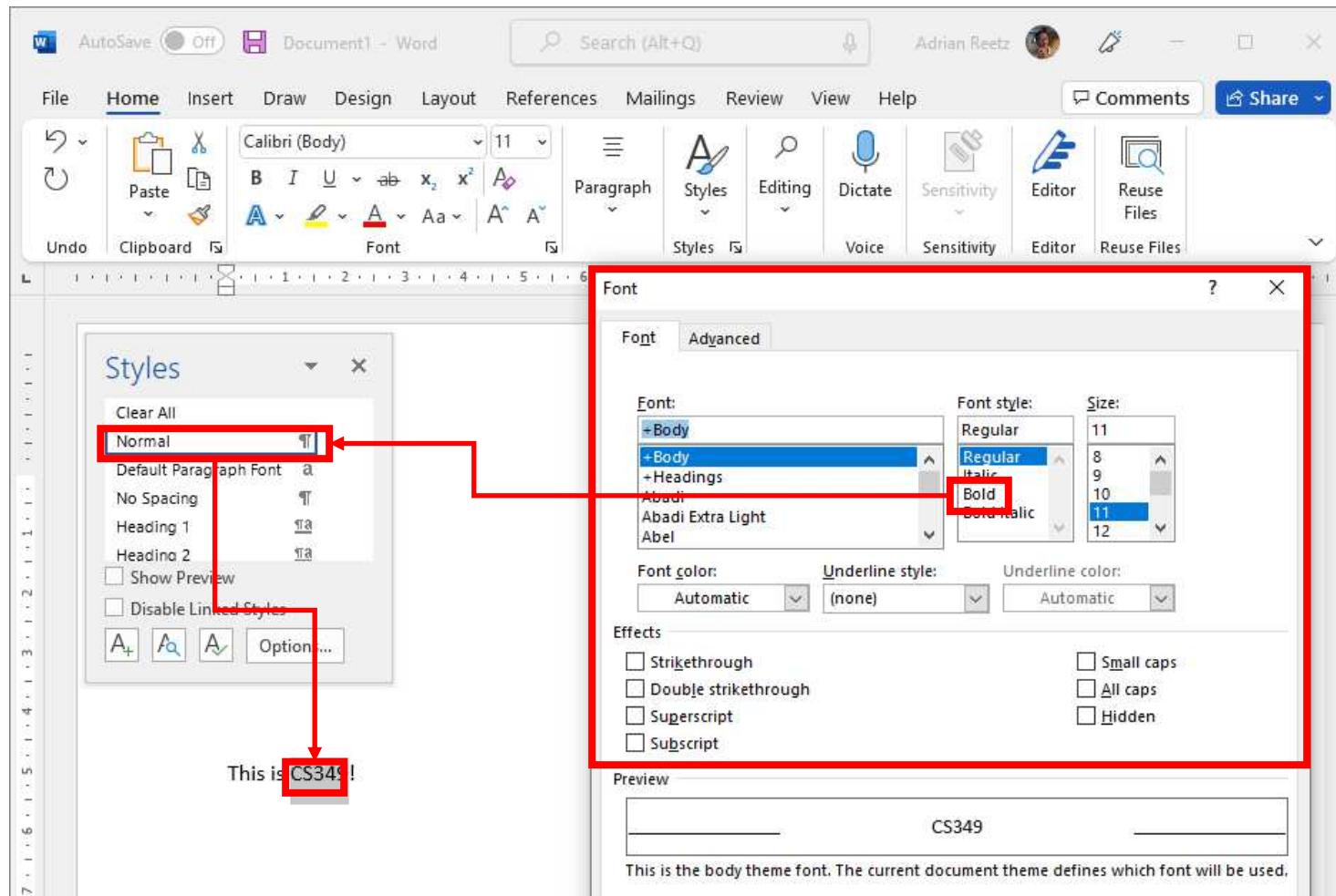
# Instrumental Interaction – Reification

Below, a property (font weight) of a domain object (“CS349”) is changed (using the interaction instrument “[B]old” button).



# Instrumental Interaction – Reification

Below, the property of an interaction instrument (Style “Normal”) is changed (using the interaction instrument “Font” window).



# Instrumental Interaction – Reification

**Reification:** turning concepts into something concrete.

For example, the single concept of “text style” is a reification of multiple concepts, such as, font type, font weight, font size, alignment, baseline spacing, letter spacing, word spacing, kerning, shifting, direction, orientation, ...

This reification can become a domain object in itself!

Domain object “CS349”      ↫ interaction instrument “Style”,

Domain object “Style ‘Normal’”    ↫ interaction instrument “Font style selection box”.

# Instrumental Interaction – Meta-Instruments

**Meta-instrument:** an instrument that acts on reified domain object.

Domain object “CS349”      ↫ interaction instrument “Style”,

Domain object “Style ‘Normal’” ↫ **meta** interaction instrument “Font style’ selection box”.

# Instrumental Interaction – Meta-Instruments

Domain object “CS349”

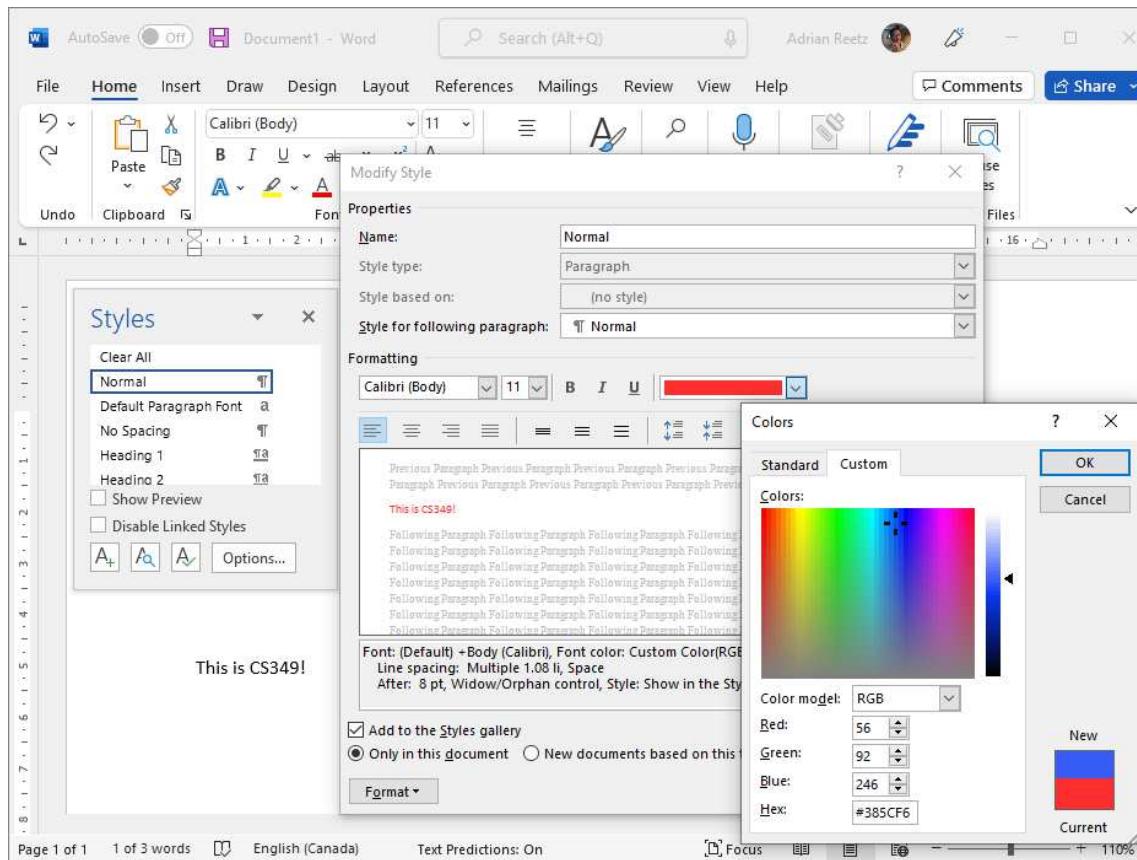
→ interaction instrument “Style”,

Domain object “Style ‘Normal’”

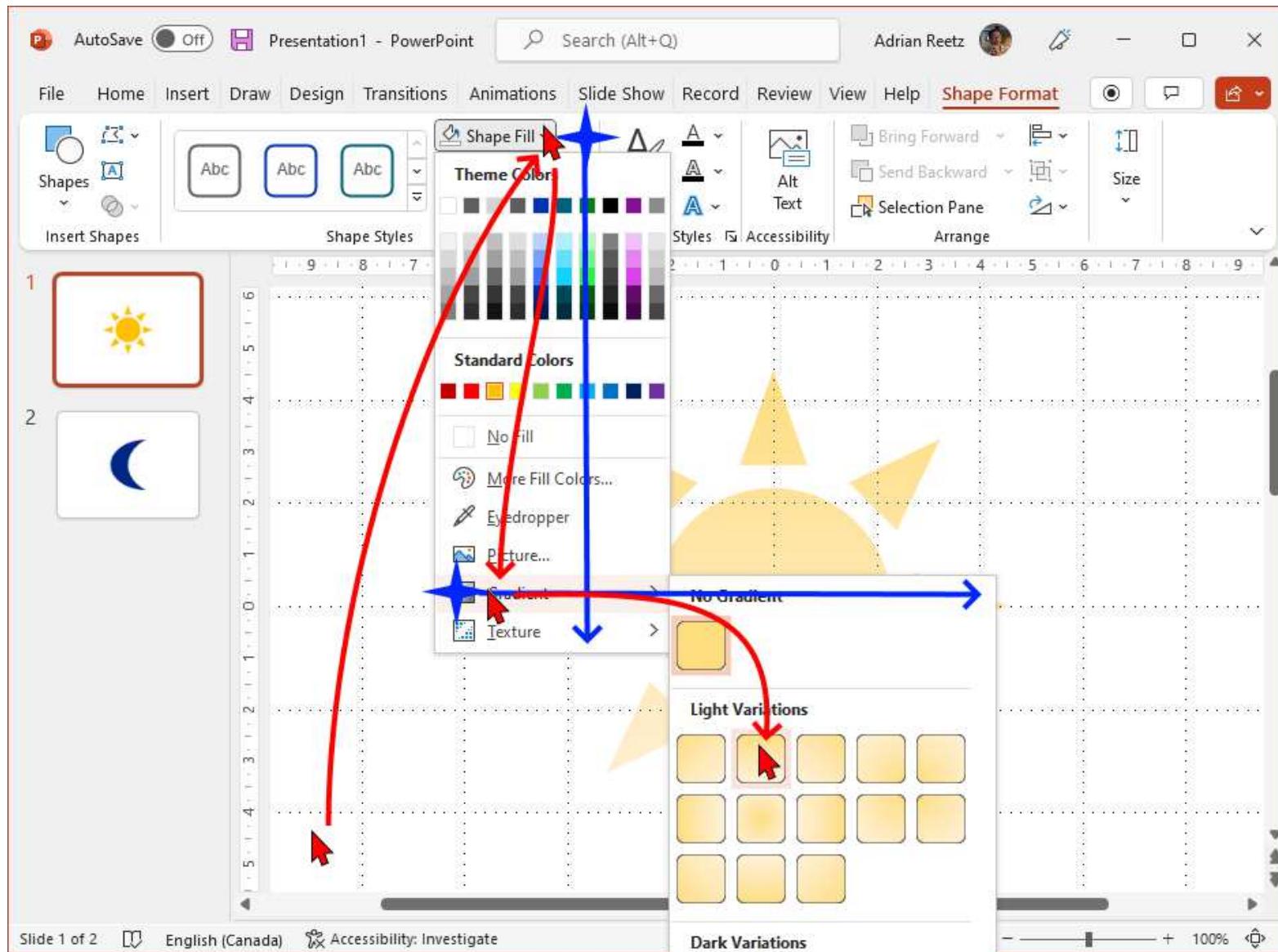
→ **meta** interaction instrument “Color selection box”.

Domain object “Color ‘Red’”

→ **meta** interaction instrument “Color picker”



# Instrumental Interaction – Evaluation



# **Instrumental Interaction – Evaluation**

How do we describe instruments? How do we evaluate their effectiveness?

## Degree of Indirection

- Spatial / temporal offset between instrument and action on object

## Degree of Integration

- Suitability of input device for manipulating instrument
- (Ratio of degrees of freedom of instrument to degrees of freedom of input device)

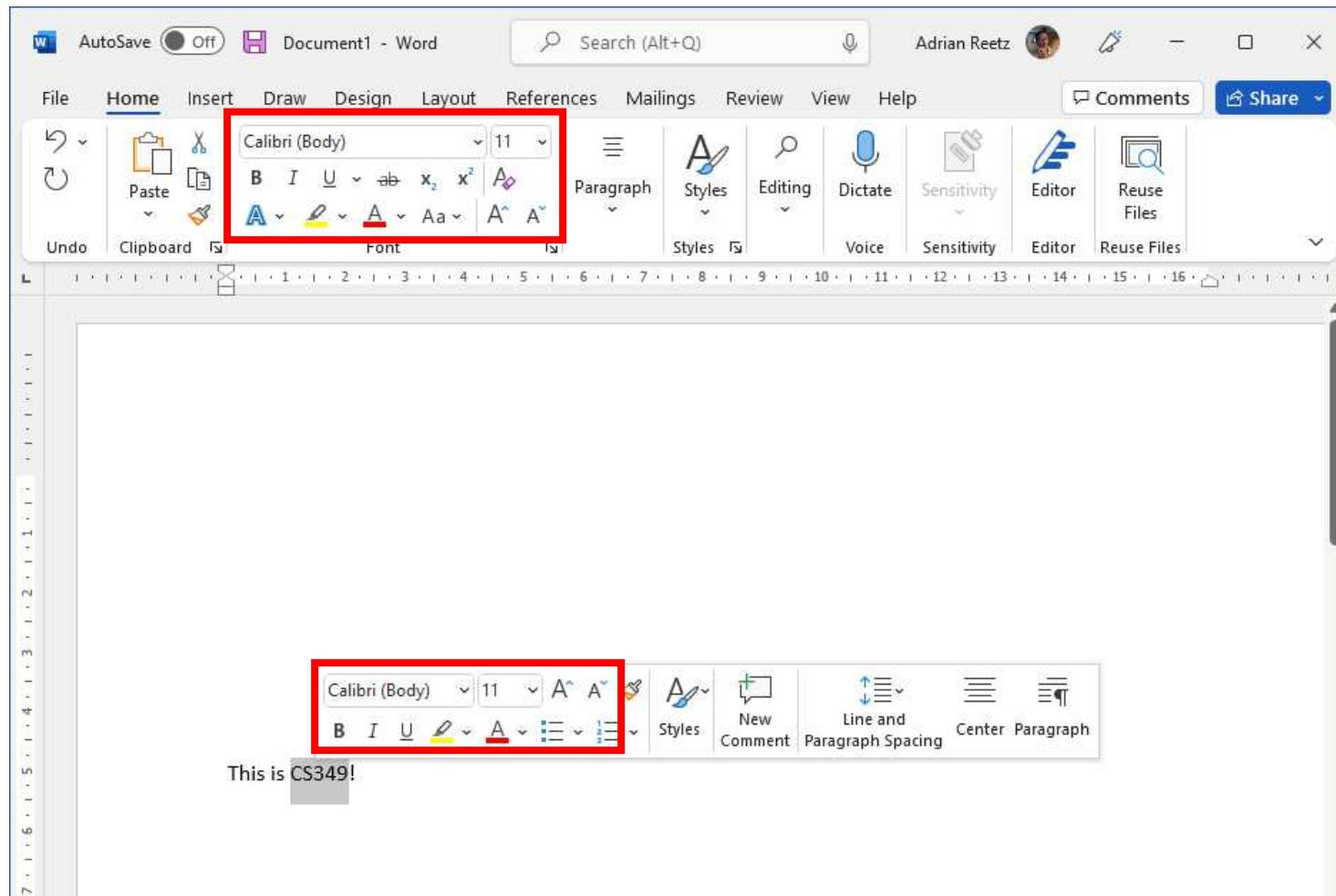
## Degree of Compatibility

- Similarity of action on control device/instrument to action on object

# Instrumental Interaction – Degree of Indirection

Near, e.g., drag to translate,  
handles on to resize

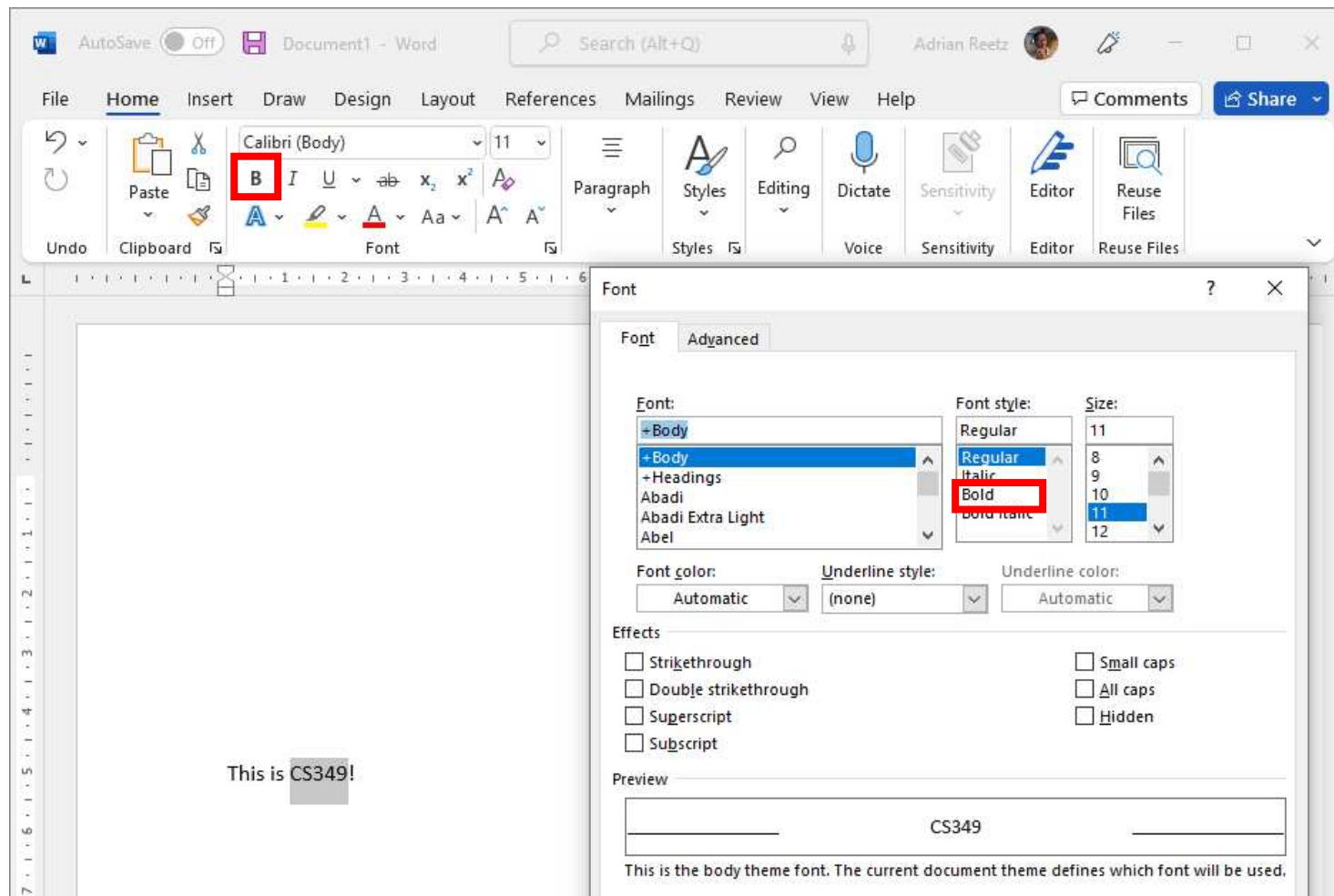
Far, e.g., menu bar, scroll bar



# Instrumental Interaction – Degree of Indirection

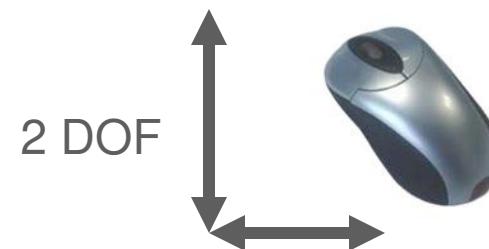
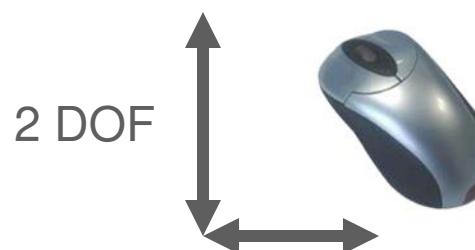
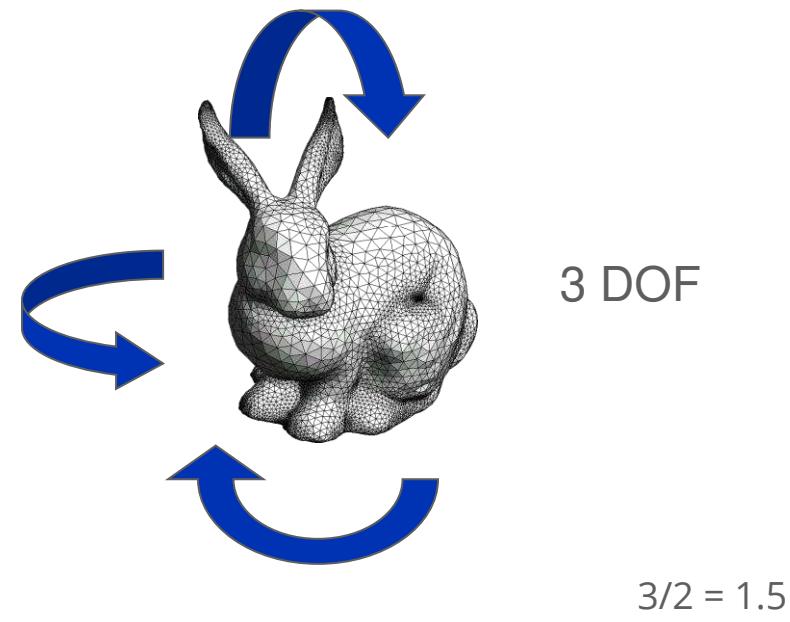
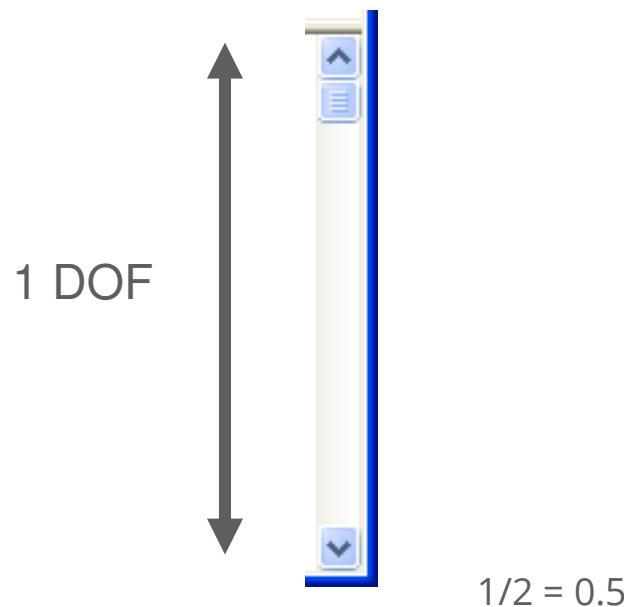
Short, e.g., directly accessible buttons; less information to process

Long, e.g., menu activation, animation; more information to process



# Instrumental Interaction – Degree of Integration

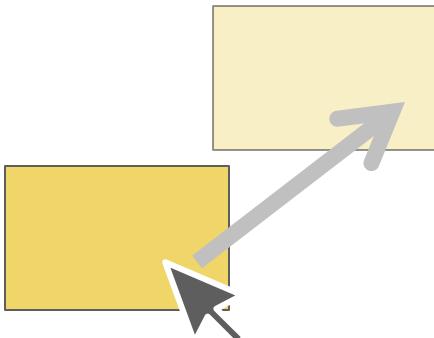
Degree of integration is the ratio between the number of degrees of freedom (DOF) of the instrument and the DOF captured by input device (reflects suitability)



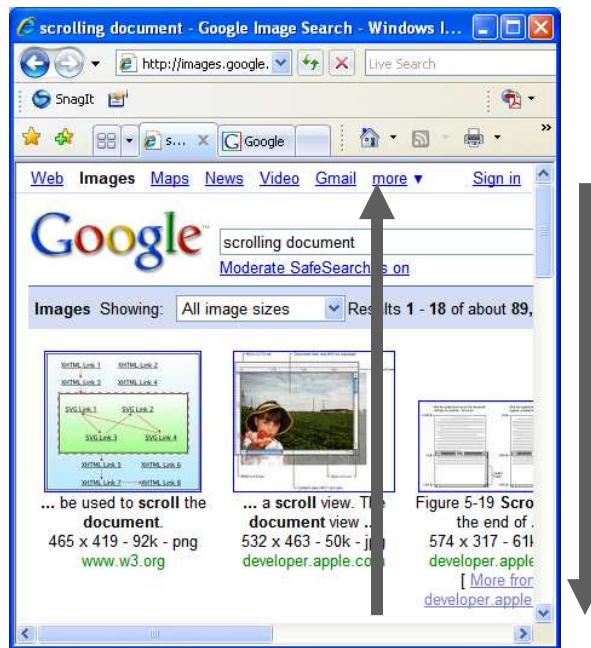
# Instrumental Interaction – Degree of Compatibility

Degree of Compatibility is the similarity between the physical action on the instrument and the response of the object (similarity makes actions feel natural or intuitive).

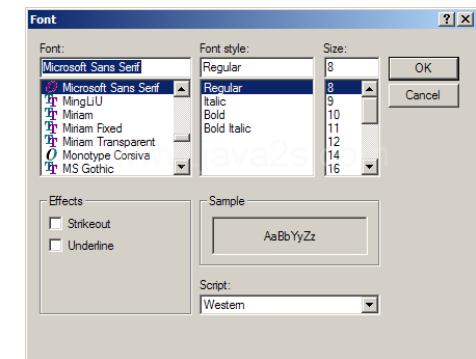
Dragging = high



Scrolling = medium



Dialog = low



# Direct Manipulation & Instrumental Interaction

A direct manipulation interface allows a user to directly act on a set of objects in the interface.

- Low degree of indirection (i.e., low spatial and temporal offsets)
- High degree of integration (1:1 correspondence)
- High degree of compatibility (similarity of action and effect)

Direct means instruments are visually indistinguishable from objects they control

- The actions on instrument / object entities are analogous to actions on similar objects in the real world.
- The actions on instrument/object entities preserve the conceptual linkage between instrument and object.



Bret Victor, Inventing on Principle (talk from CUSEC 2012)

[https://www.student.cs.uwaterloo.ca/~cs349/videos/Bret Victor - Inventing on Principle-HD.mp4](https://www.student.cs.uwaterloo.ca/~cs349/videos/Bret%20Victor%20-%20Inventing%20on%20Principle-HD.mp4)

# End of the Chapter



- Direct Manipulation
  - what it is
  - its components
  - its advantages and disadvantages
- Instrumental Interaction
  - what it is
  - its components
  - its advantages and disadvantages

-



X

# Drawing

Primitives

Graphics Context

The Painter's Algorithm

U

CS 349

February 13

-



X

# Primitives

U

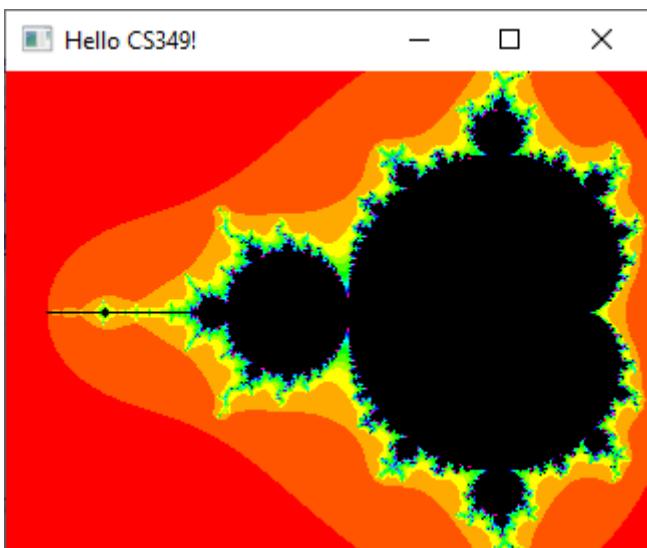
CS 349

# Drawing Primitives – Pixel

Three conceptual models for drawing:

## 1. Pixel

- `SetPixel(x, y, color)`
- `DrawImage(x, y, image_source)`



# Drawing Primitives – Strokes

Three conceptual models for drawing:

## 2. Stroke

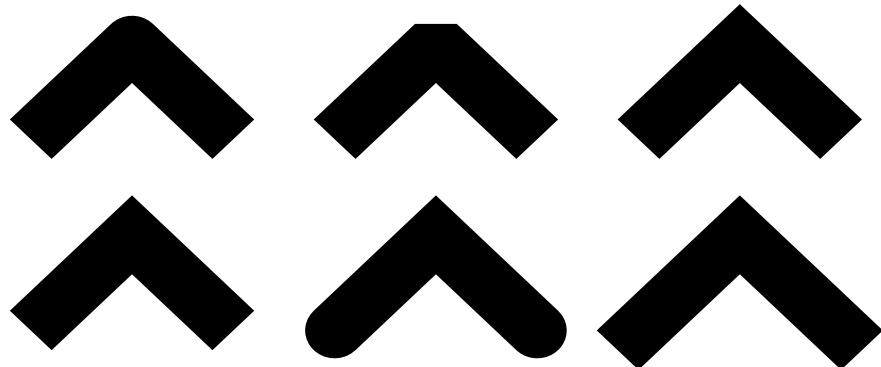
- `DrawLine(x1, y1, x2, y2, line_style)`
- `DrawRect(x, y, width, height, border_style)`
- `DrawPolyline(x1, y1, x2, y2, ..., xn, yn, border_style)`
- `DrawArc(x, y, width, height, start, end, border_style)`



# Drawing Primitives – Stroke Styles

Many options:

- Colour
- Thickness
- Style: solid, dashed
- Joints: round, bevel, miter
- Cap: butt, round square



# Drawing Primitives – Regions

Three conceptual models for drawing:

## 3. Region

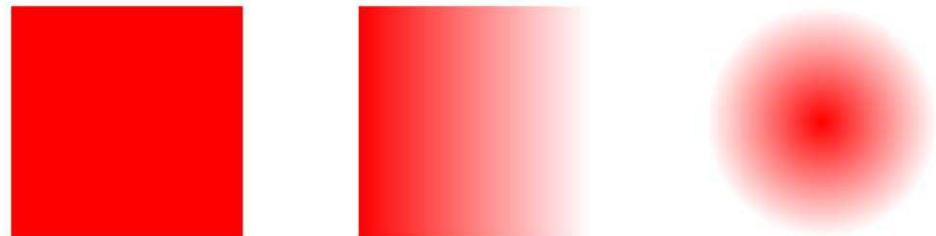
- `DrawText("CS349", x, y, text_style)`
- `DrawRect(x, y, width, height, border_style, fill_style)`



# Drawing Primitives – Region Styles

Some fill options:

- Colour: solid, gradient, pattern
- Opacity



Some text options:

- Family or Name
- Size
- Weight: thin, ..., black
- Slope: normal, italic

a b c d e

# JavaFX Primitives

Primitives drawing commands include:

- Rectangles: `strokeRect()`, `fillRect()`
- Rounded rectangles: `strokeRoundRect()`, `fillRoundRect()`
- Oval (and Circle): `strokeOval()`, `fillOval()`
- Polygons: `strokePolygon()`, `fillPolygon()`
- Arcs: `strokeArc()`, `fillArc()`
- Text: `strokeText()`, `fillText()`
- Line: `strokeLine()`, `strokePolyline()`

-



X

# Graphics Context

U

CS 349

# JavaFX Canvas and Graphics Context

The JavaFX Canvas Node provides a canvas for drawing onto:

- it has a buffer where the drawing is rendered
- It has a single Graphics Context that is modified to set drawing parameters state (e.g., border\_style) and issue drawing commands to.

```
val canvas = Canvas(320.0, 240.0)
canvas.graphicsContext2D.apply {

    stroke = Color.RED
    LineWidth = 7.0
    strokeLine(15.0, 10.0, 90.0, 55.0)

    stroke = Color.PINK
    LineWidth = 1.5
    font = Font.font("Console", 48.0)
    strokeText("CS349", 170.0, 190.0, 180.0)

    drawImage(Image("plumber.png"), 80.0, 10.0)
}
```

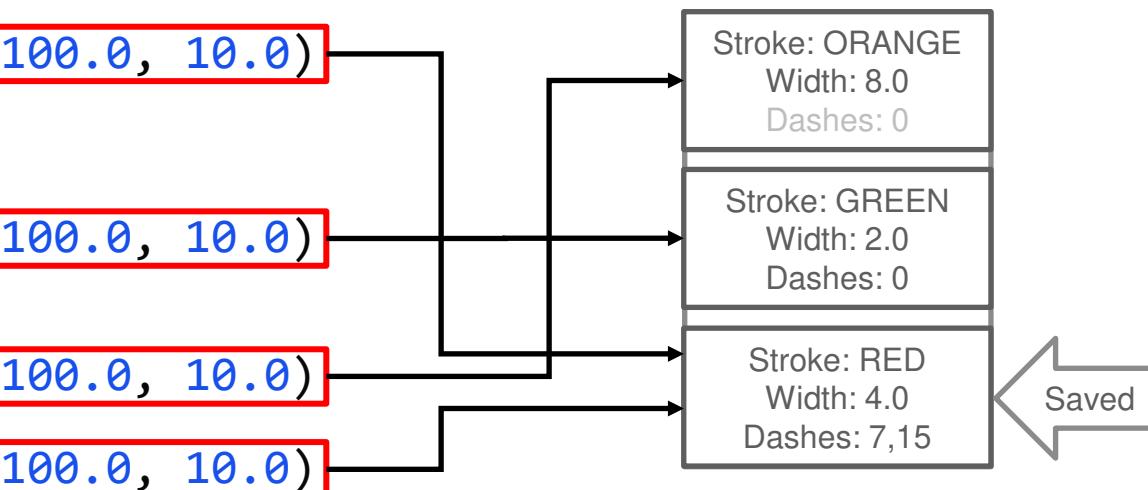


# JavaFX Canvas and Graphics Context

A graphics context contains drawing parameters and all device-specific information that the drawing system needs to perform any subsequent drawing commands.

The graphics content can be pictured as a stack, and it is possible to save and restore previous states.

```
stroke = Color.RED  
LineWidth = 4.0  
setLineDashes(7.0, 15.0)  
save()  
strokeRect(10.0, 10.0, 100.0, 10.0)  
stroke = Color.GREEN  
LineWidth = 2.0  
setLineDashes(0.0)  
strokeRect(10.0, 30.0, 100.0, 10.0)  
stroke = Color.ORANGE  
LineWidth = 8.0  
strokeRect(10.0, 50.0, 100.0, 10.0)  
restore()  
strokeRect(10.0, 70.0, 100.0, 10.0)
```



# JavaFX Canvas and Graphics Context

A common approach is to maintain the current state of all drawing options in a *Graphics Context*. A drawing command, e.g., `strokeLine`, is then rendered using the current set of options.

```
stroke = Color.RED  
LineWidth = 10.0  
strokeLine(50.0, 25.0, 150.0, 125.0)
```

```
LineWidth = 20.0  
strokeLine(100.0, 25.0, 100.0, 125.0)
```

```
stroke = Color.GREEN  
strokeLine(150.0, 25.0, 50.0, 125.0)
```

```
LineWidth = 10.0  
LineCap = StrokeLineCap.ROUND  
setLineDashes(25.0)  
strokeLine(25.0, 75.0, 175.0, 75.0)
```



# JavaFX Canvas and Graphics Context

Graphics context attributes include:

- Fill options
- Stroke options
- Text option
- Rendering: clipping, blend, transforms

# Drawing using Graphics Context

1. Create Canvas
2. Use the *graphicsContext2D* to set drawing attributes
3. Use *graphicsContext2D* to draw shapes

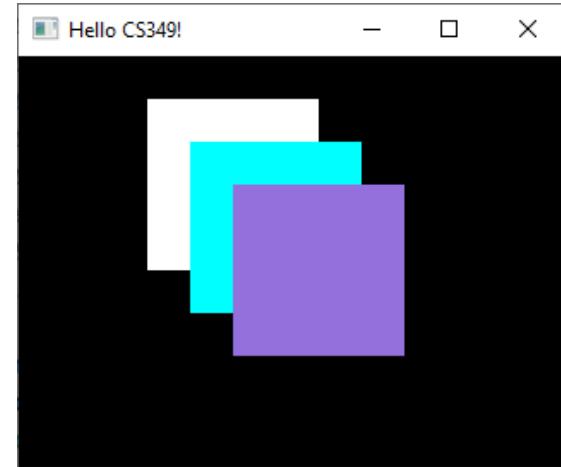
```
override fun start(stage: Stage) {  
    val canvas = Canvas(320.0, 240.0)  
    canvas.graphicsContext2D.apply {  
        stroke = Color.WHITE  
        fill = Color.WHITE  
        fillRect(75.0, 25.0, 100.0, 100.0)  
        fill = Color.AQUA  
        fillRect(100.0, 50.0, 100.0, 100.0)  
        fill = Color.MEDIUMPURPLE  
        fillRect(125.0, 75.0, 100.0, 100.0)  
    }  
    stage.apply {  
        scene = Scene(Group(canvas), 320.0, 240.0, Color.BLACK)  
        title = "Hello CS349!"  
    }.show()  
}
```



# Drawing using Convenience Classes

JavaFX Nodes include Shape classes, which can be drawn directly, i.e., without using a Canvas.

```
override fun start(stage: Stage) {
    val rectList = listOf(
        Rectangle(100.0, 100.0, Color.WHITE).apply {
            x = 75.0; y = 75.0 },
        Rectangle(100.0, 100.0, Color.AQUA).apply {
            x = 100.0; y = 100.0 },
        Rectangle(100.0, 100.0, Color.MEDIUMPURPLE).apply {
            x = 125.0; y = 125.0 }
    )
    stage.apply {
        scene = Scene(Group(rectList), 320.0, 240.0, Color.BLACK)
        title = "Hello CS349!"
    }.show()
}
```



-



X

# The Painter's Algorithm

U

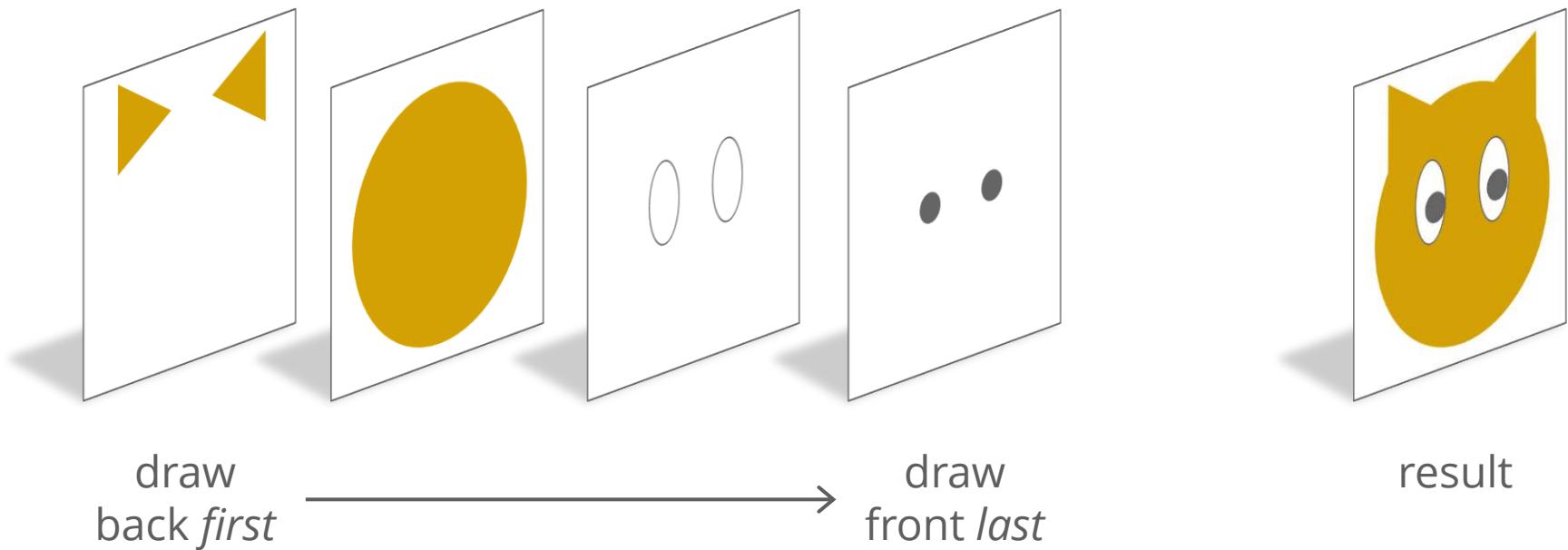
CS 349

# Painter's Algorithm

Basic graphics primitives are (really) *primitive*. To draw more complex shapes:

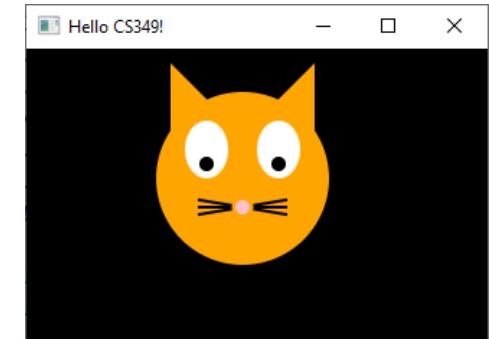
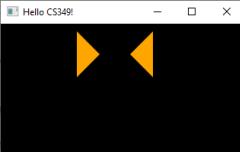
- Combine primitives
- Draw back-to-front, layering the image

This approach is called “Painter’s Algorithm”.



# Painter's Algorithm

```
canvas.graphicsContext2D.apply {  
    fill = Color.ORANGE  
    fillPolygon(mutableListOf(100.0, 130.0, 100.0).toDoubleArray(),  
               mutableListOf(10.0, 40.0, 70.0).toDoubleArray(), 3)  
    fillPolygon(mutableListOf(200.0, 170.0, 200.0).toDoubleArray(),  
               mutableListOf(10.0, 40.0, 70.0).toDoubleArray(), 3)  
    fillOval(90.0, 30.0, 120.0, 120.0)  
    fill = Color.WHITE  
    fillOval(110.0, 50.0, 30.0, 40.0)  
    fillOval(160.0, 50.0, 30.0, 40.0)  
    fill = Color.BLACK  
    fillOval(120.0, 75.0, 10.0, 10.0)  
    fillOval(170.0, 75.0, 10.0, 10.0)  
    stroke = Color.BLACK  
    lineWidth = 2.0  
    strokeLine(120.0, 105.0, 180.0, 115.0)  
    strokeLine(120.0, 110.0, 180.0, 110.0)  
    strokeLine(180.0, 105.0, 120.0, 115.0)  
    fill = Color.PINK  
    fillOval(145.0, 105.0, 10.0, 10.0)  
}
```



Amazing City spray paint art

<https://www.youtube.com/watch?v=k4gH7xx3tVc>

# Drawable Objects & Their Interface

Extend a shape class with Z-index information

```
class RectangleZ(x: Double, y: Double,  
                 width: Double, height: Double,  
                 fill: Paint, val zindex: Int,  
) : Rectangle(x, y, width, height) {  
    init {  
        this.fill = fill  
        stroke = Color.WHITE  
    }  
}
```

# List of Displayables

```
val rectList = listOf(  
    RectangleZ(75.0, 25.0, 100.0, 100.0, Color.WHITE, 1),  
    RectangleZ(125.0, 75.0, 100.0, 100.0, Color.MEDIUMPURPLE, 3)  
    RectangleZ(100.0, 50.0, 100.0, 100.0, Color.AQUA, 2),  
)  
  
val root = Group(rectList.sortedBy {  
    rectangleZ -> rectangleZ.zindex  
})
```



# End of the Chapter



- How to draw primitives using the Canvas
- How to draw primitives using Convenience Classes

-



X

# Graphics

Graphics Pipeline & Transformations

U

CS 349

February 15

# Graphics Pipeline & Transformations

U

CS 349

# Graphic Models and Images

Computer Graphics is the creation, storage, and manipulation of images and their models.

- **Model:** a mathematical representation of an image containing the important properties of an object (location, size, orientation, color, texture, etc.) in data structures
- **Rendering:** Using the properties of the model to create an image to display on the screen
- **Image:** The rendered model

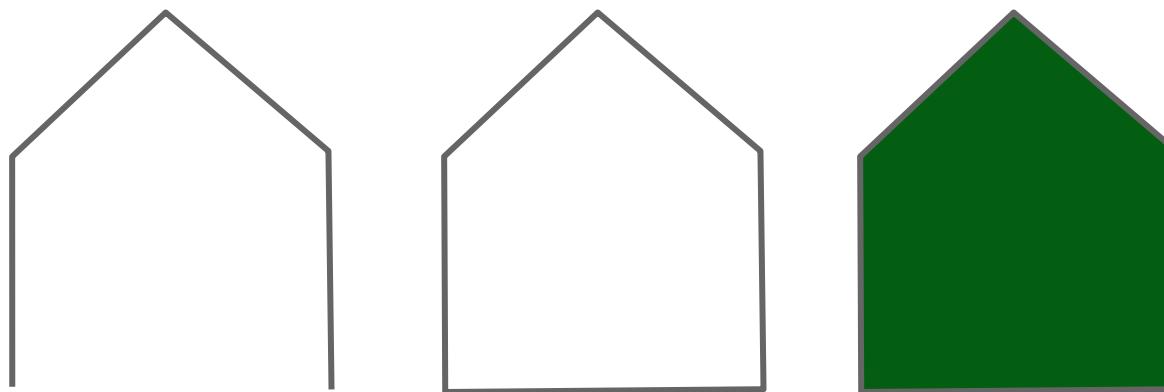


# Shape Model

An array of points (or vertices)  $\{P_1, P_2, \dots, P_n\}$  defines a shape.

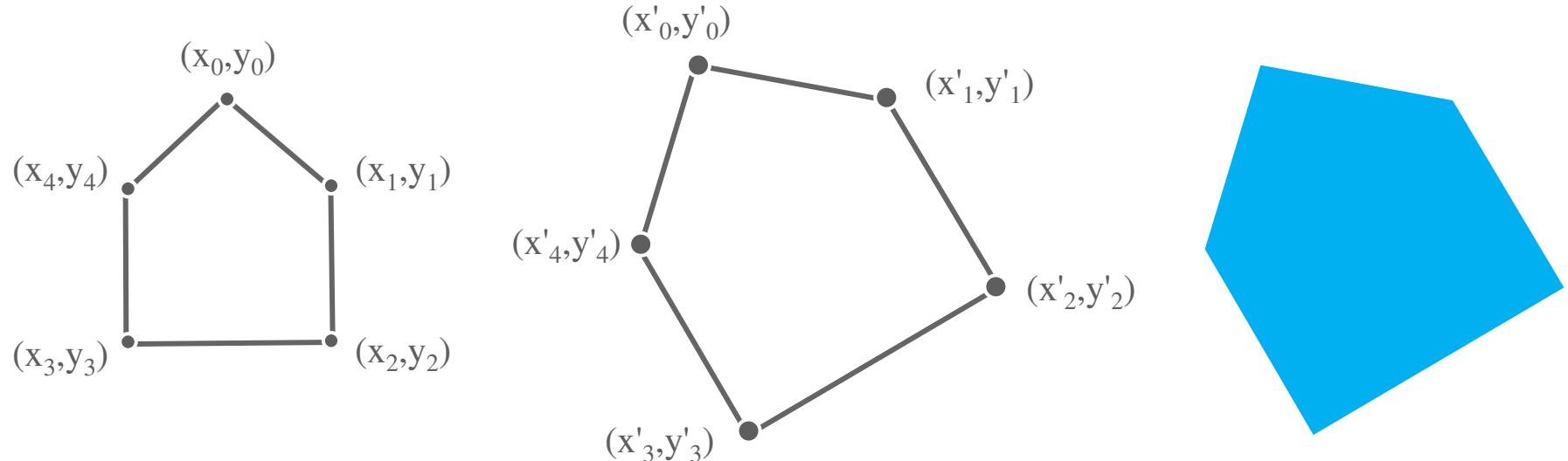
Properties that determine how the shape is drawn

- `isClosed` flag (shape is polyline or polygon)
- `isFilled` flag (polygon is filled or not)
- stroke thickness, colours, etc.



# Transforming Shape Models

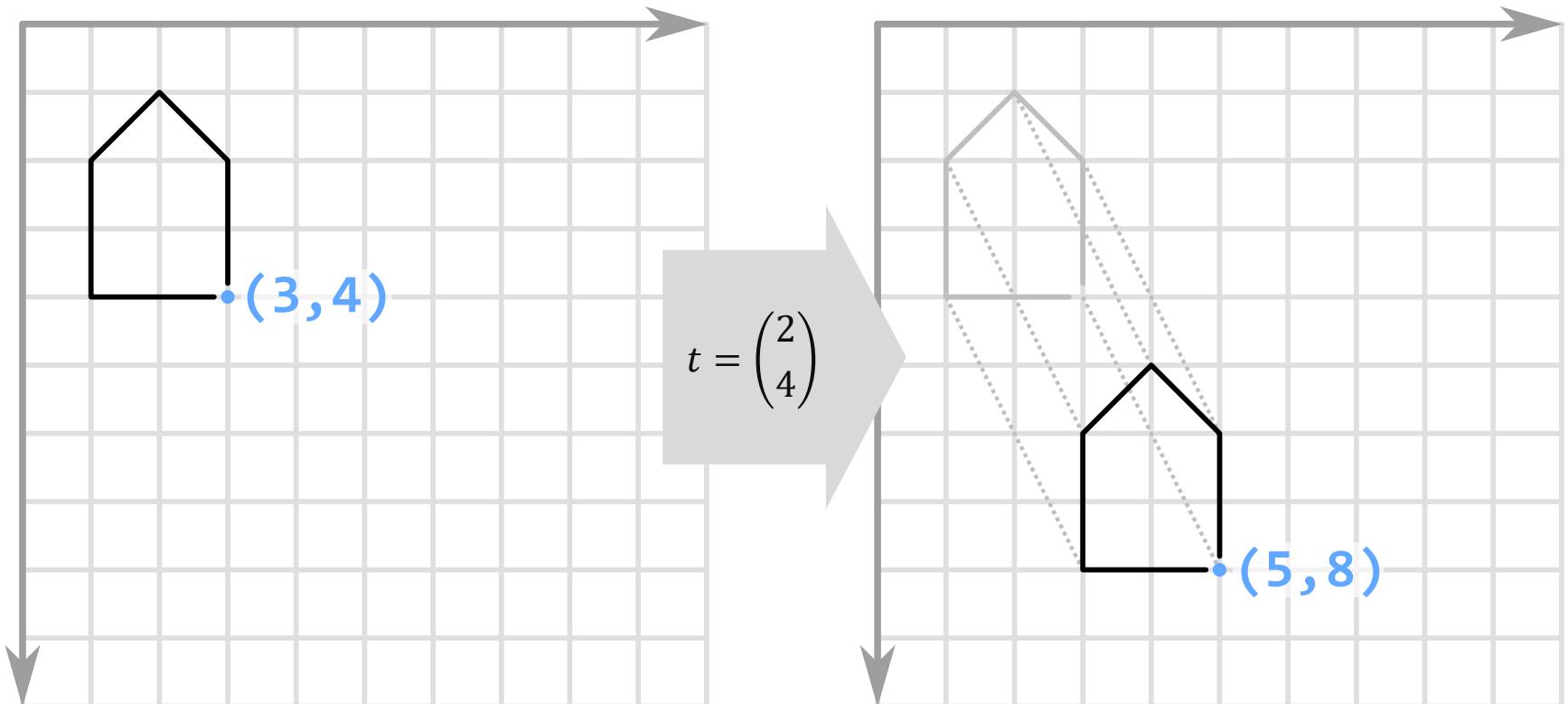
Shape model points are defined relative to a base coordinate system. The model is transformed to a location before rendering through translation, rotation, and scaling.



$$(x'_i, y'_i) = f(x_i, y_i)$$

# Translation

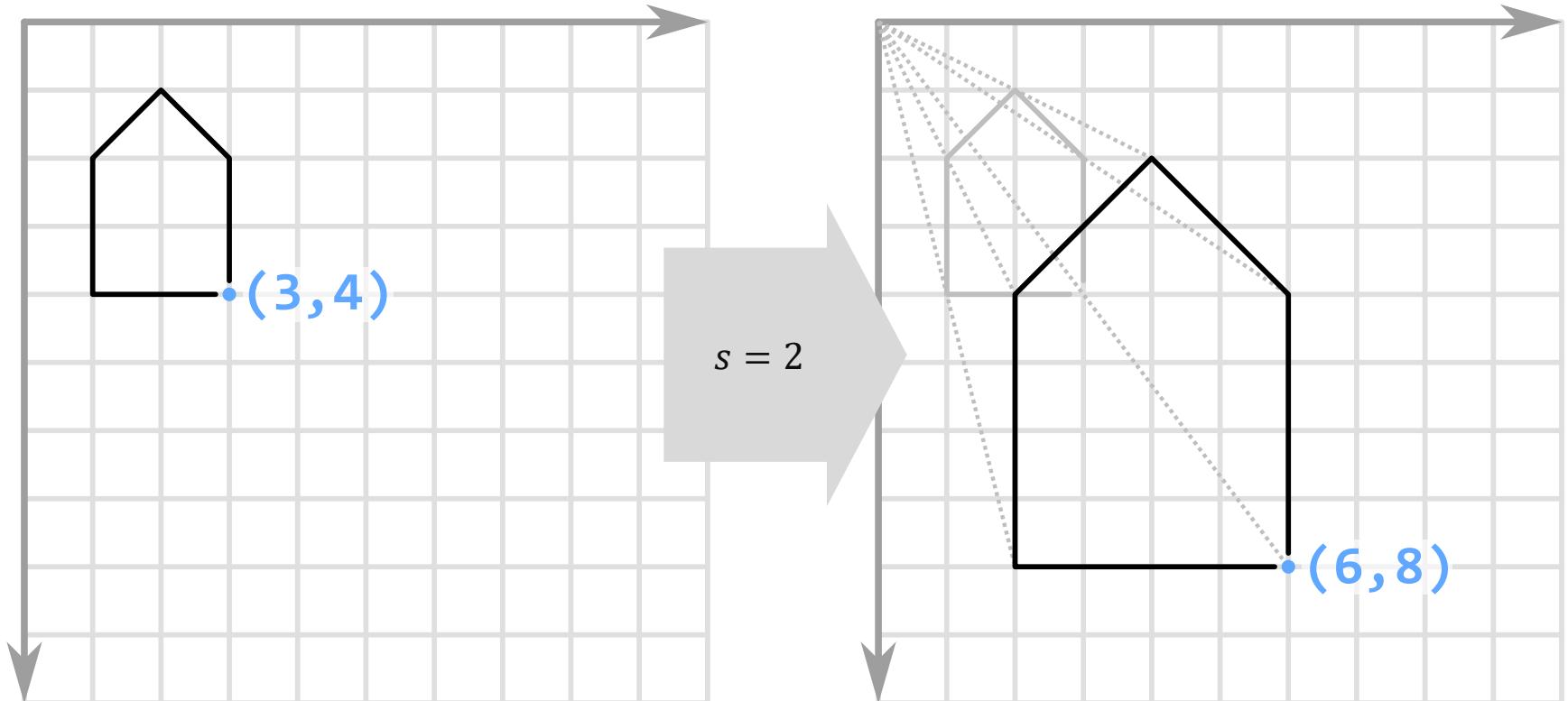
Translation adds the same vector  $t$  to each vertex  $v$ .



$$v' : \begin{cases} x' = x + t_x \\ y' = y + t_y \end{cases}$$

# Uniform Scaling

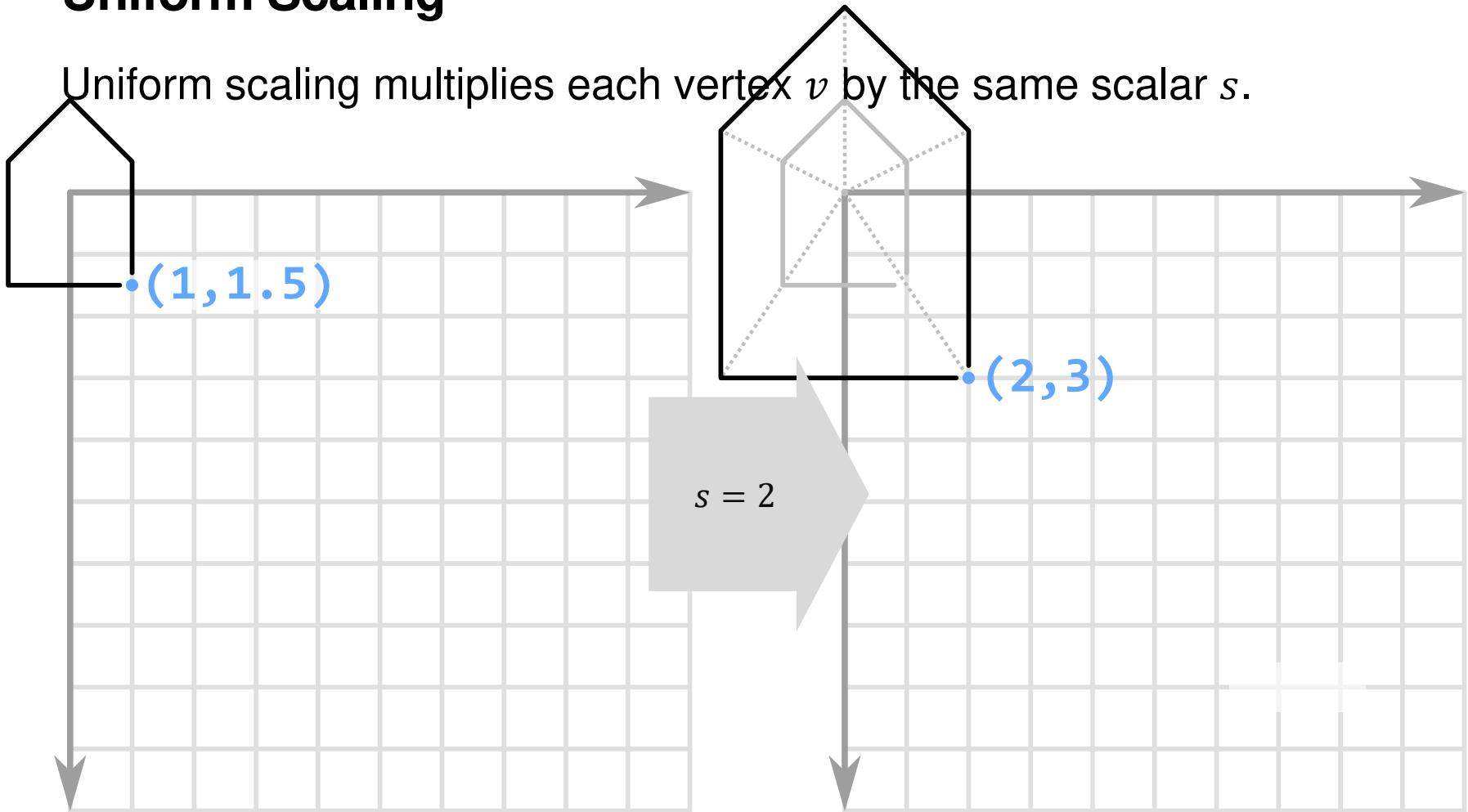
Uniform scaling multiplies each vertex  $v$  by the same scalar  $s$ .



$$v' : \begin{cases} x' = s \times x \\ y' = s \times y \end{cases}$$

# Uniform Scaling

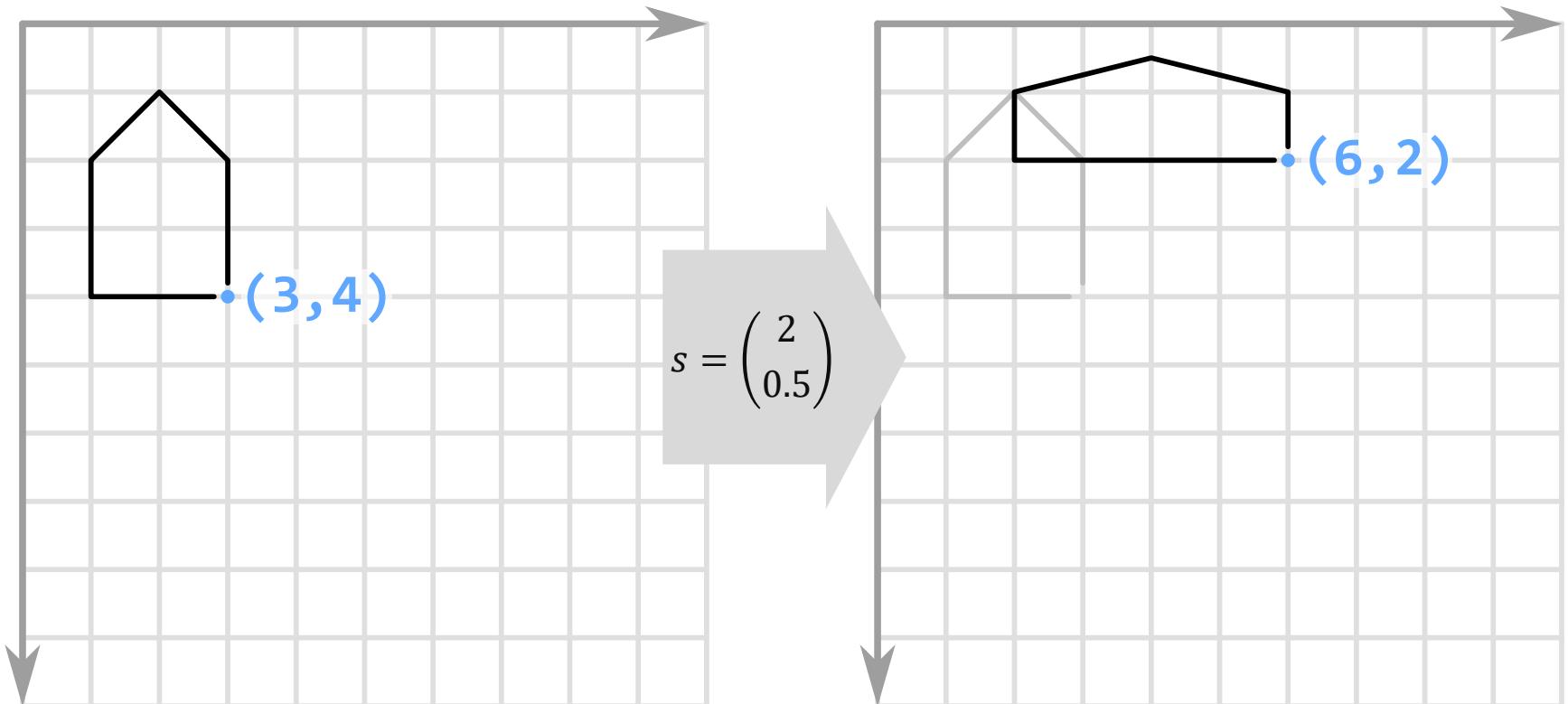
Uniform scaling multiplies each vertex  $v$  by the same scalar  $s$ .



$$v' : \begin{cases} x' = s \times x \\ y' = s \times y \end{cases}$$

# Non-Uniform Scaling

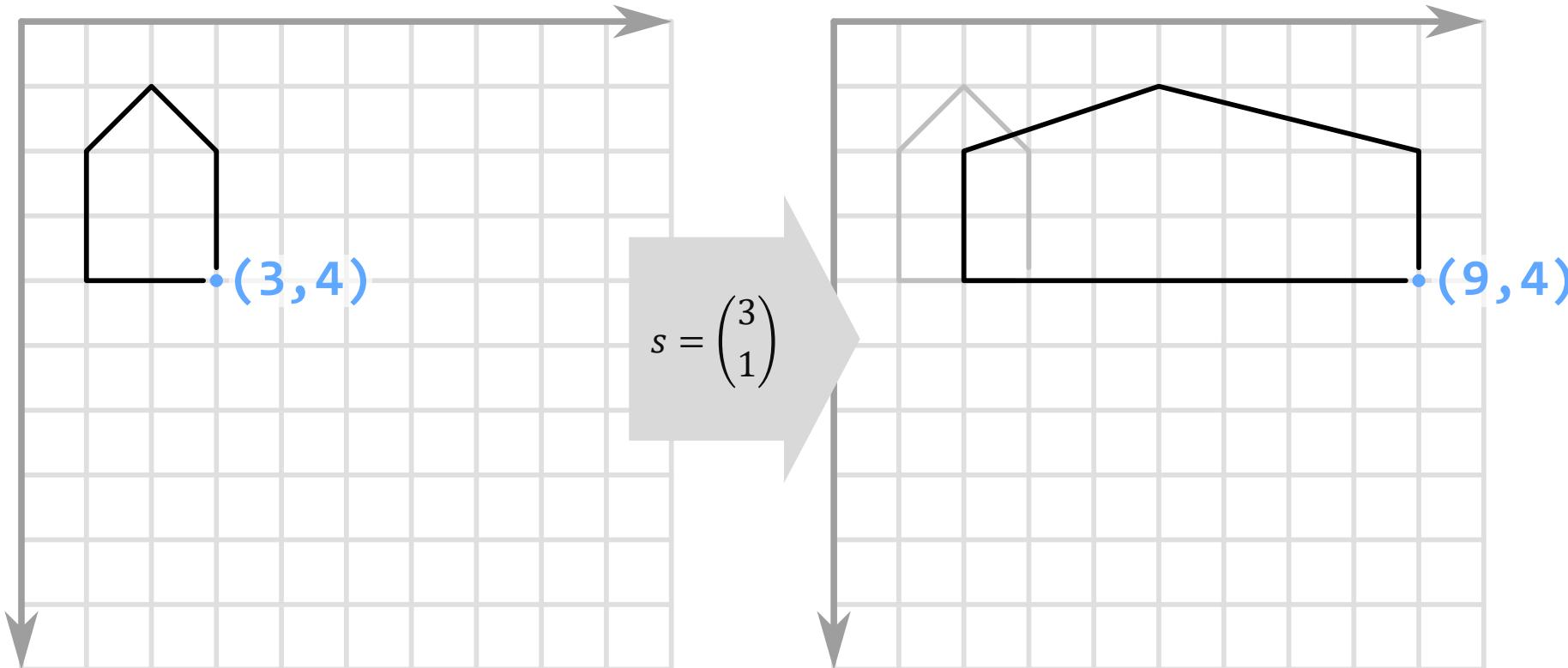
Non-uniform scaling applies the same scalar  $s$  to each vertex  $v$ .



$$v' : \begin{cases} x' = s_x \times x \\ y' = s_y \times y \end{cases}$$

# Non-Uniform Scaling

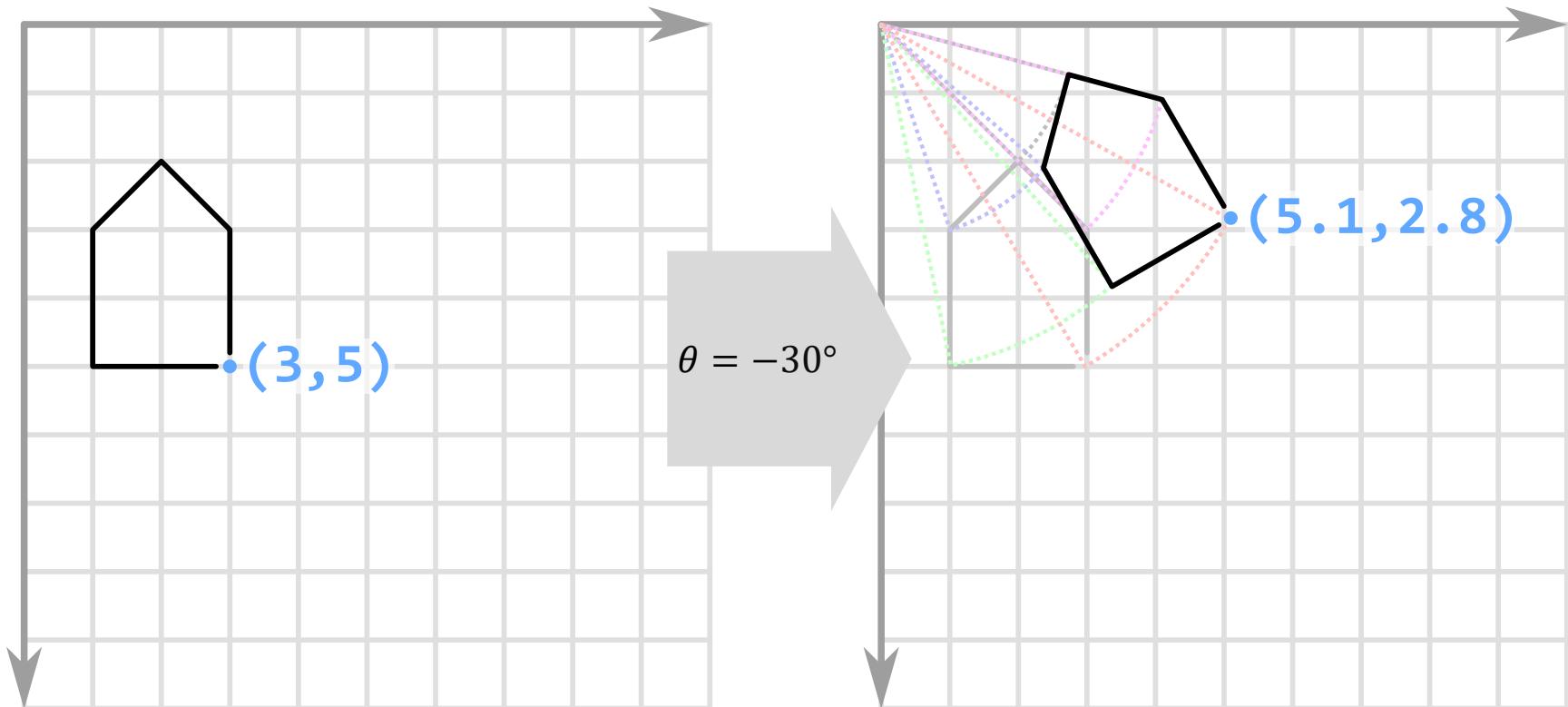
Non-uniform scaling applies the same scalar  $s$  to each vertex  $v$ .



$$v' : \begin{cases} x' = s_x \times x \\ y' = s_y \times y \end{cases}$$

# Rotation

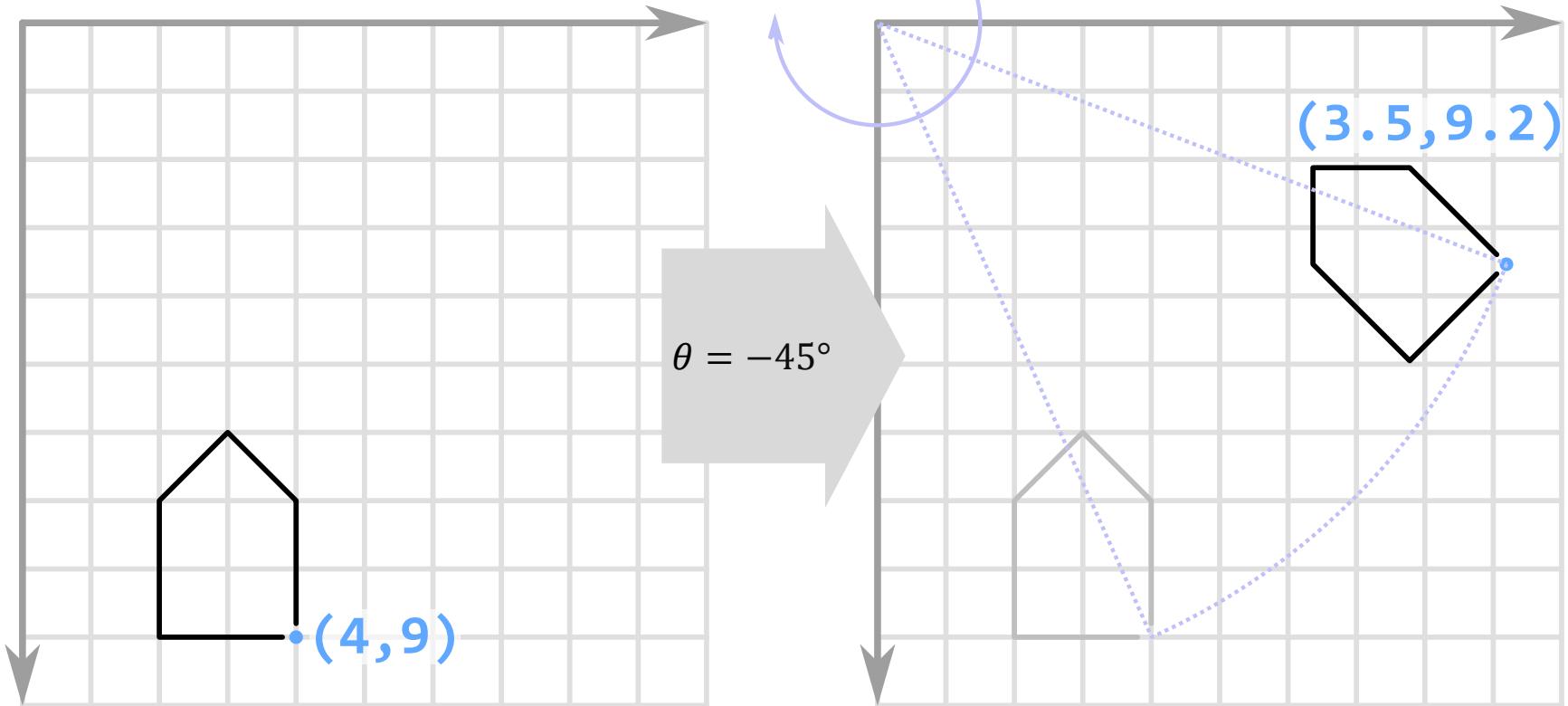
Rotation rotates each vertex  $v$  around the origin of the base coordinate system by  $\theta$  degrees.



$$v' : \begin{cases} x' = x \cos \theta - y \sin \theta \\ y' = x \sin \theta + y \cos \theta \end{cases}$$

# Rotation

Rotation direction, i.e., clockwise (CW) or counter-clockwise (CCW), depends on the “handedness” of the coordinate system (LHS or RHS)

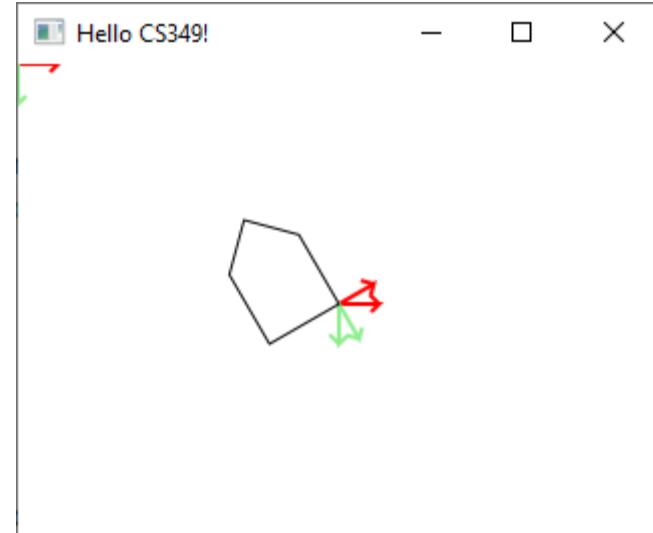


$$v' : \begin{cases} x' = x \cos \theta - y \sin \theta \\ y' = x \sin \theta + y \cos \theta \end{cases}$$

# Transforms: Using the Graphics Context (GC)

We can use the Graphics Context to draw and transform. This has the effect of setting up common transformations that all shapes will use. Transformations are relative to the base coordinate system.

```
val canvas = Canvas(640.0, 480.0)
canvas.graphicsContext2D.apply {
    drawCoords(this)
    translate(160.0, 120.0)
    drawCoords(this)
    rotate(-30.0)
    drawCoords(this)
    strokePolygon(
        listOf(0.0, 0.0, -20.0, -40.0, -40.0).toDoubleArray(),
        listOf(0.0, -40.0, -60.0, -40.0, 0.0).toDoubleArray(), 5)
}
val scene = Scene(Group(canvas), 320.0, 240.0)
```

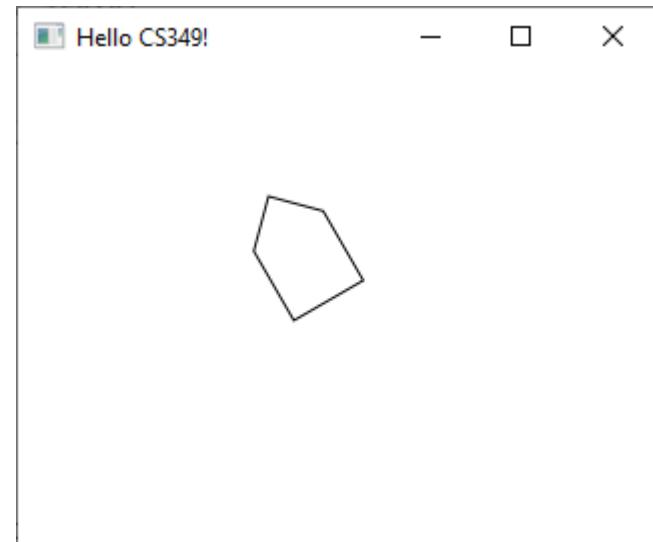


# Transforms: Using Shape Properties

We can also use the built-in functions in a Shape class.

Transformations are relative to the centre of the shape's bounding box.

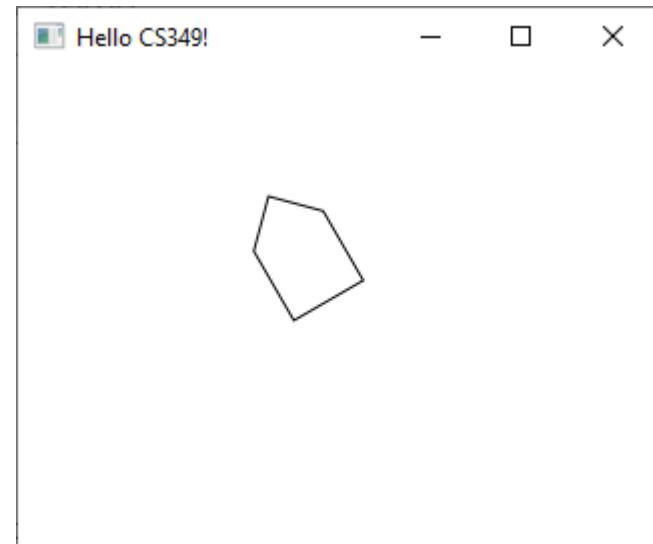
```
val house = Polygon(0.0, 0.0, 0.0, -40.0, -20.0, -60.0,  
                    -40.0, -40.0, -40.0, 0.0).apply {  
    translateX = 160.0  
    translateY = 120.0  
    rotate = -30.0  
    fill = Color.TRANSPARENT  
    stroke = Color.BLACK  
}  
  
val scene = Scene(Group(house),  
                  320.0, 240.0)
```



# Transforms: Using Containers

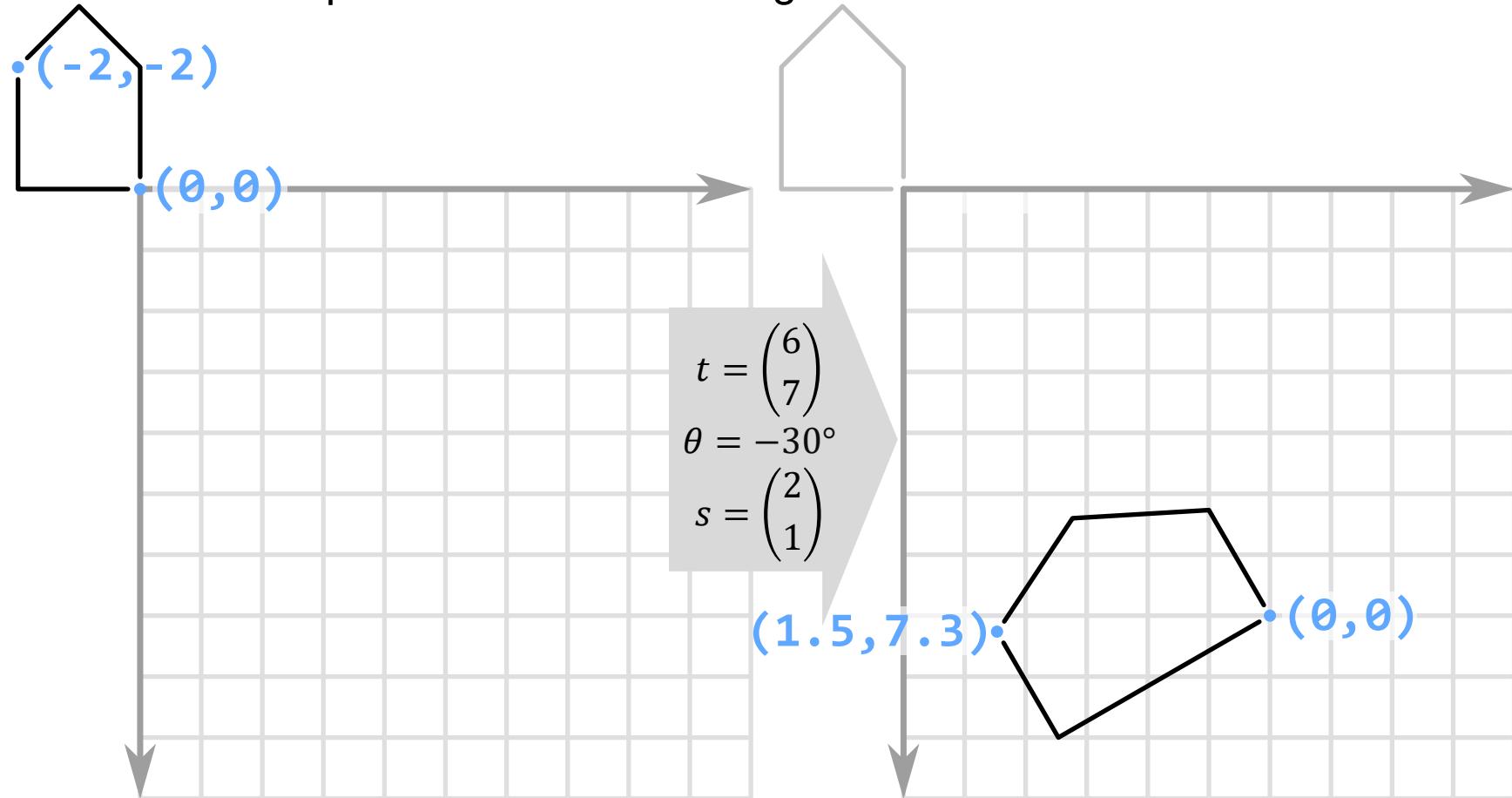
Transformations can also be applied to a container. They will be applied to all its children. Transformations are relative to the centre of the shape's bounding box.

```
val house = Polygon(0.0, 0.0, 0.0, -40.0, -20.0, -60.0,  
                    -40.0, -40.0, -40.0, 0.0).apply {  
    fill = Color.TRANSPARENT  
    stroke = Color.BLACK  
}  
  
val houseGroup = Group(house).apply {  
    rotate = -30.0  
    translateX = 160.0  
    translateY = 120.0  
}  
  
val scene = Scene(houseGroup,  
                  320.0, 240.0)
```



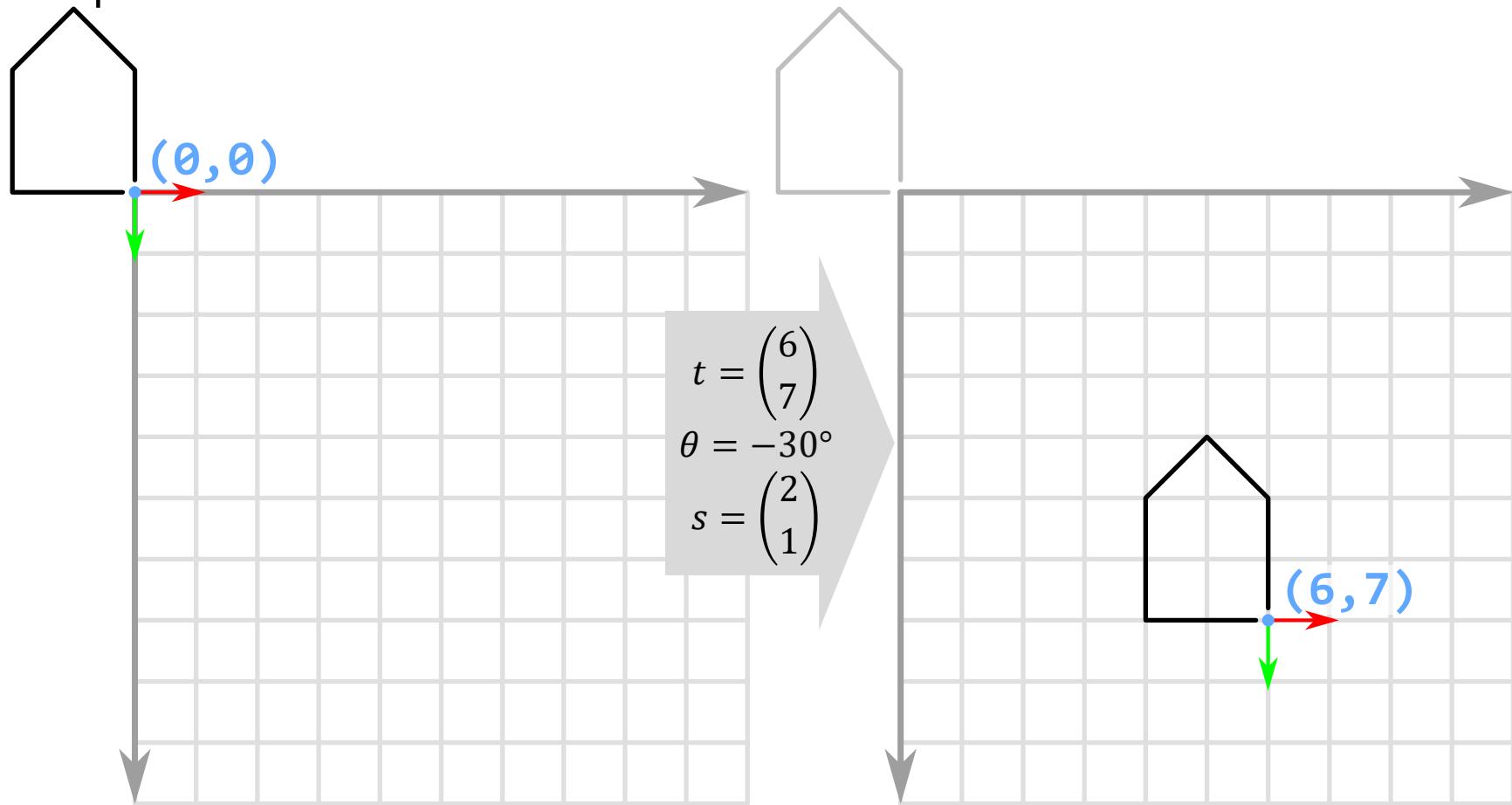
# Combining Transformations

Combine multiple transformations together:



# Combining Transformations

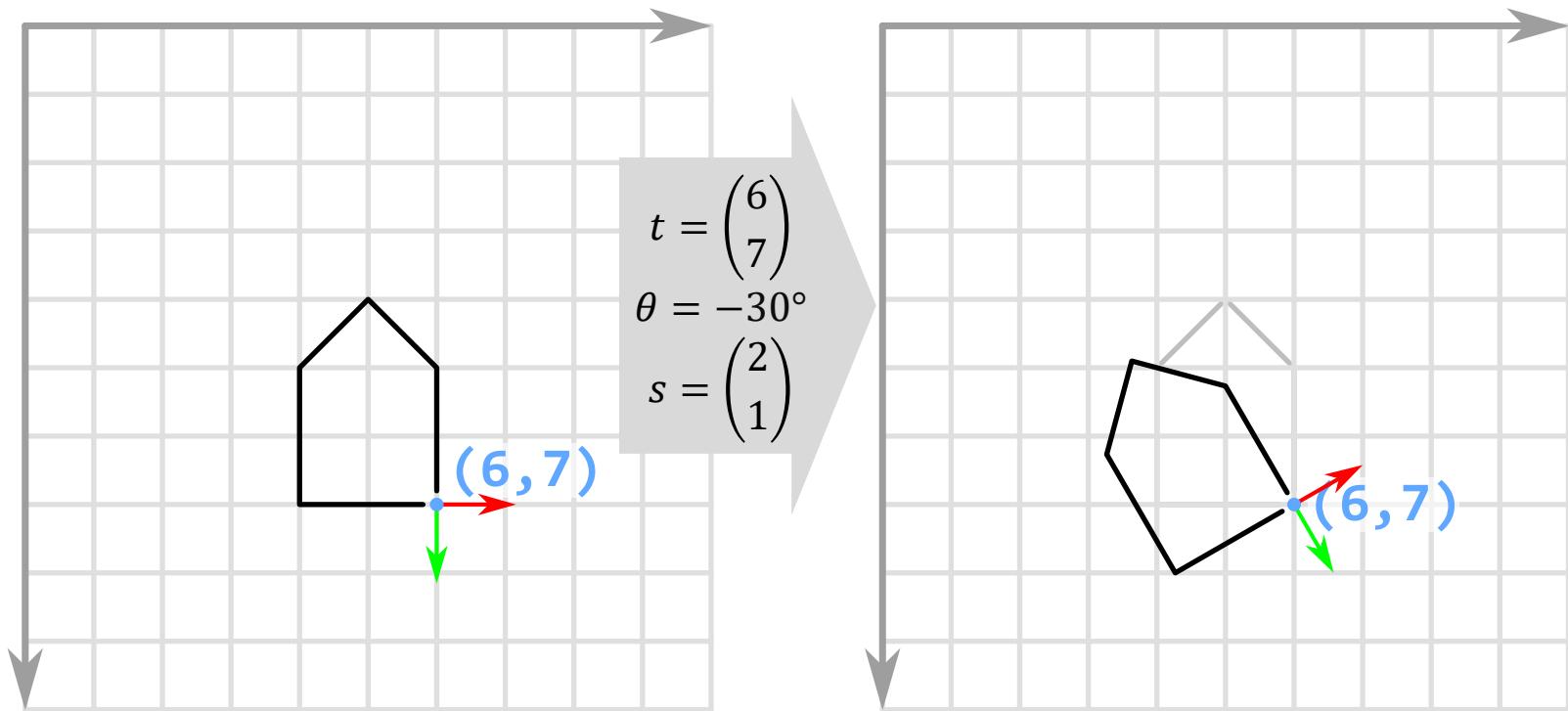
Step 1: Translate



$$v' : \begin{cases} x' = x + 6 \\ y' = y + 7 \end{cases}$$

# Combining Transformations

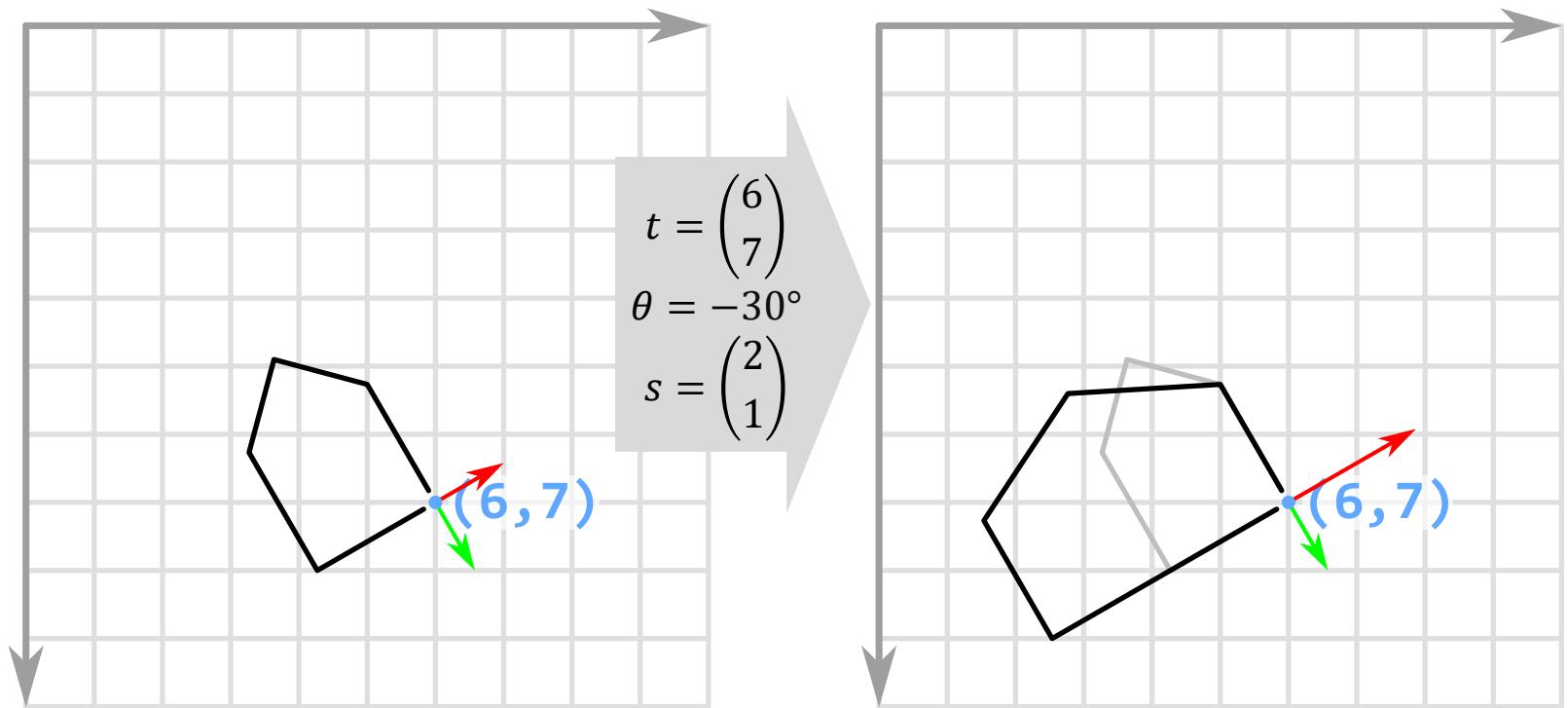
Step 2: Rotate



$$v'' : \begin{cases} x'' = x \cos(-30) - y \sin(-30) + 6 \\ y'' = x \sin(-30) + y \cos(-30) + 7 \end{cases}$$

# Combining Transformations

Step 3: Scale

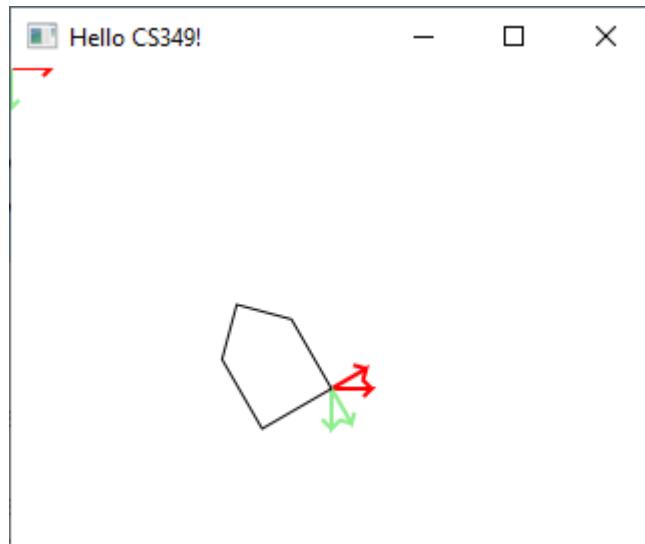


$$v''' : \begin{cases} x''' = (2x) \cos(-30) - (1y) \sin(-30) + 6 \\ y''' = (2x) \sin(-30) + (1y) \cos(-30) + 7 \end{cases}$$

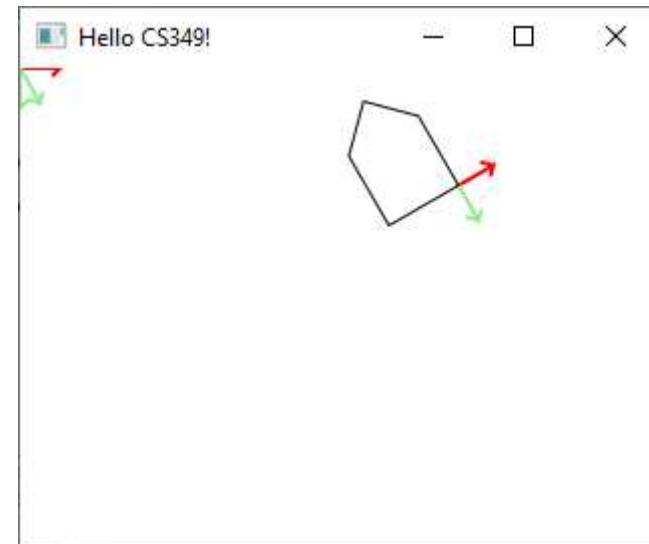
# Combining Transformations: GC Order

The order in which transformations are applied can matter!

```
translate(160.0, 160.0)
rotate(-30.0)
strokePolygon(
    listOf(...).toDoubleArray(),
    listOf(...).toDoubleArray(),
    5)
```

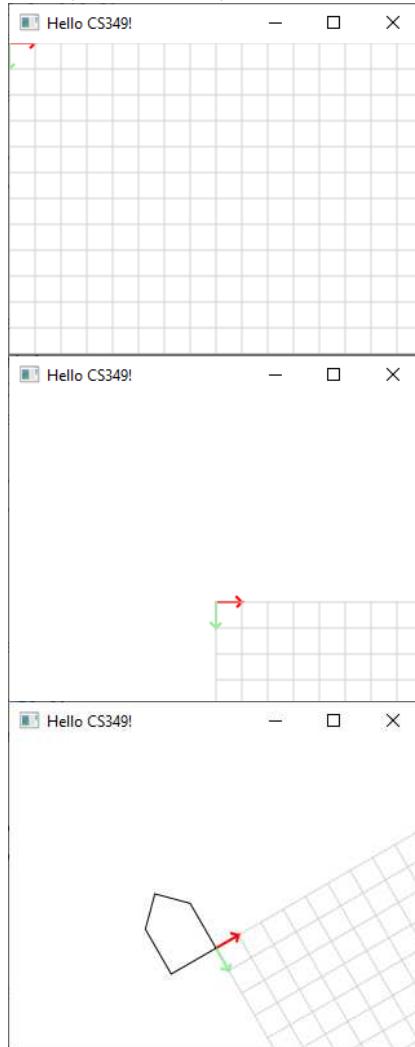


```
rotate(-30.0)
translate(160.0, 160.0)
strokePolygon(
    listOf(...).toDoubleArray(),
    listOf(...).toDoubleArray(),
    5)
```

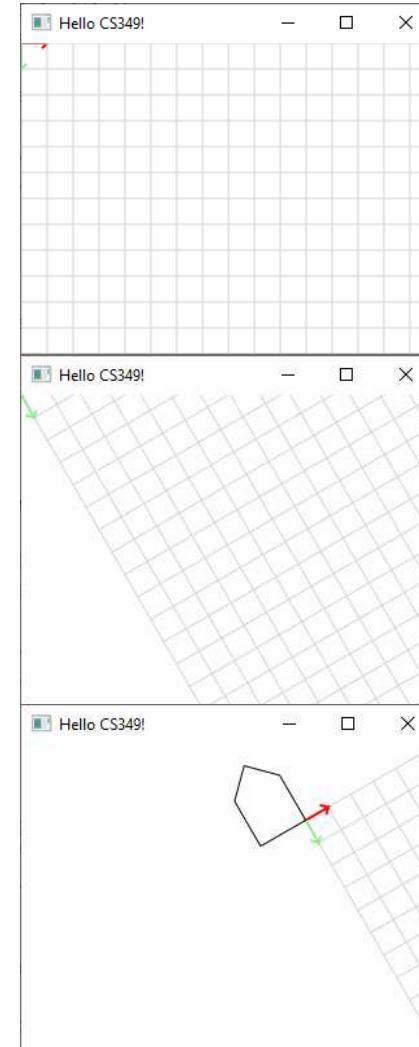


# Combining Transformations: GC Order

```
translate(160.0, 160.0)  
rotate(-30.0)
```



```
rotate(-30.0)  
translate(160.0, 160.0)
```



# Combining Transformations: Shape Order

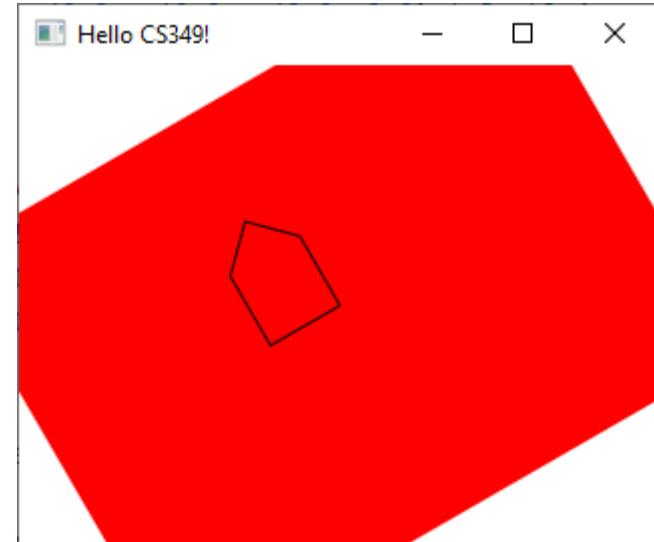
```
val house = Polygon(...).apply {  
    fill = Color.TRANSPARENT  
    stroke = Color.BLACK  
    rotate = -30.0  
}
```

```
val hgroup = Group(house).apply {  
    translateX = 160.0  
    translateY = 120.0  
}
```



```
val house = Polygon(...).apply {  
    fill = Color.TRANSPARENT  
    stroke = Color.BLACK  
    translateX = 160.0  
    translateY = 120.0  
}
```

```
val hgroup = Group(house).apply {  
    rotate = -30.0  
}
```



# Combining Transformations

We could create transformation equations, but

$$v''' : \begin{cases} x''' = (2x) \cos(-30) - (1y) \sin(-30) + 6 \\ y''' = (2x) \sin(-30) + (1y) \cos(-30) + 7 \end{cases}$$

is hard to express and implement in code.

Matrices are a better way to express multiple transformations:

- always a fixed size regardless of number of transformations
- easy to transform many vertices in code (i.e., matrix multiplication)
- easy to parallelize code (i.e., can perform calculations on GPU)

# Goal: Matrix Representation

Represent a 2D transformation as a matrix:  $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$

Represent each vertex as a  $2 \times 1$  column vector:  $\begin{pmatrix} x \\ y \end{pmatrix}$

Multiply matrixes to apply the transformation and get new vertex:

$$\begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned} \Leftrightarrow \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

# Matrix Representation

Transformations can be combined by matrix multiplication

- Transformations are associative: we can multiply them together

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} \begin{pmatrix} i & j \\ k & l \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

This creates a single **transformation matrix**

- represents *the result of multiple individual transformations*
- can be easily and quickly applied to vertices in multiple Shapes
- can be loaded into the GPU for performance

# Matrix Representations – Naïve Approach

Scale:

$$\begin{aligned}x' &= x + s_x \\y' &= y + s_y\end{aligned}\Leftrightarrow \begin{pmatrix}x' \\ y'\end{pmatrix} = \begin{pmatrix}s_x & 0 \\ 0 & s_y\end{pmatrix} \begin{pmatrix}x \\ y\end{pmatrix}$$

Rotation :

$$\begin{aligned}x' &= x \cos(\theta) - y \sin(\theta) \\y' &= x \sin(\theta) + y \cos(\theta)\end{aligned}\Leftrightarrow \begin{pmatrix}x' \\ y'\end{pmatrix} = \begin{pmatrix}\cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta)\end{pmatrix} \begin{pmatrix}x \\ y\end{pmatrix}$$

Translation:

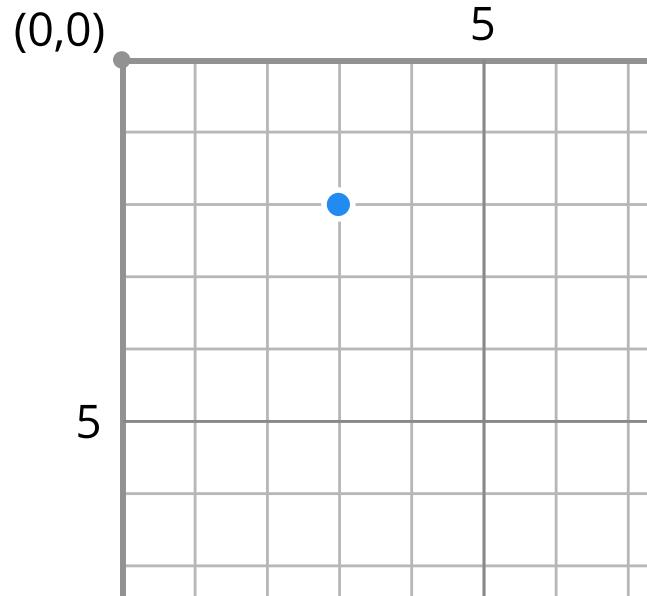
$$\begin{aligned}x' &= x + t_x \\y' &= y + t_y\end{aligned}\Leftrightarrow \begin{pmatrix}x' \\ y'\end{pmatrix} = \begin{pmatrix}1 & \frac{t_x}{y} \\ \frac{t_y}{x} & 1\end{pmatrix} \begin{pmatrix}x \\ y\end{pmatrix}$$

# Matrix Representation – Using Homogeneous Coordinates

Adding a 3<sup>rd</sup> component  $w$  to coordinates: if  $w$  is 0, the coordinates represent a vector, otherwise it is a vertex.

Dividing  $x$  and  $y$  by  $w$  yields the corresponding Cartesian point:

$[x, y, w]$  represents a point at Cartesian location  $[x/w, y/w]$



$$\begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 7.5 \\ 5.0 \\ 2.5 \end{bmatrix} \dots$$

Cartesian coordinates      homogeneous coordinates

The diagram shows three equivalent homogeneous coordinate representations of the same point (3, 2) in Cartesian coordinates. The first row is the original 2D Cartesian coordinates. The second row shows the addition of a third component w=1, resulting in a 3D homogeneous coordinate. The third row shows another valid homogeneous coordinate where both x and y are doubled. Ellipses indicate that there are infinitely many such representations.

There are infinite homogeneous coordinates representing the same vertex in Cartesian coordinates.

# Homogeneous Matrix Representation

Each homogenous vertex is a  $3 \times 1$  column matrix with  $w = 1$ :

$$\begin{bmatrix} x \\ y \end{bmatrix} \Leftrightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

As a result,  $2 \times 2$  transformation matrixes will not work:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = ???$$

Instead, we use  $3 \times 3$  transformation matrixes:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax & by & c \\ dx & ey & f \\ gx & hy & i \end{bmatrix}$$

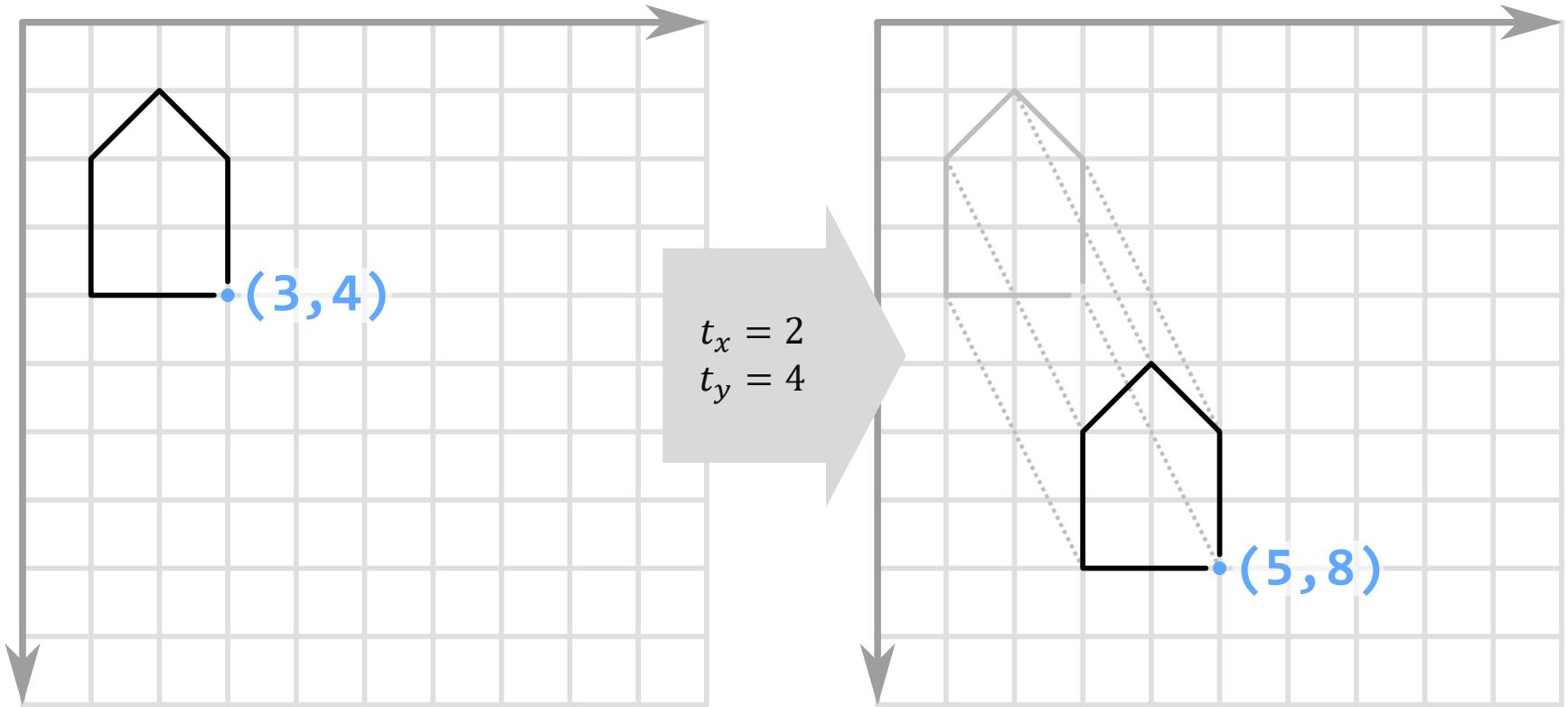
# Homogeneous Matrix Representation: Translation

Now we can represent 2D translation with a  $3 \times 3$  matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

# Homogeneous Matrix Representation: Translation

Translation adds the scalars  $t_x$ ,  $t_y$  to each vertex  $v$ .



$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 8 \\ 1 \end{bmatrix}$$

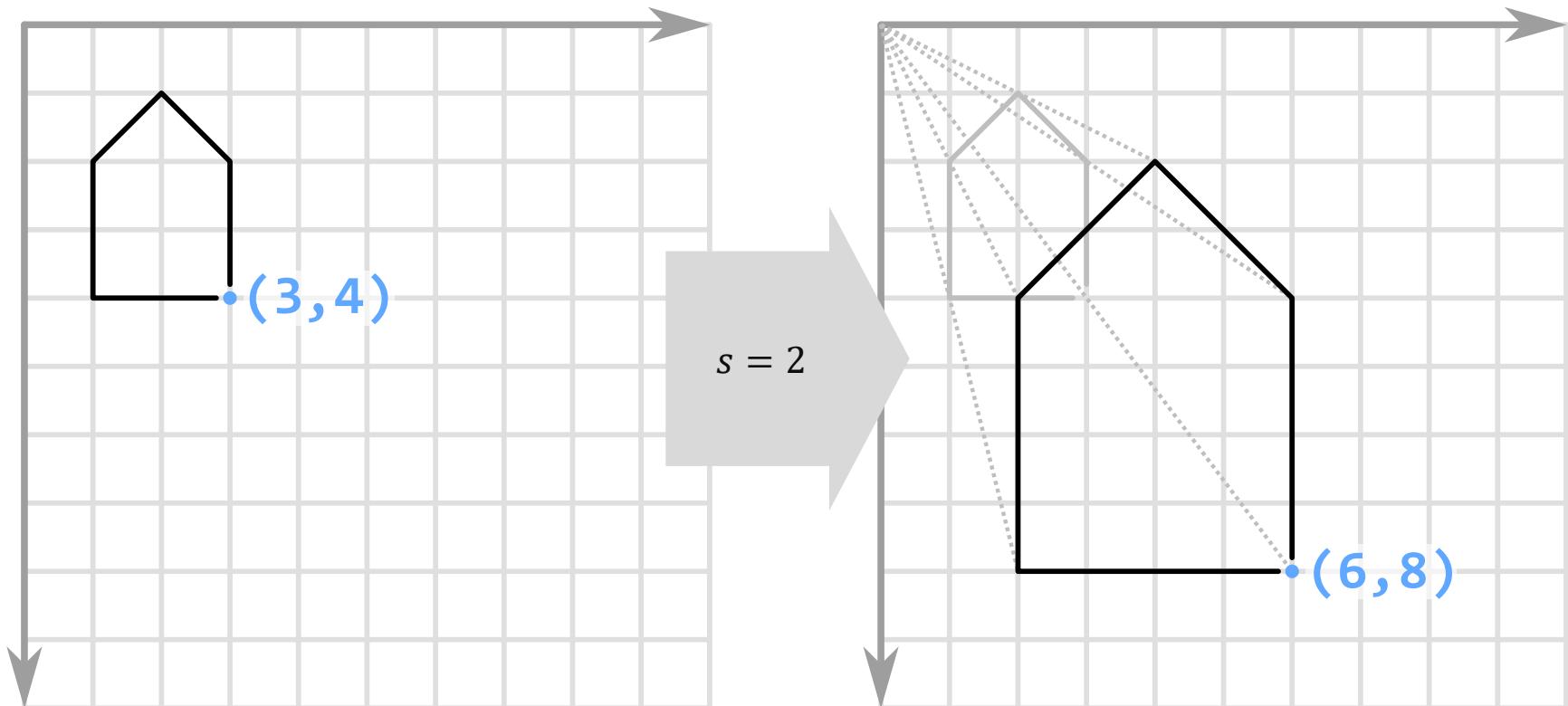
# Homogeneous Matrix Representation: Scale

Now we can represent 2D scale with a  $3 \times 3$  matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x \times s_x \\ y \times s_y \\ 1 \end{bmatrix}$$

# Homogeneous Matrix Representation: Scale

Multiply each point coordinate by the same scalar.



$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 8 \\ 1 \end{bmatrix}$$

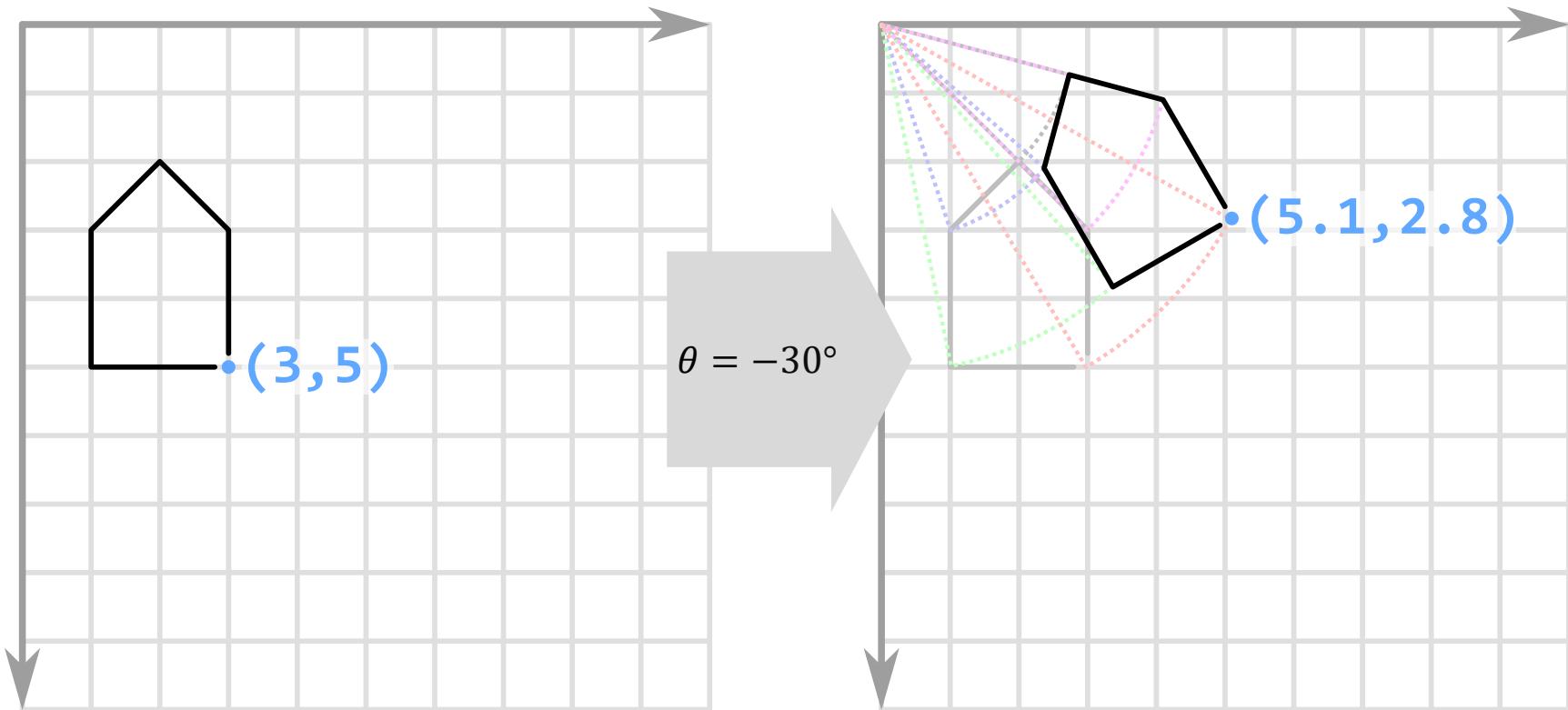
# Homogeneous Matrix Representation: Rotation

Now we can represent 2D rotation with a  $3 \times 3$  matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \\ 1 \end{bmatrix}$$

# Rotation

Each new point coordinate component is a function of an angle  $\theta$  and both of the previous point coordinates.



$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0.87 \\ 0.5 \\ 0 \end{bmatrix} \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 5.1 \\ 2.8 \\ 1 \end{bmatrix}$$

# Transformation Matrices

Translate

$$\mathbf{T}(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Scale

$$\mathbf{S}(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotate

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

# Affine Transformation Matrix: Concatenation

These 3x3 matrices are examples of an **Affine Transformation Matrix**. They can express any combination of translation, rotation, and scaling: Individual transformations can be combined using matrix multiplication, which is often called transformation concatenation.

The original vector  $v$  is first translated, then rotated, and finally scaled:

$$v' = T(t_x, t_y) \cdot R(\theta) \cdot S(s_x, s_y) \cdot v$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

# Affine Transformation Matrix: Concatenation

Matrix multiplication is not commutative: if  $A$  and  $B$  are matrices, “Not Commutative” means:

$$A \times B \neq B \times A$$

The order in which transformations are concatenated can change the result:

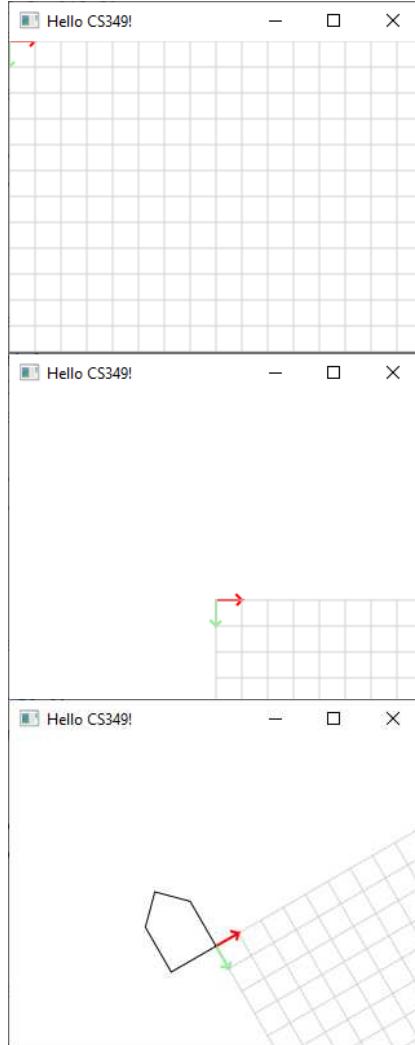
$$T(2,3) \cdot S(4,5) = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 2 \\ 0 & 5 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

whereas:

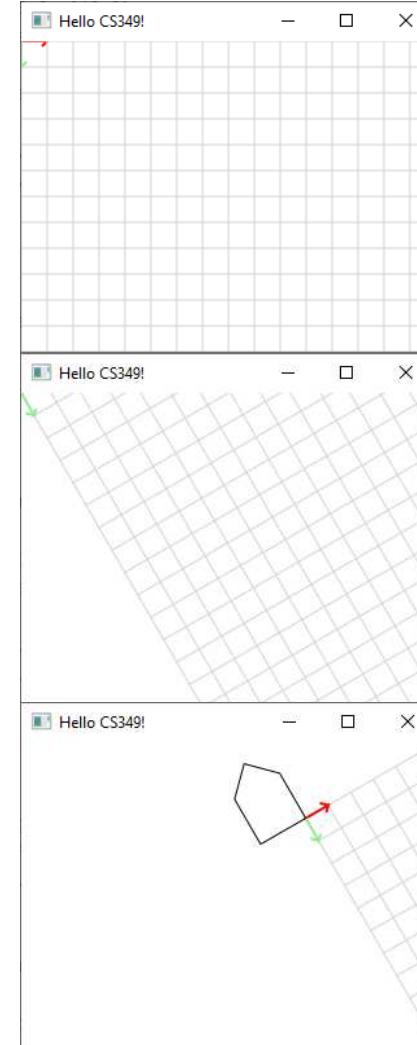
$$S(4,5) \cdot T(2,3) = \begin{bmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 0 & 8 \\ 0 & 5 & 15 \\ 0 & 0 & 1 \end{bmatrix}$$

# Affine Transformation Matrix: Concatenation

```
translate(160.0, 160.0)  
rotate(-30.0)
```



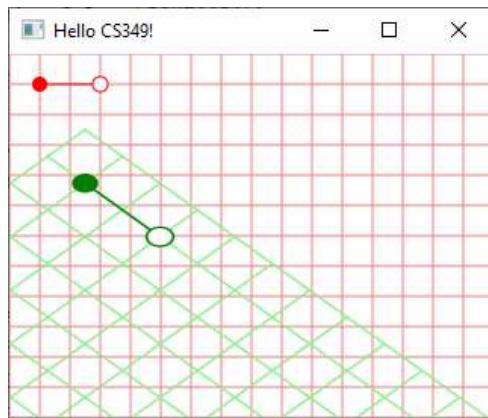
```
rotate(-30.0)  
translate(160.0, 160.0)
```



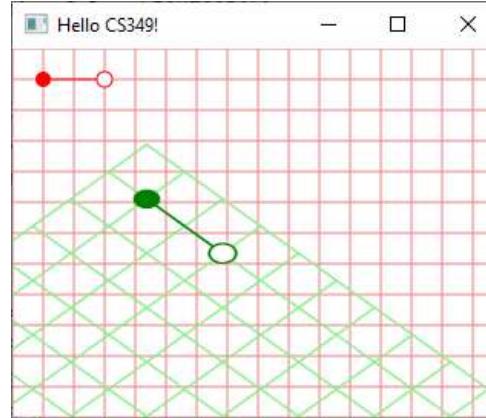
# Affine Transformation Matrix: Order Examples

[S]cale(1.75, 1.25); [R]otate(45.0); [T]ranslate(50.0, 50.0)

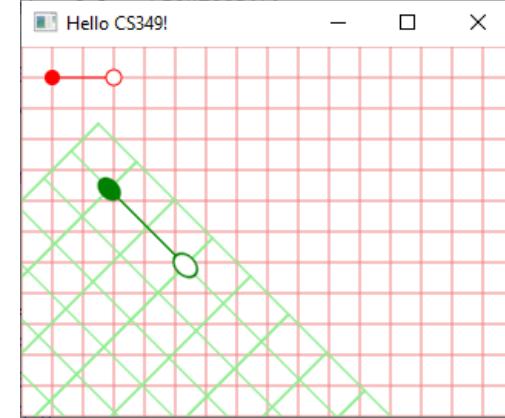
TSR



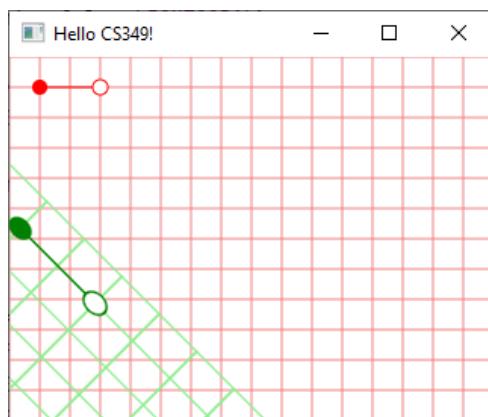
STR



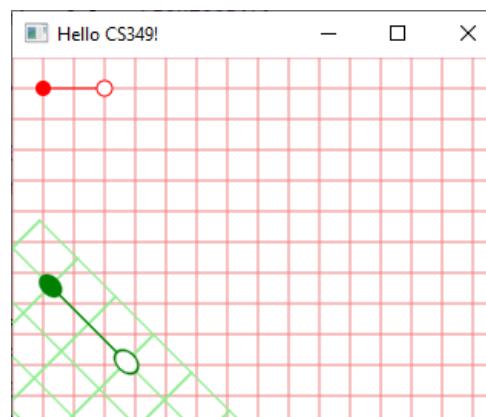
TRS



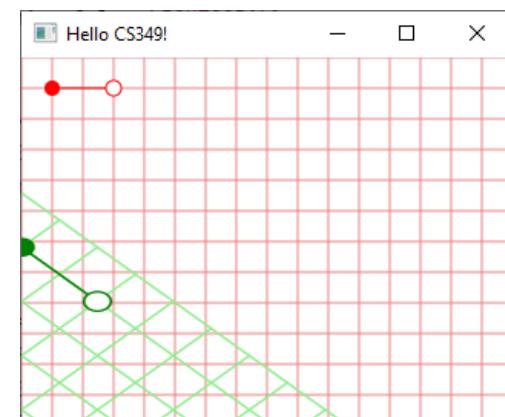
RTS



RST



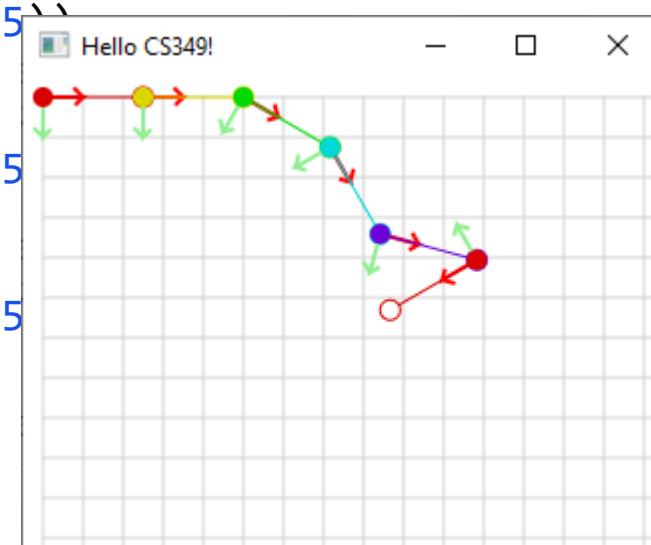
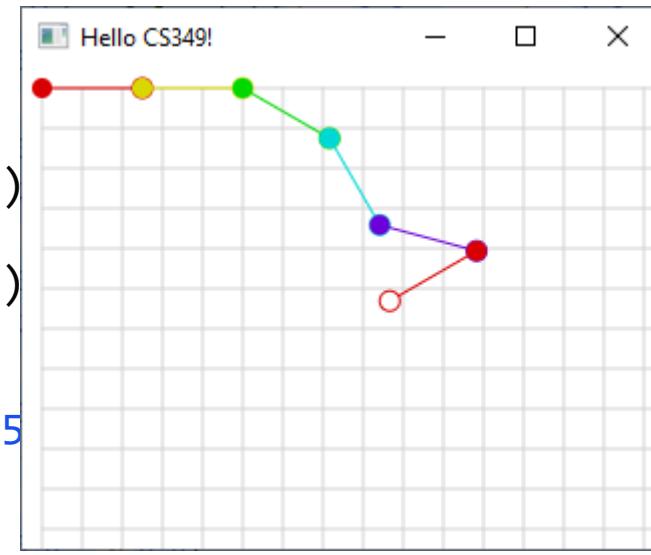
SRT



# Cumulative Transformations

It can be useful to draw shapes relative to the current transform:

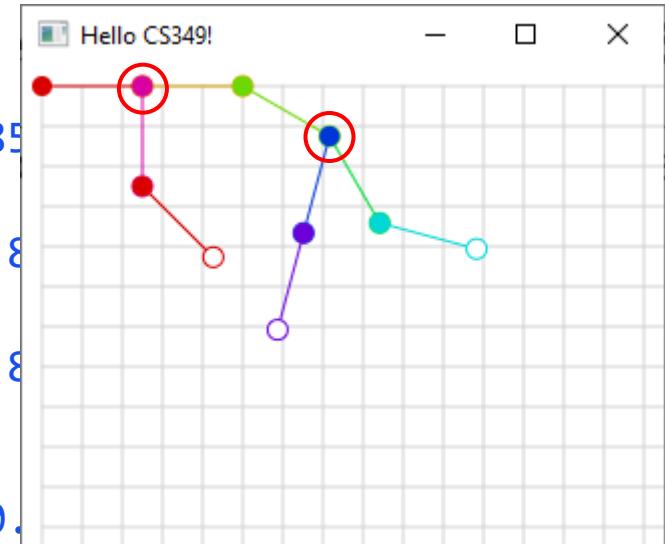
```
canvas.graphicsContext2D.apply {  
    translate(10.0, 10.0)  
    drawGrid(this, Color.LIGHTGRAY)  
    drawHandle(this, Color.hsb(0.0, 1.0, 0.85))  
    translate(50.0, 0.0)  
    drawHandle(this, Color.hsb(60.0, 1.0, 0.85))  
    translate(50.0, 0.0)  
    rotate(30.0)  
    drawHandle(this, Color.hsb(120.0, 1.0, 0.85)  
    translate(50.0, 0.0)  
    rotate(30.0)  
    drawHandle(this, Color.hsb(180.0, 1.0, 0.85)  
    translate(50.0, 0.0)  
    rotate(-45.0)  
    drawHandle(this, Color.hsb(270.0, 1.0, 0.85)  
    translate(50.0, 0.0)  
    rotate(135.0)  
    drawHandle(this, Color.hsb(360.0, 1.0, 0.85  
}
```



# Saving and Restoring Transformation State

It can be useful to save the graphics context at some point, then restore it again, e.g., building complex geometries:

```
canvas.graphicsContext2D.apply {
    drawHandle(this, Color.hsb(0.0, 1.0, 0.85))
    translate(50.0, 0.0)
    save()
    drawHandle(this, Color.hsb(45.0, 1.0, 0.85))
    translate(50.0, 0.0)
    rotate(30.0)
    drawHandle(this, Color.hsb(90.0, 1.0, 0.85))
    translate(50.0, 0.0)
    save()
    rotate(30.0)
    drawHandle(this, Color.hsb(135.0, 1.0, 0.85))
    translate(50.0, 0.0)
    rotate(-45.0)
    drawHandle(this, Color.hsb(180.0, 1.0, 0.85))
    restore()
    rotate(75.0)
    drawHandle(this, Color.hsb(225.0, 1.0, 0.85))
    translate(50.0, 0.0)
    drawHandle(this, Color.hsb(270.0, 1.0, 0.85))
    restore()
    rotate(90.0)
    drawHandle(this, Color.hsb(315.0, 1.0, 0.85))
    translate(50.0, 0.0)
    rotate(-45.0)
    drawHandle(this, Color.hsb(360.0, 1.0, 0.85))
}
```



# Saving and Restoring Transformation State

It can be useful to save the graphics context at some point, then restore it again, e.g., for additional drawings:

```
fun drawHandle(gc: GraphicsContext, col: Color) {  
    gc.save()  
    gc.stroke = col  
    gc.fill = col  
    gc.strokeLine(0.0, 0.0, 50.0, 0.0)  
    gc.fillOval(-5.0, -5.0, 10.0, 10.0)  
    gc.fill = Color.WHITE  
    gc.fillOval(45.0, -5.0, 10.0, 10.0)  
    gc.strokeOval(45.0, -5.0, 10.0, 10.0)  
    gc.restore()  
}
```

# End of the Chapter



- Graphics pipeline
- Transformations
  - Which one exists
  - How does the math work (conceptually)
- Affine Transformation Matrices
  - Why we use them



Any further questions?

-



X

# Hit Testing

U

CS 349

February 27

# Implementing GUI Direct Manipulation

Graphical elements are directly manipulated using a pointing device

- includes graphical content, widgets, etc.

Key requirement is to detect *what* the mouse cursor is pointing at

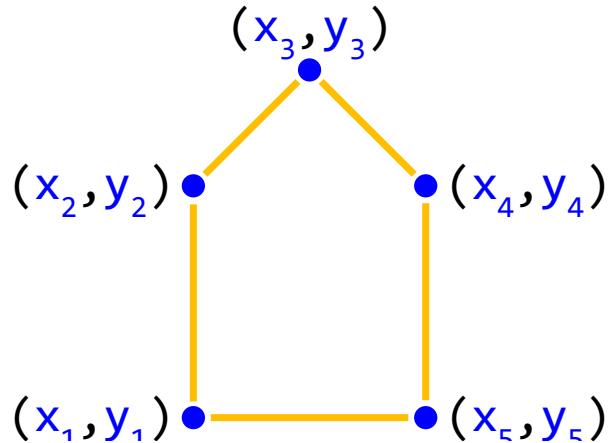
- all graphical content shape can be described as a “shape”
- shape could be filled, outlined, or special case like text
- need to consider reasonable tolerances for usability  
(consider near misses as hits for small / narrow shapes)

Today we walk through how to do this from the ground up

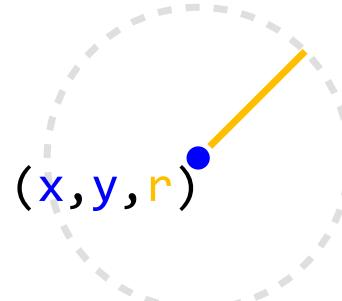
- a general **model** to describe shapes
- **hit-tests** to detect when a cursor is inside shape or on its edge

# Shape Model Geometry

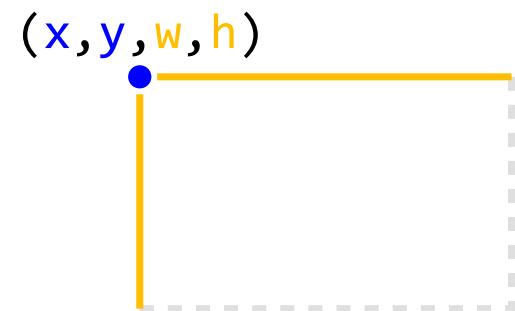
Different shapes have different *geometric representations*:



**Polygon**  
list of points



**Circle**  
center, radius



**Rectangle**  
top-left corner,  
width,height

- Many alternate geometric representations possible
- Many other kinds of shapes: Line, Polyline, Ellipse, ...
- Shape models can even be combinations of (different) shapes

# Simple Shape Model Class

- geometry that defines the Shape
- geometry properties (isFilled, isStroked)
- visual style properties (fill, stroke, strokeWeight)
- method to draw into a provided graphics context (i.e. render)
- method to do hit-testing with an x-y cursor position - new

# Shape Model Implementation

Define a Shape base class:

```
abstract class Drawable(var x: Double, var y: Double,  
                      var col: Color) {  
  
    abstract fun draw(gc: GraphicsContext)  
    abstract fun isHit(x: Double, y: Double): Boolean  
  
    override fun toString(): String {  
        return col.getName()  
    }  
}  
  
// Extension function for Double  
fun Double.between(low: Double, high: Double): Boolean {  
    return this in low .. high  
}
```

# Rectangle Shape Model Implementation

```
class FillRect(x: Double, y: Double,
               var w: Double, var h: Double,
               col: Color):
    Drawable(x, y, col) {

    override fun draw(gc: GraphicsContext) {
        gc.apply {
            save()
            fill = col
            fillRect(x, y, w, h)
            restore()
        }
    }

    override fun isHit(x: Double, y: Double): Boolean {
        // ...
    }
}
```

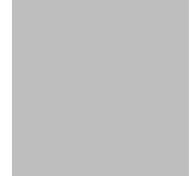
# Circle Shape Model Implementation

```
class FillCirc(x: Double, y: Double,  
                var d: Double,  
                col: Color):  
    Drawable(x, y, col) {  
  
    override fun draw(gc: GraphicsContext) {  
        gc.apply {  
            save()  
            fill = col  
            filloval(x, y, d, d)  
            restore()  
        }  
    }  
  
    override fun isHit(x: Double, y: Double): Boolean {  
        // ...  
    }  
}
```

# Hit-Test Paradigms

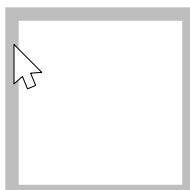
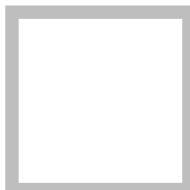
Inside Hit-Test: is mouse cursor inside shape?

Applies to closed and filled shapes like ovals, rectangles, and polygons.



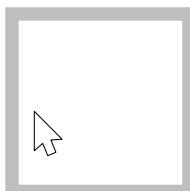
Edge Hit-Test: is mouse cursor on shape outline?

Applies to open and “non-filled” shapes like strokes, lines, and polylines.



A hit-test is tailored to the shape type and properties

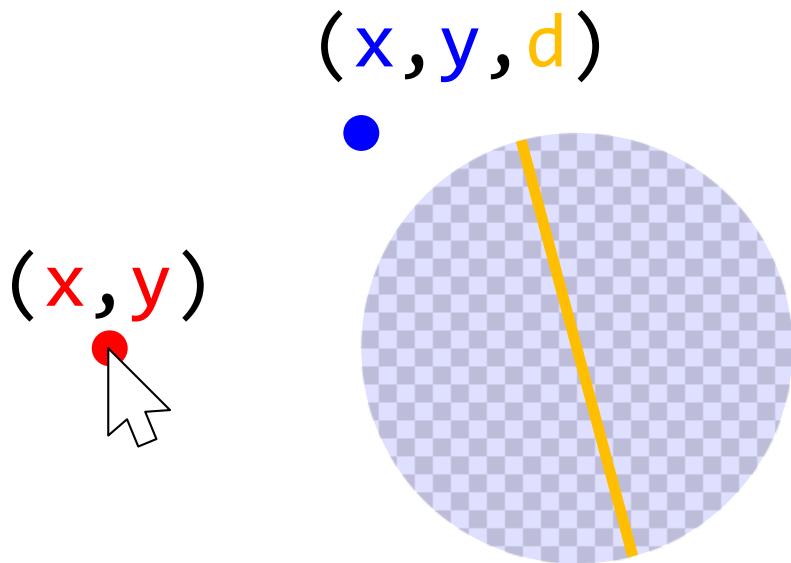
- if no fill, hit-test should be on shape outline only
- hit-test should factor in thickness of stroke



# Filled Circle Hit-Test

Given:

- Mouse position  $(x, y)$
- Upper-left bound of circle  $(x, y)$
- Diameter  $d$



Hit:

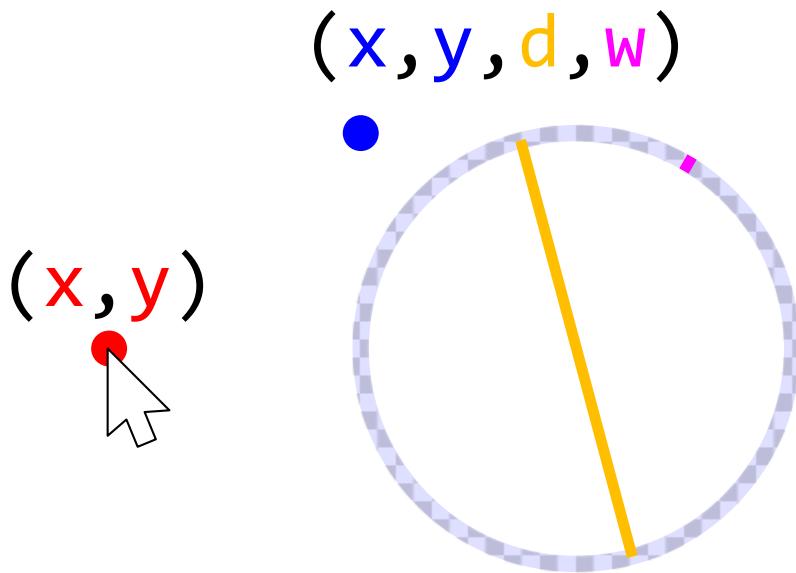
```
if distance
    from (x,y)
    to (x + d/2, y + d/2)
<= d/2
```

```
override fun isHit(x: Double, y: Double): Boolean {
    return sqrt(sqr(x - this.x - d / 2.0) +
               sqr(y - this.y - d / 2.0)) <= d / 2.0
}
```

# Stroke Circle Hit-Test

Given:

- Mouse position  $(x, y)$
- Upper-left bound of circle  $(x, y)$
- Diameter  $d$
- Stroke width  $w$



Hit:

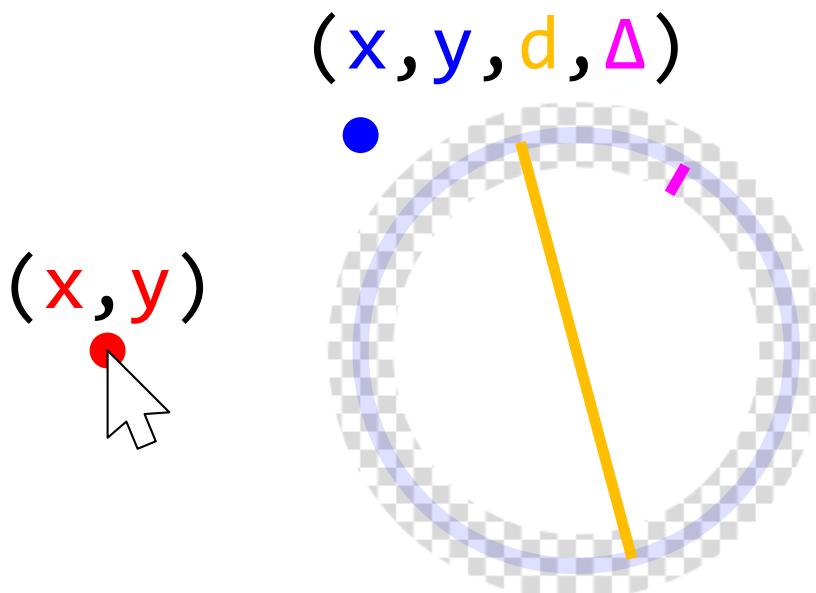
```
if distance
    from (x, y)
    to (x + d/2, y + d/2)
    between d/2 + w/2 and d/2 - w/2
```

```
override fun isHit(x: Double, y: Double): Boolean {
    return sqrt(sqrt(x - this.x - d / 2.0) +
        sqrt(y - this.y - d / 2.0)).between(d/2.0 + 0.5,
                                            d/2.0 - 0.5)
}
```

# Stroke Circle Hit-Test

Given:

- Mouse position  $(x, y)$
- Upper-left bound of circle  $(x, y)$
- Diameter  $d$
- Delta  $\Delta$



Hit:

```
if distance
    from (x, y)
    to (x + d/2, y + d/2)
    between d/2 - Δ and d/2 + Δ
```

```
override fun isHit(x: Double, y: Double): Boolean {
    return sqrt(sqrt(x - this.x - d / 2.0) +
               sqrt(y - this.y - d / 2.0)).between(d/2.0 - delta,
                                                       d/2.0 + delta)
}
```

# Filled Rectangle Hit-Test

Given:

- Mouse position ( $x, y$ )
- Upper-left bound of rectangle ( $x, y$ )
- Width and height ( $w, h$ )

( $x, y, w, h$ )

( $x, y$ )



Hit:

```
if  $x$  between  $x$  and  $x + w$  and  
 $y$  between  $y$  and  $y + h$ 
```

```
override fun isHit(x: Double, y: Double): Boolean {  
    return x.between(this.x, this.x + w) and  
        y.between(this.y, this.y + h)  
}
```

# Stroke Rectangle Hit-Test

Given:

- Mouse position ( $x, y$ )
- Upper-left bound of rectangle ( $x, y$ )
- Width and height ( $w, h$ )
- Delta  $\Delta$

( $x, y, w, h, \Delta$ )

Hit:

```
if ( $x, y$ ) is  
    inside ( $x - \Delta, y - \Delta, w + 2\Delta, h + 2\Delta$ ) and  
    not inside ( $x + \Delta, y + \Delta, w - 2\Delta, h - 2\Delta$ )
```

( $x, y$ )

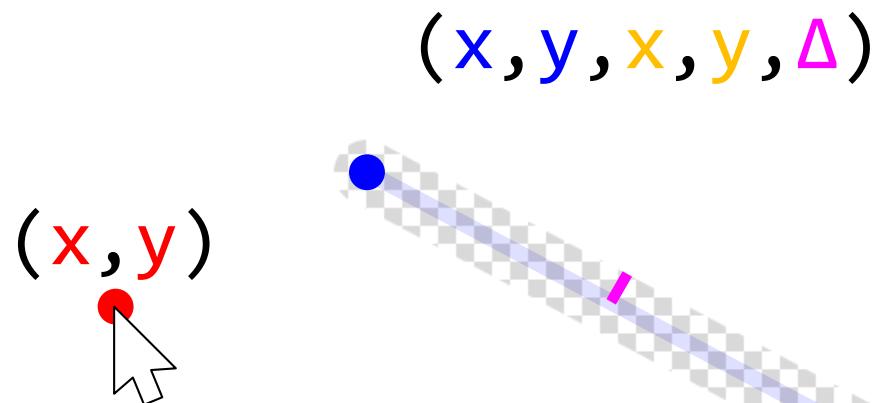


```
override fun isHit(x: Double, y: Double): Boolean {  
    return (x.between(this.x - delta, this.x + w + delta) and  
            x.between(this.x + delta, this.x + w - delta).not() and  
            y.between(this.y - delta, this.y + h + delta)) or  
        (y.between(this.y - delta, this.y + h + delta) and  
            y.between(this.y + delta, this.y + h - delta).not() and  
            x.between(this.x - delta, this.x + w + delta))  
}
```

# Stroke Line Hit-Test

Given:

- Mouse position  $(x, y)$
- Start vertex  $(x, y)$
- End vertex  $(x, y)$
- Delta  $\Delta$



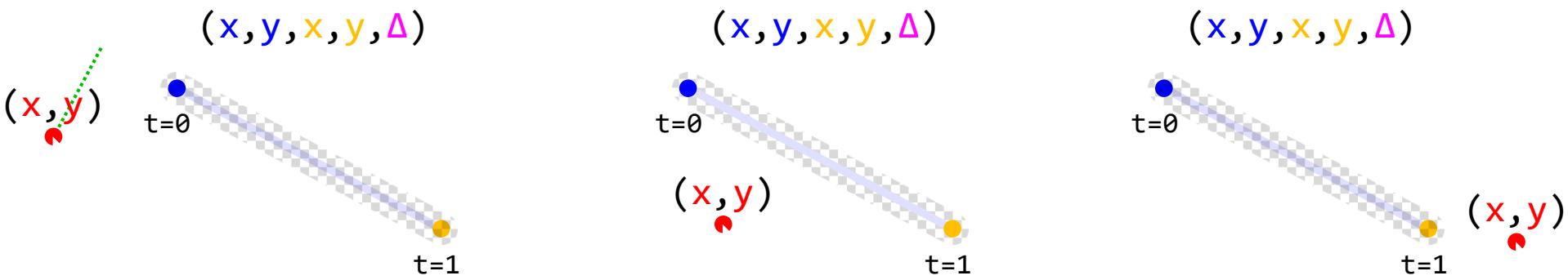
Hit:

```
if distance  
  from  $(x, y)$   
  closest point on  $(x, y, x, y)$   
   $\leq \Delta$ 
```

# Stroke Line Hit-Test

Hit:

```
if distance  
    from ( $x, y$ )  
    closest point on ( $\textcolor{blue}{x}, \textcolor{blue}{y}$ ,  $\textcolor{orange}{x}, \textcolor{orange}{y}$ )  
     $\leq \Delta$ 
```



Closest point on  $(\textcolor{blue}{x}, \textcolor{blue}{y}, \textcolor{orange}{x}, \textcolor{orange}{y})$  from  $(x, y)$ :

$$\vec{u} = \vec{m} - \vec{s}$$

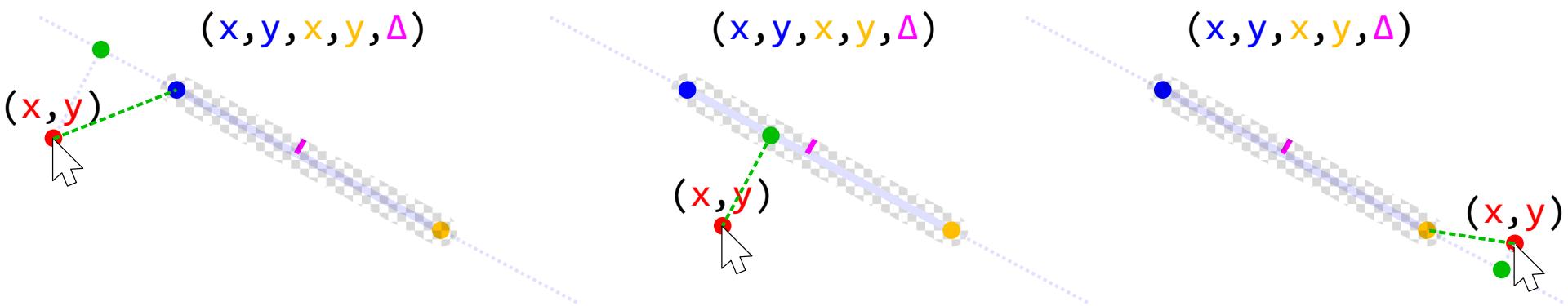
$$\vec{v} = \vec{e} - \vec{s}$$

$$t = \frac{\vec{u} \cdot \vec{v}}{\vec{v} \cdot \vec{v}}$$

# Stroke Line Hit-Test

Hit:

```
if distance  
    from (x,y)  
    closest point on (x,y, x,y)  
    <= Δ
```



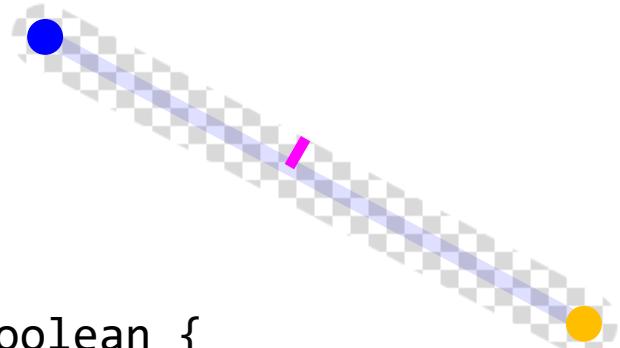
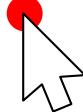
Closest point on  $(x,y, x,y)$  from  $(x,y)$ :

$$= \begin{cases} \vec{s}, & t \leq 0 \\ \vec{s} + t\vec{v}, & 0 < t < 1 \\ \vec{e}, & t \geq 1 \end{cases}$$

# Stroke Line Hit-Test

( $x, y, x, y, \Delta$ )

( $x, y$ )



```
override fun isHit(x: Double, y: Double): Boolean {  
    val ux = x - this.x  
    val uy = y - this.y  
    val vx = xe - this.x  
    val vy = ye - this.y  
    val t = (vx * ux + vy * uy) / ((vx * vx) + (vy * vy))  
    val dst = if (t < 0.0) sqrt(sqr(x - this.x) + sqr(y - this.y))  
        else if (t > 1.0) sqrt(sqr(x - xe) + sqr(y - ye))  
        else sqrt(sqr(x - (this.x + vx*t)) + (y - sqr(this.y+vy*t)))  
    return dst <= hitDelta  
}
```

# Polyline / Stroke Polygon Hit-Test

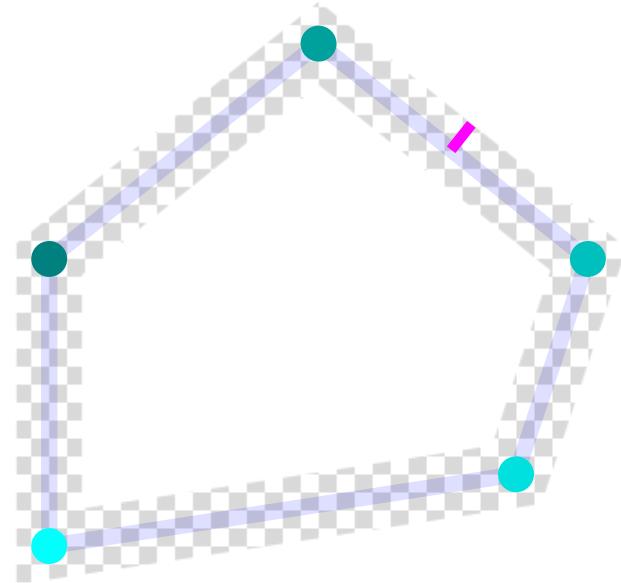
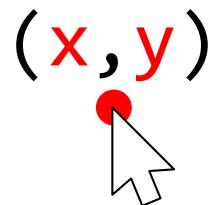
Given:

- Mouse position  $(x, y)$
- Vertices  $(x, y, \dots, x, y)$
- Delta  $\Delta$

$(x, y, \dots, x, y, \Delta)$

Hit:

if any line segment is hit

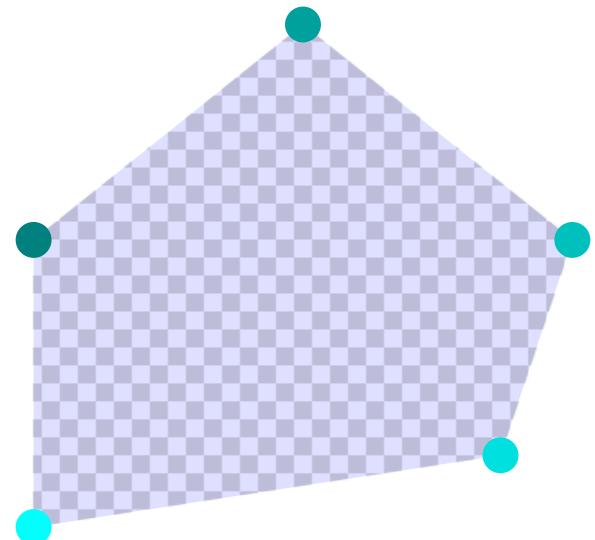


# Filled Polygon Hit-Test

Given:

- Mouse position ( $x, y$ )
- Vertices ( $x, y, \dots, x, y$ )

( $x, y, \dots, x, y$ )



# Filled Polygon Hit-Test

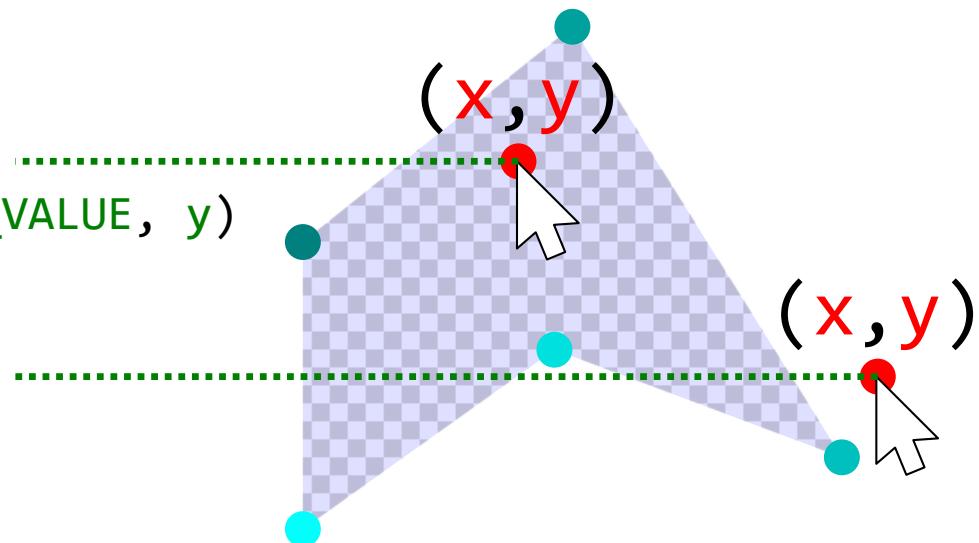
Given:

- Mouse position ( $x, y$ )
- Vertices ( $x, y, \dots, x, y$ )

( $x, y, \dots, x, y$ )

Hit:

```
if ray cast to (Double.MIN_VALUE, y)  
intersects 2k+1 edges (*)
```



# Filled Polygon Hit-Test

Given:

- Mouse position ( $x, y$ )
- Vertices ( $x, y, \dots, x, y$ )

( $x, y, \dots, x, y$ )

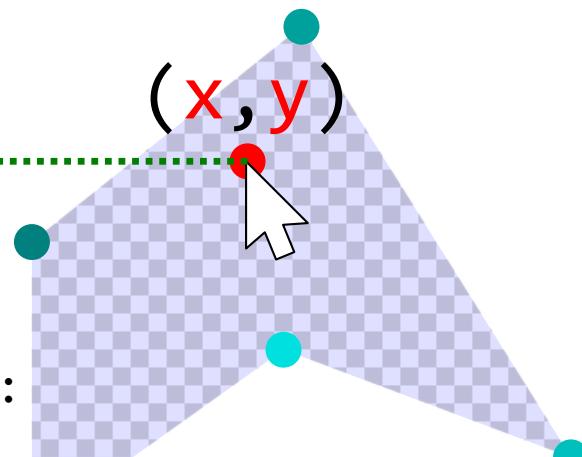
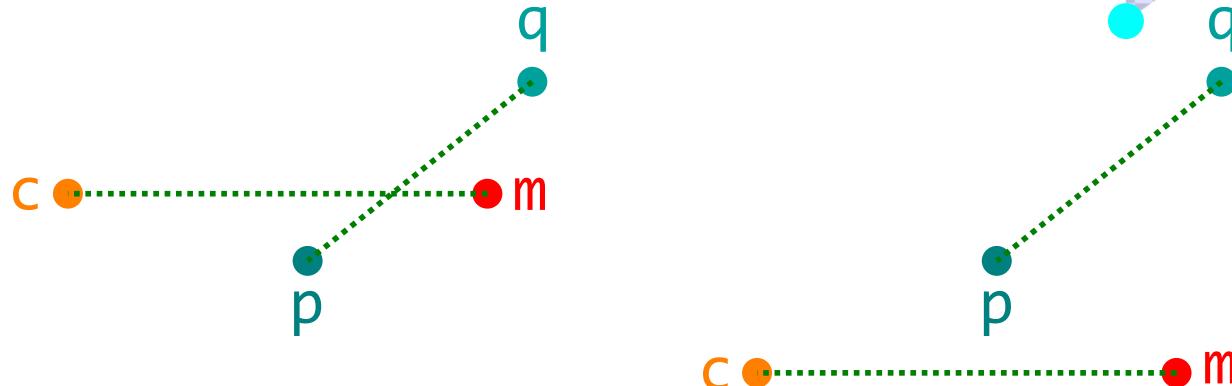
Intersection:

two line segments ( $m, c$ ), ( $p, q$ )

intersect if the orientations

( $m, c, p$ ), ( $m, c, q$ ) and

( $p, q, m$ ), ( $p, q, c$ ) have different signs:



# Filled Polygon Hit-Test

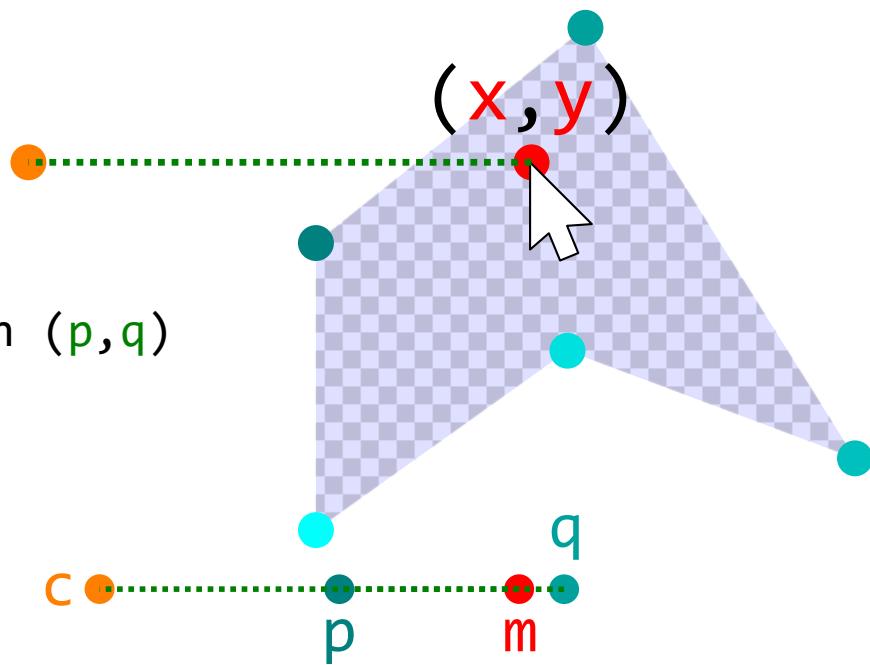
Given:

- Mouse position ( $x, y$ )
- Vertices ( $x, y, \dots, x, y$ )

( $x, y, \dots, x, y$ )

Intersection:

if ( $m, c, p$ ), ( $m, c, q$ ),  
( $p, q, m$ ), and ( $p, q, c$ ) are  
co-linear, check if  $m$  is on ( $p, q$ )



# Filled Polygon Hit-Test

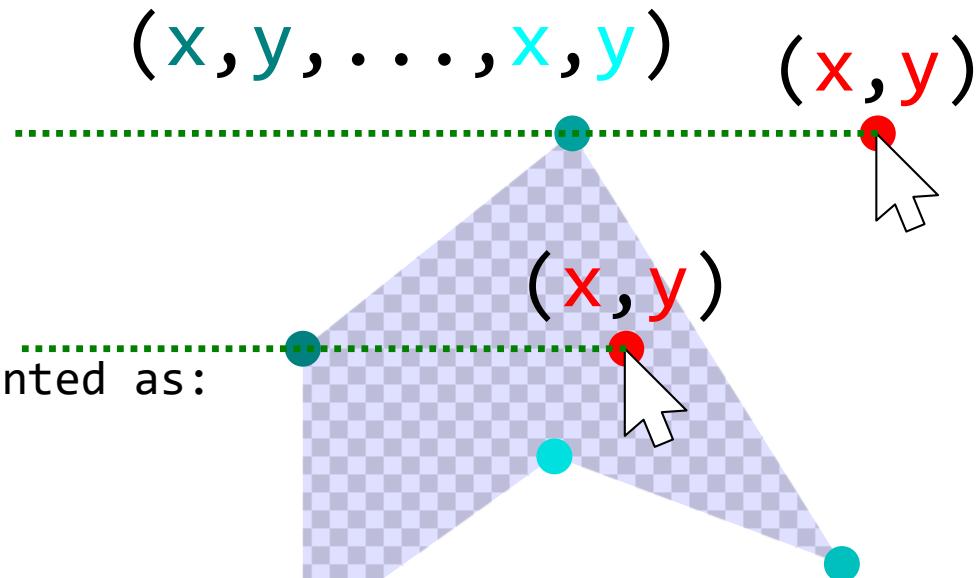
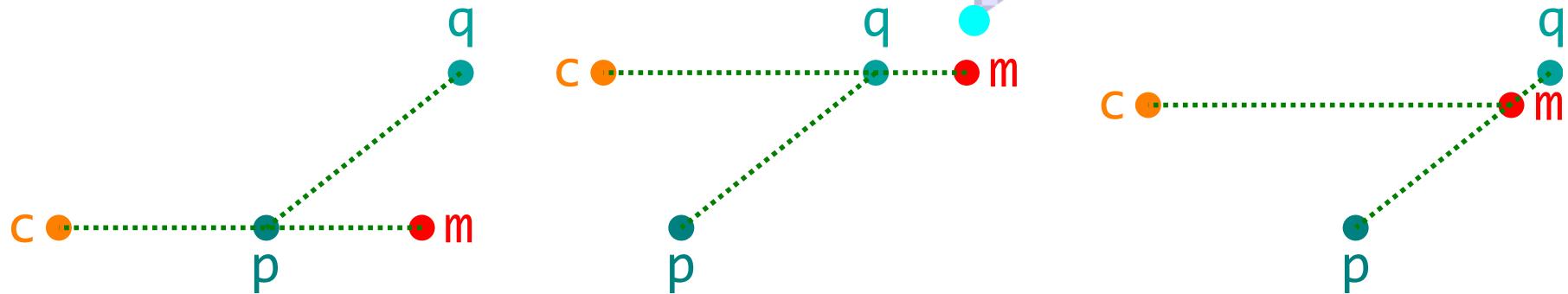
Given:

- Mouse position  $(x, y)$
- Vertices  $(x, y, \dots, x, y)$

Hit:

if  $(m, c, p)$  or  $(m, c, q)$

are co-linear, they are counted as:



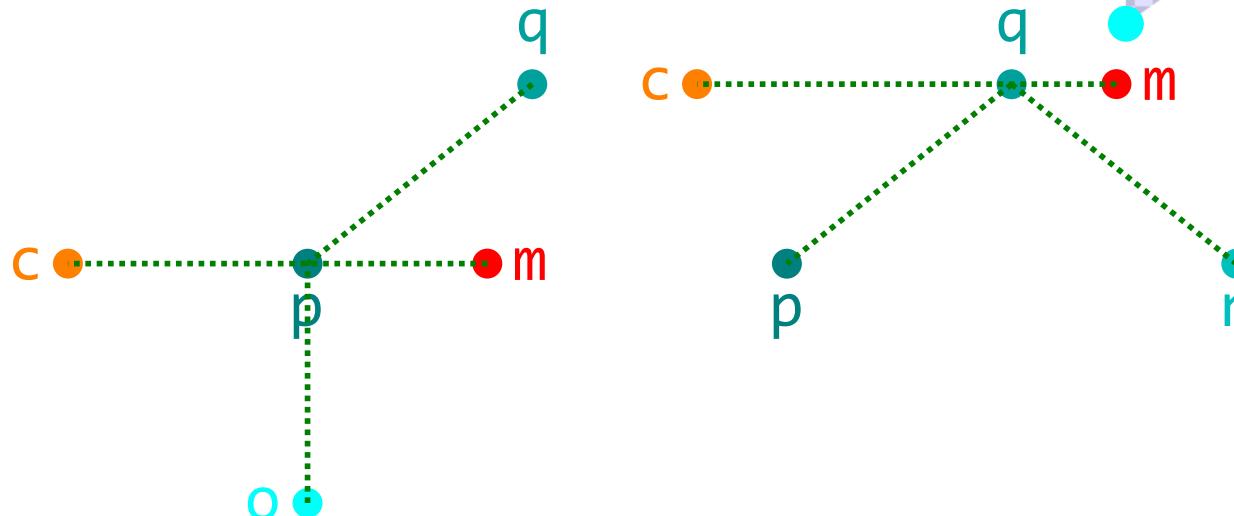
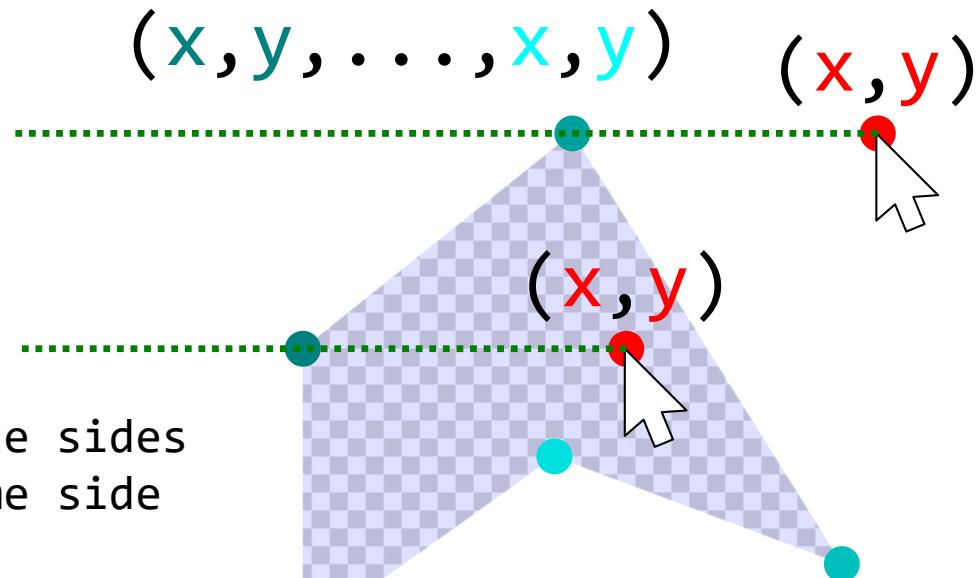
# Filled Polygon Hit-Test

Given:

- Mouse position  $(x, y)$
- Vertices  $(x, y, \dots, x, y)$

Hit:

- + 1 if edges are on opposite sides
- + 0 if edges are on the same side



# Optimizations

Hit-testing could become computationally intensive

- There could be hundred of shapes in a scene
- Polygon or Polyline shapes could have hundreds of edges

Approaches to reduce hit-testing computation:

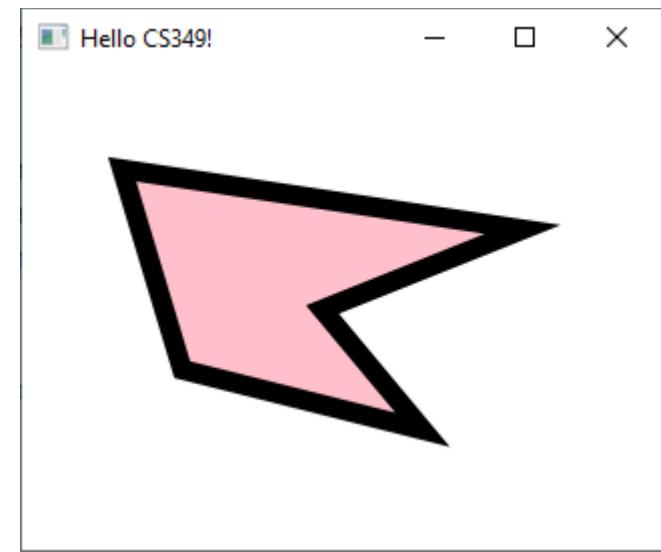
- Avoid `sqrt` in distance calculations  
(for circles, check if `sqr(dist)` is less than `sqr(diameter / 2.0)`)
- Use simpler less precise hit-test first for an “early” reject  
(e.g., start with a bounding-rectangle, or bounding circle hit-test)
- Split scene into cells, and track which ones each shape is in (e.g., octree or binary space partition)

# JavaFX Shape Hit-Testing

All JavaFX Shapes implement a contains to hit-test against a point:

```
val poly = Polygon().apply {
    fill = Color.PINK
    points.addAll( 50.0, 50.0, 250.0, 80.0, 150.0, 120.0,
                   200.0, 180.0, 80.0, 150.0)
    strokeWidth = 10.0
    stroke = Color.BLACK
}
val scene = Scene(Group(poly), 320.0, 240.0).apply {
    scene.addEventFilter(MouseEvent.MOUSE_MOVED) {
        println(poly.contains(it.sceneX, it.sceneY))
}}
```

It handles stroke thickness (hit if point is on visible stroke) and unfilled shapes (true if point is on visible stroke area).



# JavaFX Shape Hit-Testing

All JavaFX Shapes implement a `contains` to hit-test against a point:

```
val poly = Polygon().apply {
    fill = Color.PINK
    points.addAll( 50.0, 50.0, 250.0, 80.0, 150.0, 120.0,
                   200.0, 180.0, 80.0, 150.0)
    strokeWidth = 10.0
    stroke = Color.BLACK
    rotate = 90.0
}
val scene = Scene(Group(poly), 320.0, 240.0).apply {
    scene.addEventFilter(MouseEvent.MOUSE_MOVED) {
        println(poly.contains(it.sceneX, it.sceneY))
    }
}
```

It does not handle transformations! (Instead,  
use event handling in `Polygon` directly.)



# End of Chapter



Any further questions?

-



X

# Animation

What is Animation

Basic Animation

Smooth Animation

Keyframe Animation

U

CS 349

March 1

-



X

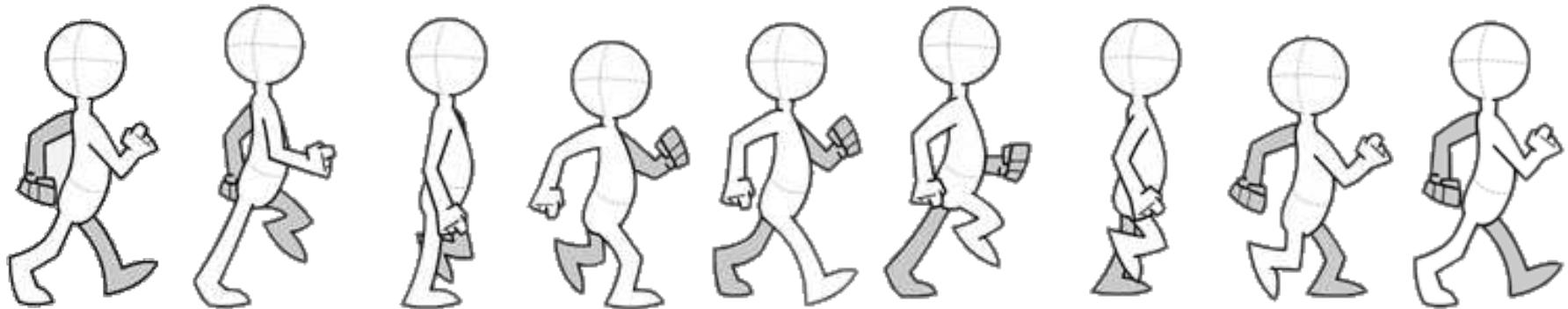
# What is Animation

U

CS 349

# Animation

Animation is the simulation of movement using a series of images (or drawings, models, etc.).



# Animation Terminology

**Frame**: each image (or state) of an animation sequence

**Frame rate**: number of frames to display per second

**Key Frame**: defines the beginning and ending points of a transition

**Tweening**: interpolation of frames between two key frames

**Easing**: a function that controls how tweening is calculated

# Animation Terminology

In user interface programming, we typically animate numerical parameters that change how graphics are drawn over time.

- parameters are often related to transformations  
(e.g. translate X and Y position to animate drawing position)
- parameters can be anything numeric: fill, stroke weight, etc.
- animating non-numeric values (e.g. a String or Image) is possible, but custom tweening methods are needed

# Frame Rate

Measured in **frames-per-second (fps)**. Can be expressed as Hertz (Hz): International System of Units (SI) measure defined as one cycle per second (e.g., 60 FPS = 60 Hz)

Common device and media frame rates:

- Hand-drawn animation: as low as 12 FPS, usually 24 fps
- GIFs: usually 15 to 24 fps
- Film: standard 24 fps, high framerate 60 fps
- Legacy Broadcast Television: NTSC: 30 fps\*, PAL 25 fps\*
- Computer displays: 60 fps or more
- Computer games: 60 fps or more
- Virtual Reality displays: 90 fps, 120 fps, or more

U

CS 349

# Basic Animation



# Animation Using `java.util.Timer`

A **timer** triggers an event after some time period

1. Set time period to time interval for desired frame rate, e.g., 50 FPS (i.e., frequency of  $f = 1/50 \text{ Hz}$ , new frame every  $1/50 \text{ s} = 20 \text{ ms}$ )
2. In the timer event handler
  1. update parameters you want to animate
  2. redraw an updated image for the frame
3. Restart the timer for the next interval

```
val animation = Timer().apply {
    scheduleAtFixedRate(object : TimerTask() {
        override fun run() {
            myCanvas.graphicsContext2D.apply {
                clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
                myDrawable.y += 2.0
                myDrawable.draw(myCanvas.graphicsContext2D)
            }
        }
    }, 0L, 20L)
}
```

# Animation Using java.util.Timer

```
private val animation = Timer()

override fun start(stage: Stage) {
    val myDrawable = FillCirc(0.0, 0.0, 50.0, Color.GREEN, "Green Circle")
    val myCanvas = Canvas(480.0, 320.0)
    animation.apply {
        scheduleAtFixedRate(object : TimerTask() {
            override fun run() {
                myCanvas.graphicsContext2D.apply {
                    clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
                    myDrawable.y += 2.0
                    myDrawable.draw(myCanvas.graphicsContext2D)
                }
            }
        }, 0L, 20L)
    }
    stage.title = "Hello CS349!"
    stage.scene = Scene(Group(myCanvas), myCanvas.width, myCanvas.height)
    stage.show()
}

override fun stop() {
    super.stop()
    animation.cancel()
}
```

# Timers and the UI Thread

Many UI frameworks are single-threaded (including JavaFX)

- the event dispatch queue is one thread to avoid deadlocks and race conditions due to unpredictable user-generated events

These UI frameworks are typically not thread-safe

- to reduce execution burden, complexity, etc.

Most modifications to the scene graph (and nodes it contains) must be performed on the UI execution thread

- otherwise, an exception is thrown

This has implications for animation timers:

The three previous timers trigger the event on the UI thread, so the handler can modify anything in the scene graph.

But the next timer isn't on the UI thread ...

# Animation Using `java.util.Timer`

```
val animation = Timer().apply {
    scheduleAtFixedRate(object : TimerTask() {
        override fun run() {
            myCanvas.graphicsContext2D.apply {
                clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
                myDrawable.y += 2.0
                myDrawable.draw(this)
            }
        }
    }, 0L, 20L)
}
```

`java.util.Timer` does not run on the JavaFX application thread: it may cause an exception if modifications to the scene graph are attempted in the event handler.

# Animation Using `java.util.Timer` & `Platform.runLater`

```
val animation = Timer().apply {
    scheduleAtFixedRate(object : TimerTask() {
        override fun run() {
            Platform.runLater {
                myCanvas.graphicsContext2D.apply {
                    clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
                    myDrawable.y += 2.0
                    myDrawable.draw(this)
                }
            }
        }
    }, 0L, 50L)
}
```

`Platform.runLater` runs the specified `Runnable` on the JavaFX application thread at some unspecified time in the future.

# Animation Using javafx.animation.AnimationTimer

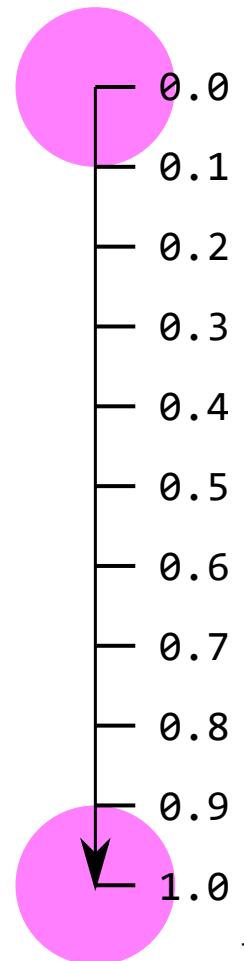
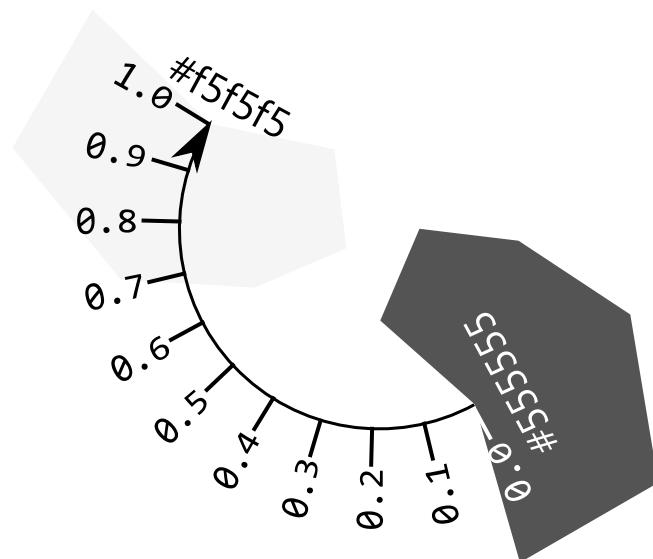
```
val animation = object : AnimationTimer() {
    override fun handle(now: Long) {
        myCanvas.graphicsContext2D.apply {
            myDrawable.y += 2.0
            clearRect(0.0, 0.0, myCanvas.width, myCanvas.height)
            myDrawable.draw(this)
        }
    }
}.start()
```

AnimationTimer runs the specified Runnable on the JavaFX UI thread at 60 fps.

# Animation Control by the Drawable

Animation can be thought of as moving through a range from  $0.0$ , which represents the start state, to  $1.0$ , which represents the end state.

The animation value is mapped to one or more visual or other property (e.g., location, colour, text), each with a definition of start and end states.



# Animation Control by the Drawable

Animation can be thought of as moving through a range from `0.0`, which represents the start state, to `1.0`, which represents the end state.

```
private var animValue = 0.0 // [0.0 ... 1.0]
```

The animation value is mapped to one or more visual or other property (e.g., location, colour, text), each with a definition of start and end states.

```
private var animPropStart = 0.0 // start state  
private var animPropEnd = 200.0 // end state
```

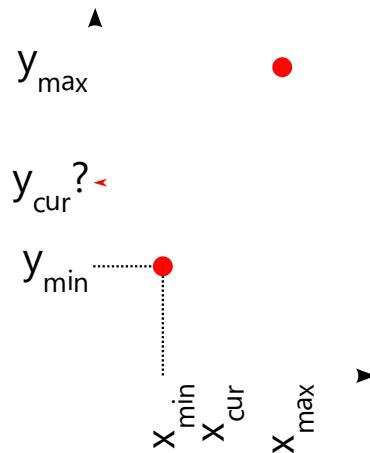
The value has a certain change rate per frame / call to `animate`.

```
private var animSpeed = 0.005
```

# Animation Control by the Drawable

While start and end states are known, intermediate states must be calculated.

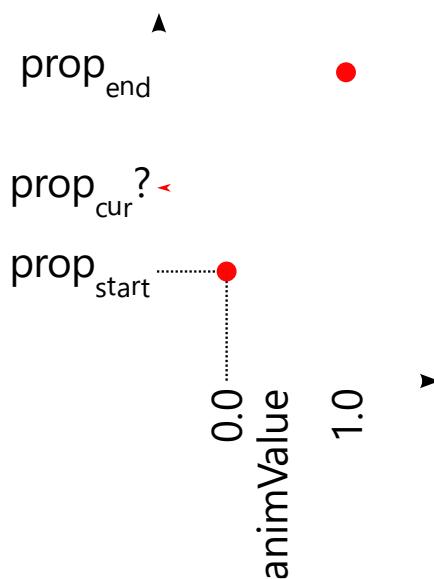
A basic approach for this is linear interpolation:



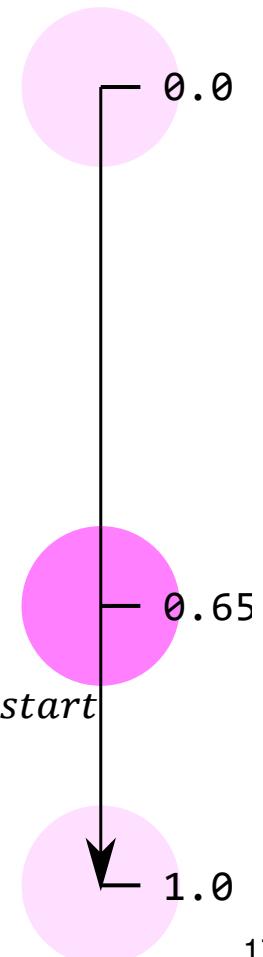
$$y_{cur} = \frac{y_{min}(x_{max}-x_{cur})+y_{max}(x_{cur}-x_{min})}{x_{max}-x_{min}}$$

# Animation Control by the Drawable

The start and end states of the property are associated with x-values 0.0 and 1.0 respectively, and intermediate values of the property  $prop_{cur}$  must be interpolated from the current value of the animation  $animValue_{cur}$  ("tweening").



$$prop(animValue) = animValue * (prop_{end} - prop_{start}) + prop_{start}$$



# Animation Control by the Drawable

```
fun Double.lerp(min: Double, max: Double) : Double {  
    return this * (max - min) + min  
}  
  
enum class Direction(val value: Double) { UP(-1.0), DOWN(1.0) }  
  
private var animValue = 0.0          // [0.0 ... 1.0]  
private var animSpeed = 0.005        // change to animValue per call  
private var animDirection = Direction.DOWN  
private var animPropertyStart = 50.0 // property start state  
private var animPropertyEnd = 250.0 // property end state  
  
override fun animate() {  
    animValue += animSpeed * animDirection.value  
    y = animValue.lerp(animPropertyStart, animPropertyEnd)  
    when {  
        animValue + animSpeed > 1.0 -> animDirection = Direction.UP  
        animValue - animSpeed < 0.0 -> animDirection = Direction.DOWN  
    }  
}
```

U

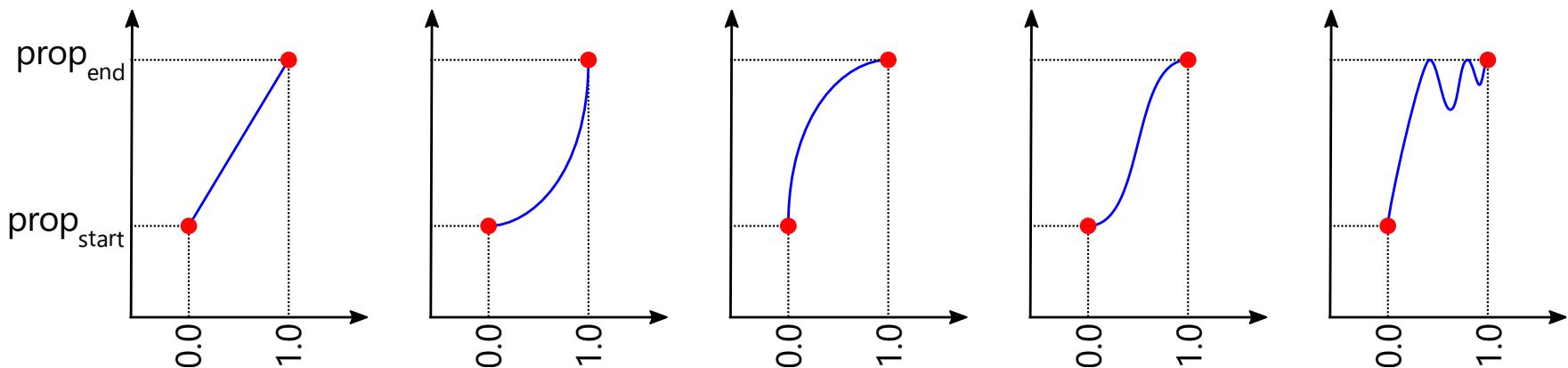
CS 349

# Smooth Animation



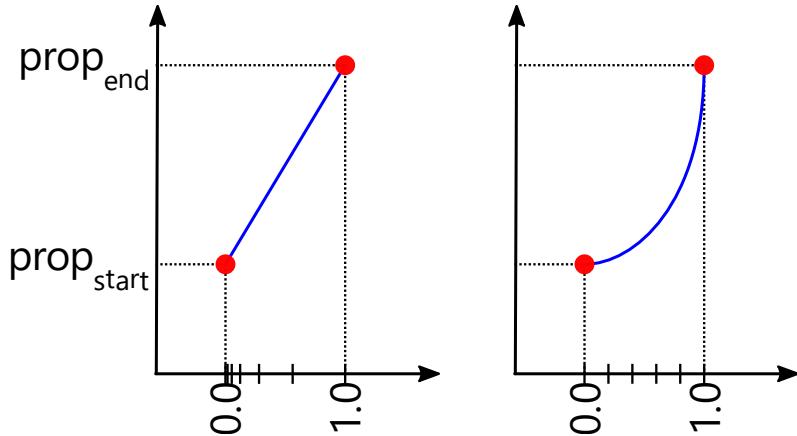
# Easing

Oftentimes, linear interpolation is not good enough (“feels unnatural”), and other types of interpolations would be preferred.



# Easing

This can be achieved by “easing” the current animation value.

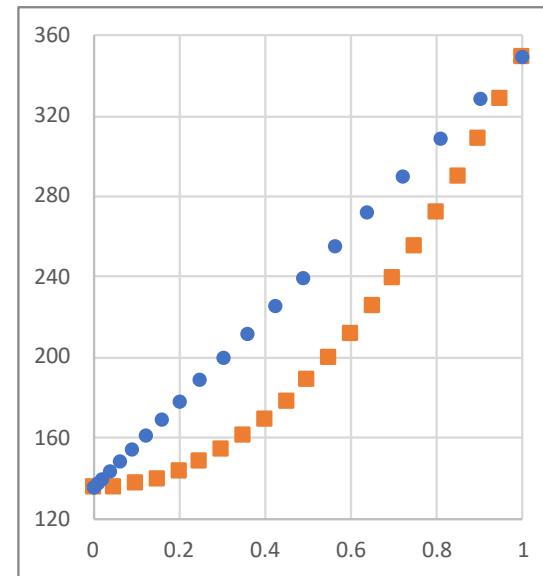
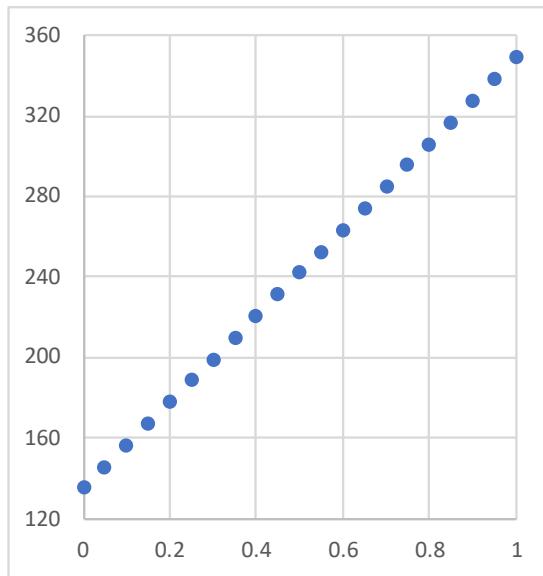
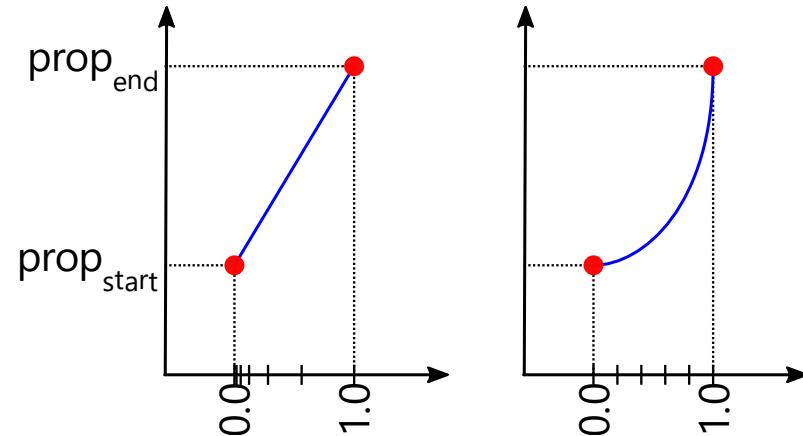
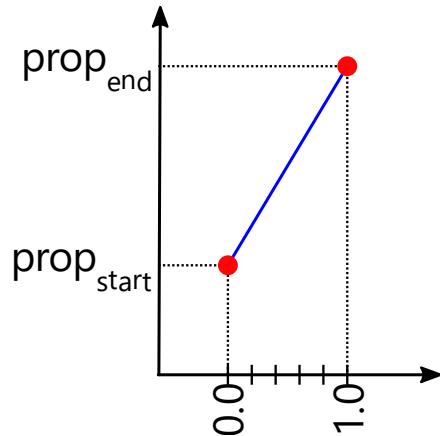


$$prop(animValue) = ease(animValue) * (prop_{end} - prop_{start}) + prop_{start}$$

```
fun Double.lerp(min: Double, max: Double) : Double {  
    return this * (max - min) + min  
}  
fun easeIn(cur: Double) : Double {  
    return x.pow(2)  
}  
curProp = easeIn(animValue).lerp(animPropStart, animPropEnd)
```

# Easing

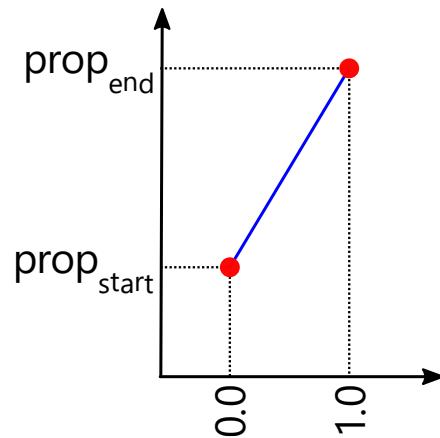
Easing the animation value results in altering the x-axis.



# Easing Functions

```
val flip = { x: Double -> 1.0 - x }
val easeIn = { x: Double -> x.pow(2) }
val easeOut = { x: Double -> flip(easeIn(flip(x))) }
val easeInOut = { x: Double -> x.lerp(easeIn(x), easeOut(x)) }

// interpolate value with no easing (i.e., linear)
curProp = animValue.lerp(animPropStart, animPropEnd)
```

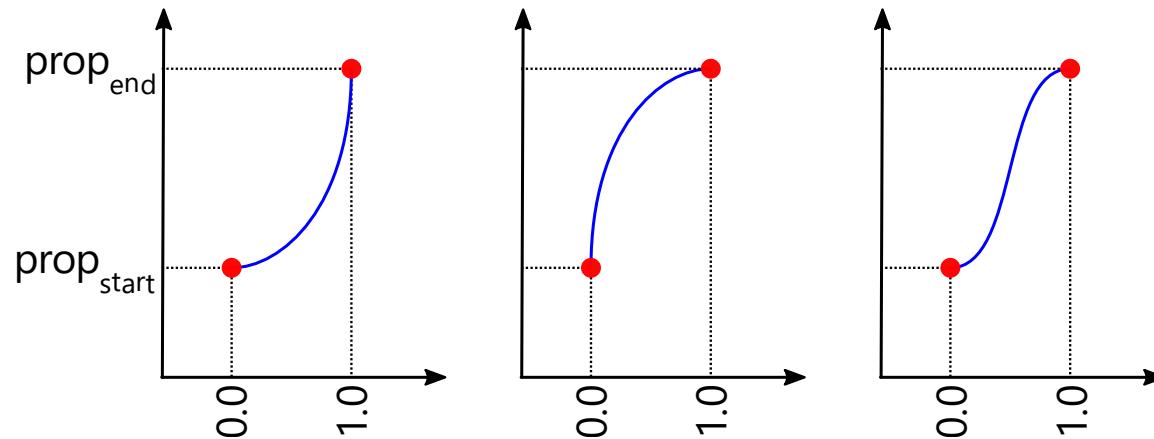


# Easing Functions

```
// interpolate value with easeIn (i.e., quadratic)
curProp = easeIn(animValue).Lerp(animPropStart, animPropEnd)

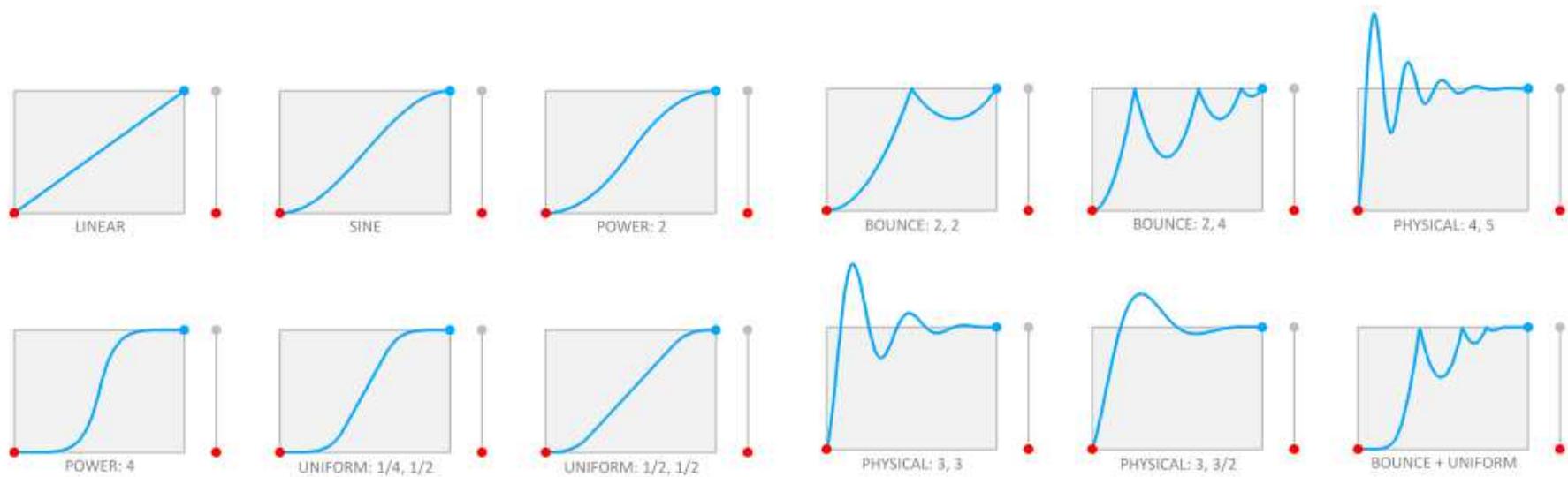
// interpolate value with easeOut (i.e., flipped quadratic)
curProp = easeOut(animValue).Lerp(animPropStart, animPropEnd)

// interpolate value with easeInOut
curProp = easeInOut(animValue).Lerp(animPropStart, animPropEnd)
```



# Easing Function Resources

- <http://robertpenner.com/easing/>
- <https://greensock.com/docs/v3/Eases>
- <https://www.febucci.com/2018/08/easing-functions/>



[VIDEO SOURCE] <https://www.alanzucconi.com/2021/01/24/piecewise-interpolation/easing-curves/>

# Animation Using javafx.animation.Transition

Basic ‘tweening’ animations include:

- TranslateTransition, RotateTransition, ScaleTransition
- FillTransition, StrokeTransition
- FadeTransition: dissolve node visibility in or out
- SequentialTransition: run multiple transitions in a sequence
- ParallelTransition: run multiple transitions at the same time

Available interpolations include:

- Interpolator.*LINEAR*
- Interpolator.*DISCRETE*
- Interpolator.*EASE\_IN*
- Interpolator.*EASE\_OUT*
- Interpolator.*EASE\_BOTH*
- custom splines

# Animation Using javafx.animation.Transition

```
override fun start(stage: Stage) {
    val drawable = Circle(20.0, 20.0, 20.0, Color.BLUE)
    val animation = TranslateTransition(Duration.millis(4000.0),
                                         drawable).apply {
        byY = 200.0
        interpolator = Interpolator.EASE_BOTH
        isAutoReverse = true
        cycleCount = Transition.INDEFINITE
    }
    animation.play()
    stage.title = "Hello CS349!"
    stage.scene = Scene(Group(drawable), 320.0, 240.0)
    stage.show()
}
```

-



X

# Keyframe Animation

U

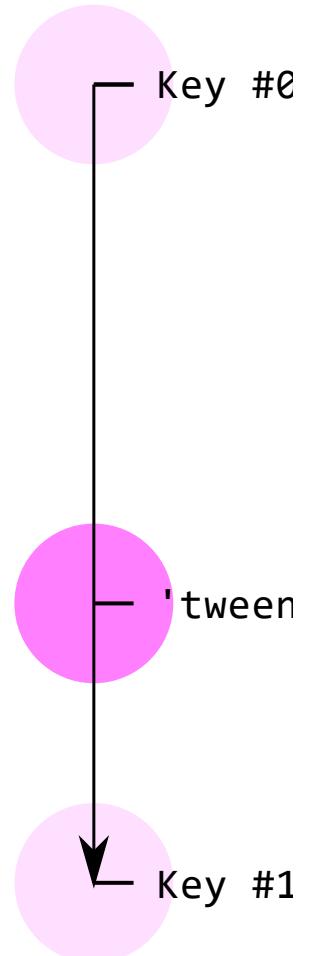
CS 349

# Key Frames and Timeline

A Timeline is a sequence of two (or more) Key Frames.

A Key Frame defines an end point of a transition of one or more Key Values.

A Key Value represents a object's Property, its desired value, and the method of interpolation.



# Animation Using javafx.animation.Timeline

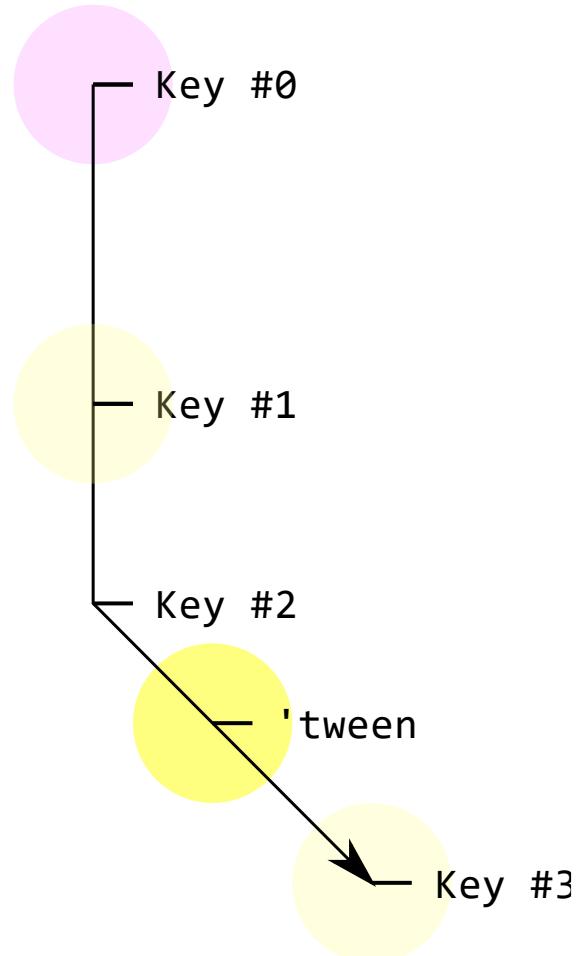
```
override fun start(stage: Stage) {
    val drawable = Circle(20.0, 20.0, 20.0, Color.BLUE)
    val animation = Timeline(KeyFrame(Duration.millis(4000.0),
        KeyValue(drawable.translateYProperty(),
            200.0,
            Interpolator.EASE_BOTH))).apply {
        cycleCount = Animation.INDEFINITE
        isAutoReverse = true
    }
    animation.play()

    stage.title = "Hello CS349!"
    stage.scene = Scene(Group(drawable), 320.0, 240.0)
    stage.show()
}
```

# Key Frames and Timeline

A Timeline can have many keyframes to create more complex transitions:

- Each key frame serves as a “snap shot” of one (or more) properties at a certain time.
  - Timeline calculates ‘tweens, i.e., the values of each affected property for every frame.
- 
- Key #0: 0.0s start state (auto-generated)
  - Key #1: 1.8s fillProperty is #FFFF7F
  - Key #2: 2.4s translateY-property is 0.0
  - Key #3: 4.0s translateX-property is 200.0  
translateY-property is 50.0



# Animation Using javafx.animation.Timeline

```
override fun start(stage: Stage) {
    val drawable = Circle(20.0, 20.0, 20.0, Color.BLUE)
    val animation = Timeline(          // Key #0 is auto-generated
        KeyFrame(Duration.millis(1800.0), // Key #1
                  KeyValue(drawable.fillProperty(), Color.GREEN, ...)),
        KeyFrame(Duration.millis(2400.0), // Key #2
                  KeyValue(drawable.translateXProperty(), 0.0, ...)),
        KeyFrame(Duration.millis(4000.0), // Key #3
                  KeyValue(drawables[0].translateYProperty(), 200.0, ...),
                  KeyValue(drawables[0].translateXProperty(), 50.0, ...))

    ).apply {
        cycleCount = Animation.INDEFINITE
        isAutoReverse = true
    }
    animation.play()

    stage.title = "Hello CS349!"
    stage.scene = Scene(Group(drawable), 320.0, 240.0)
    stage.show()
}
```

# End of Chapter



Any further questions?

-



X

# Input

Input Devices

Text Entry

Positional Input

U

CS 349

March 6

U  
CS 349

# Input Devices

# General Purpose Input Devices

Most computing platforms use general purpose input devices

Often targeted at two high level tasks:

- text entry
- positional input



# Specific Purpose Input Devices

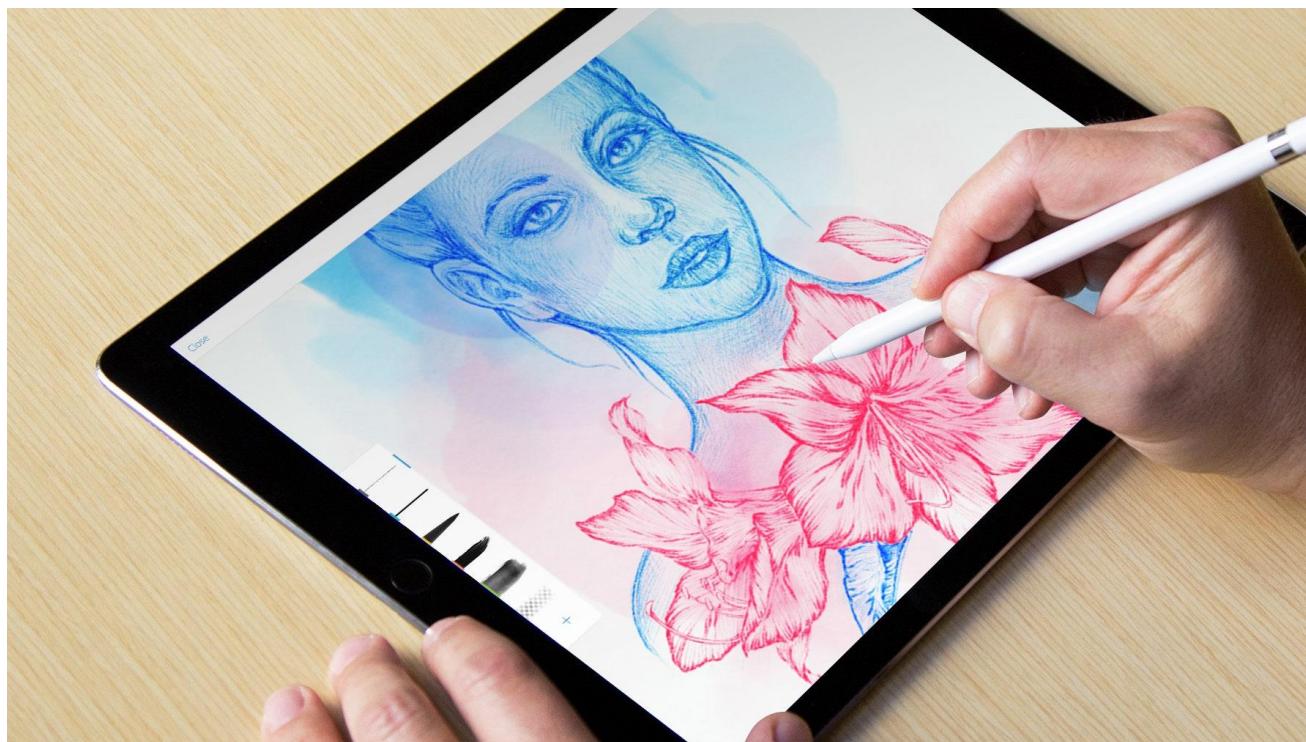
Devices can be designed for very specific UI tasks

- e.g., iPod wheel

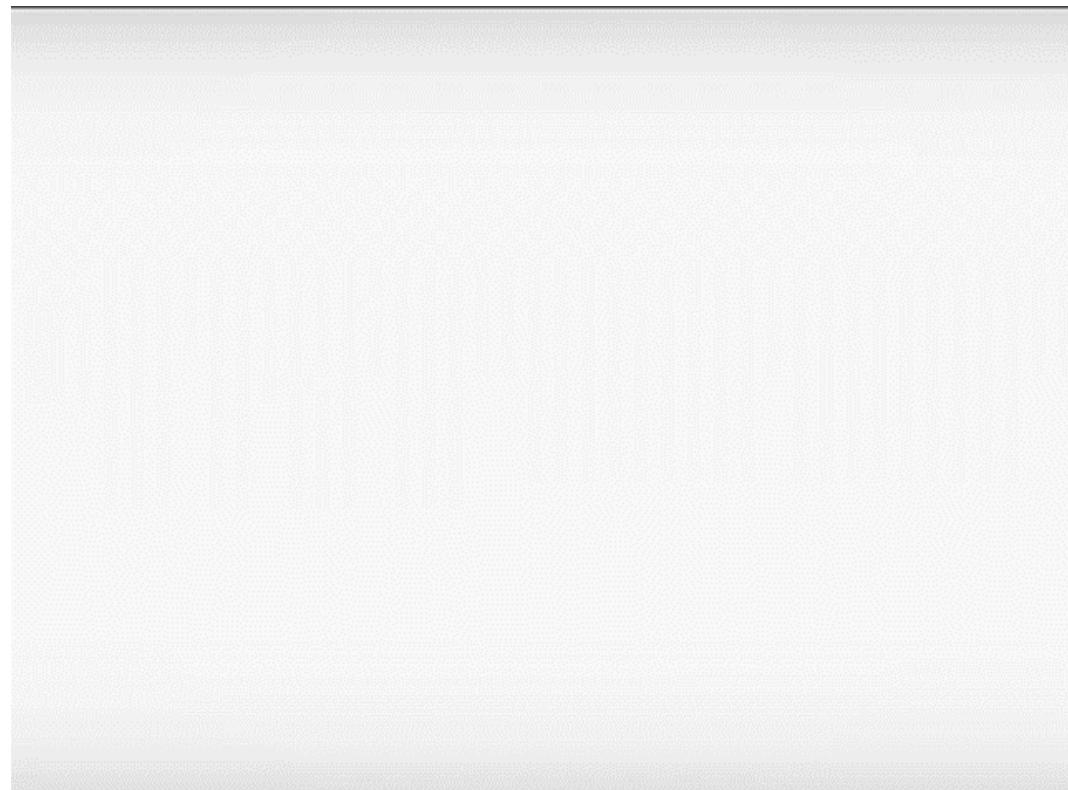


Some UI tasks are better with specific kinds of input devices

- e.g., drawing with a pen vs mouse



# Specific Purpose Input Devices



U

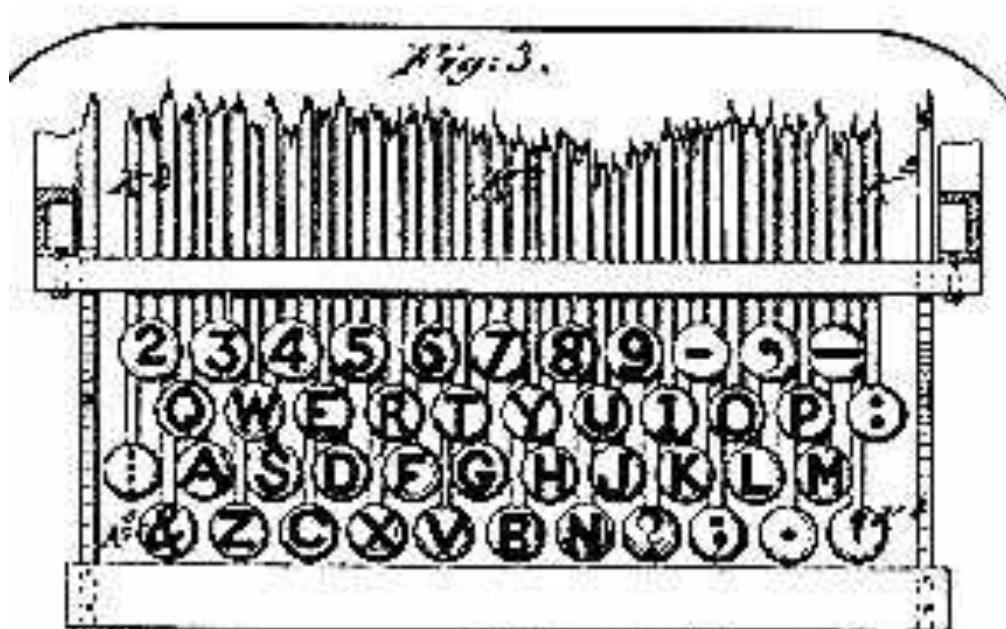
CS 349

# Text Entry

# Typewriters and QWERTY

Original design intended for typing on paper

QWERTY not designed to slow typing down.  
Instead, designed to space “typebars” to  
reduce jams and speed typing up



1874 QWERTY patent drawing



THE FIRST COMMERCIAL TYPEWRITER  
Model 1 REMINGTON, Shop No. 1.

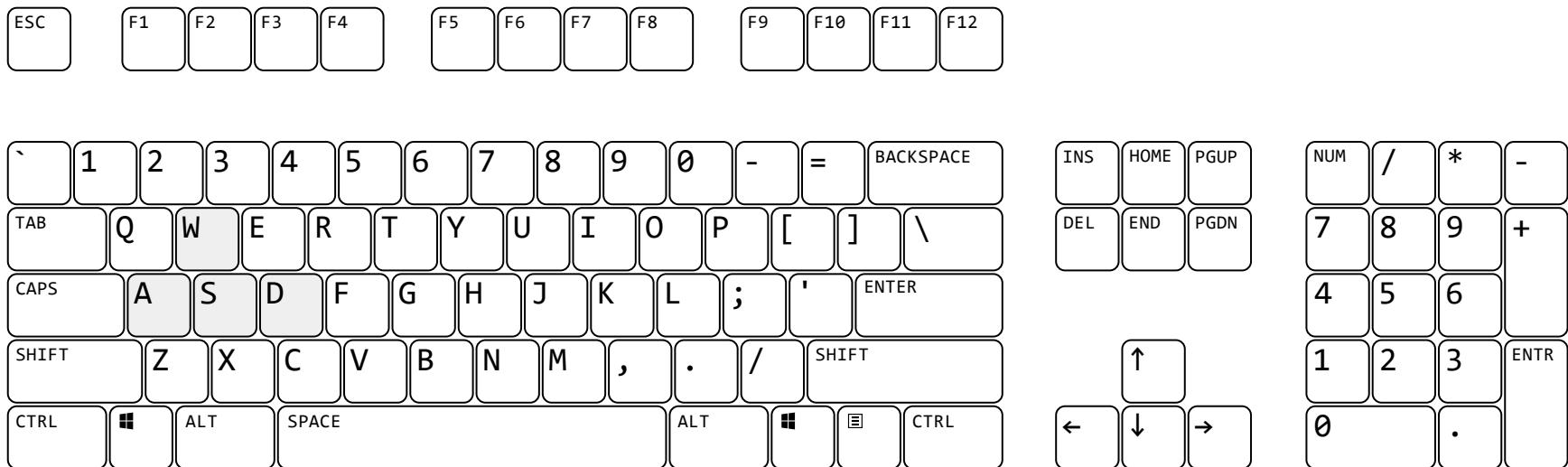
# QWERTY Problems?

## Common combinations

- awkward finger motions (e.g., t → r)
- jump over home row (e.g., b → r)
- all typed with one hand. (e.g., w → a → s, w → e → r → e)

On average more left-hand typing than right

About 16% of typing uses lower row, 52% top row, 32% home row



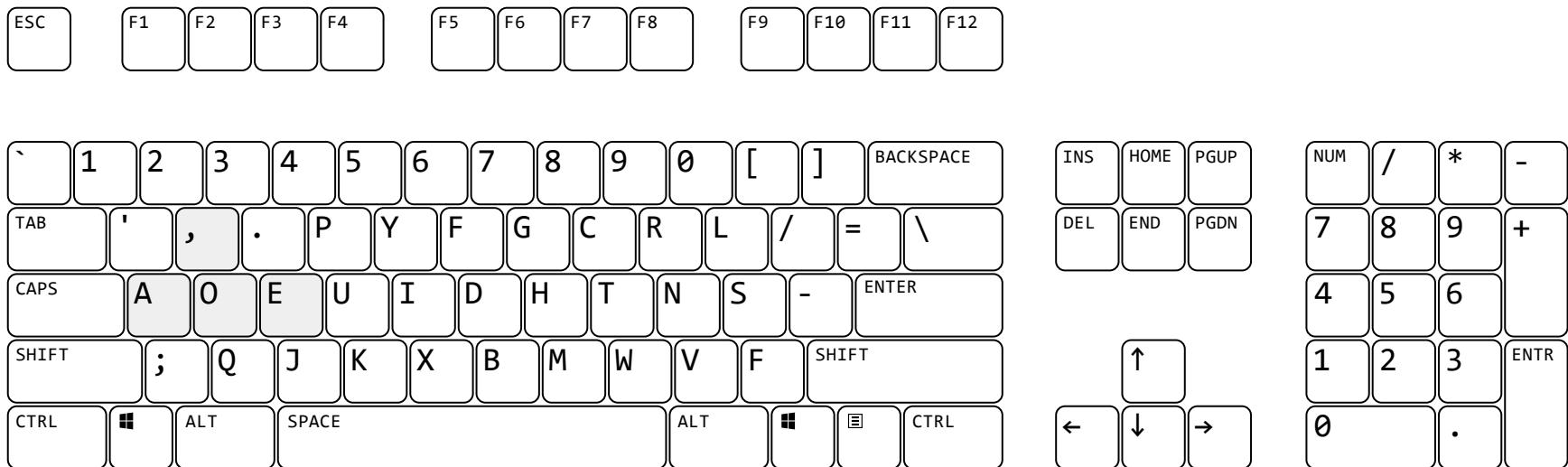
# Dvorak Optimizations

Make common letters and digraphs easiest to type

- about 70% of keyboard strokes are on home row
- least common letters on bottom row (hardest row to reach)

Right hand does more typing (assumes most people are right-handed)

Has not caught on . Why?



# Physical Keyboards

Tactile keys with activation by physical movement.



To reduce cost or increase portability, possible adjustments include:

- Rubber domes instead of springs
- Fewer and / or smaller keys
- Reduced key travel distance



Adjustment can interfere with typing efficiency

# Minimal Numeric Keyboard

Repeated presses as text entry method

- Each number is mapped to multiple letters, e.g., 2→{a,b,c}, 5→{j,k,l}, 6→{m,n,o}
- Letters are typed by pressing the associated number multiple times, e.g., 2,2,2→c
- Words are typed by typing multiple letters, e.g., 2,2→b, 6,6,6→o, [pause], 6,6,6→o, 5,5→k

Issues common to predictive text

- Generally slow due to number of button presses
- Repeated letters require additional pause



# Predictive Text for Minimal Numeric Keyboard

## T9 as text entry method

- Each number is mapped to multiple letters, e.g., 2→{a,b,c}, 5→{j,k,l}, 6→{m,n,o}
- Words are typed as a sequence of numbers, e.g., 2-6-6-5
- The word is {a,b,c}–{m,n,o}–{m,n,o}–{j,k,l}
- Given this (ambiguous) set of characters, the most likely word from a dictionary is displayed, e.g., book over anno, cook, cool, etc.



## Issues common to predictive text

- “Collisions” between common ambiguous words
- Entering words not in dictionary is difficult

# Touch Keyboards

## Problems:

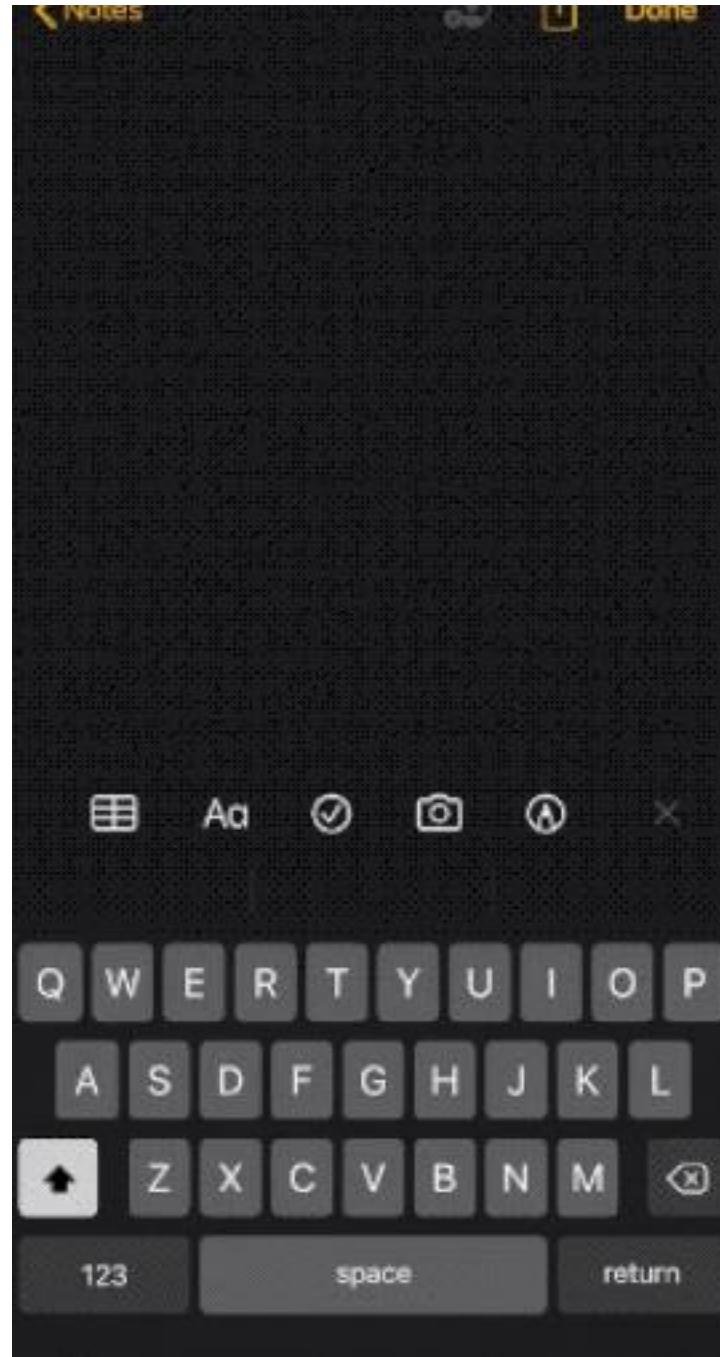
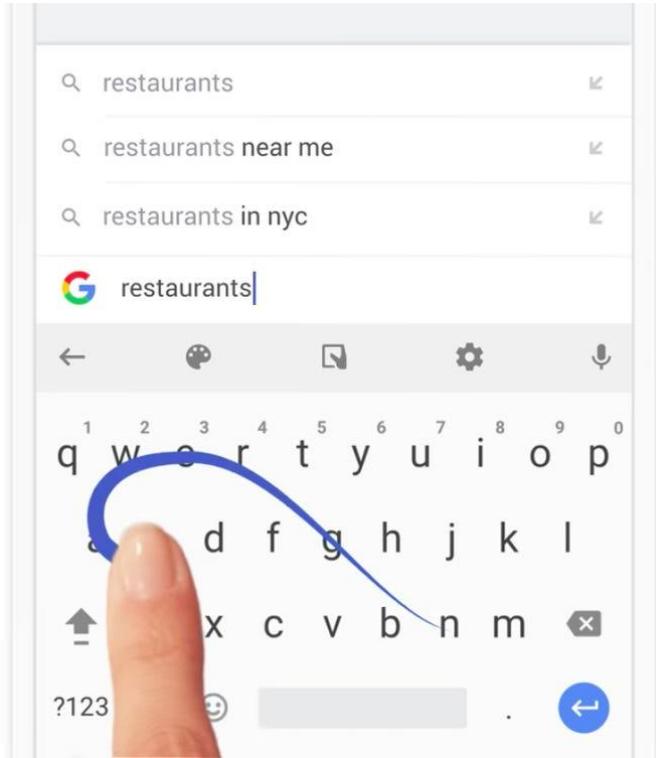
- no tactile feedback makes it hard to find the home row
- no tactile feedback makes it hard to tell if key was pressed (solved by vibration)
- resting of hands difficult
- small keys reduce accuracy

## Advantage:

- portable, no extra hardware
- customizable keys (e.g., new language, symbols, emojis)
- customizable layout and functionality (e.g., swipe, thumb layout)



# Gestural Text Input



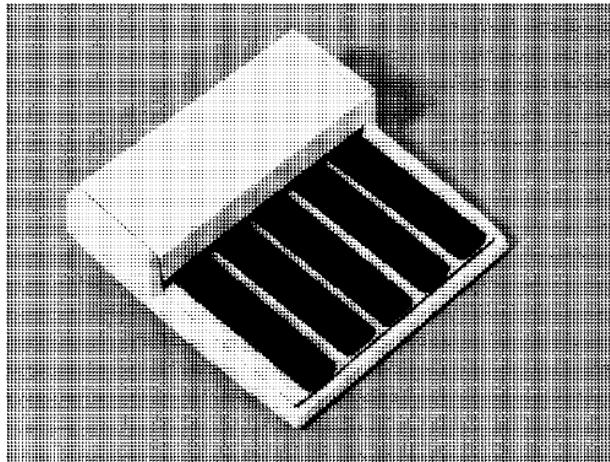
# Chording Keyboards

## Englebart's NLS Keyboard

- Multiple keys together produce letter
- No hand “targeting”, potentially very fast
- Can be small and portable
- One handed

## Thad Starner's Twiddler

- for wearable computing input



NLS Keyboard

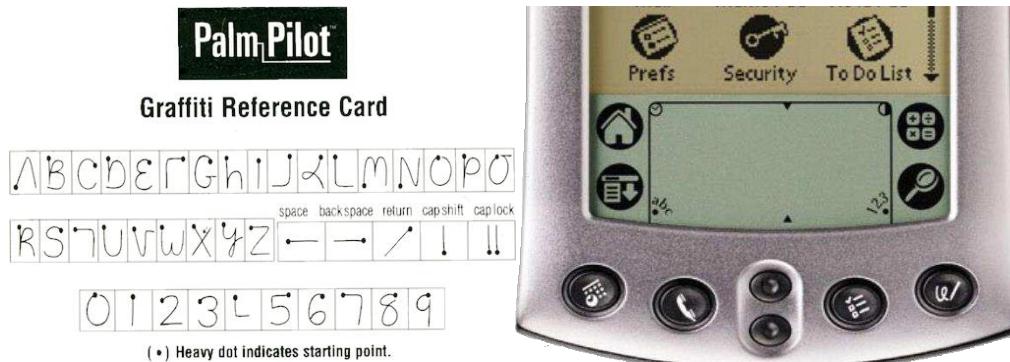


Twiddler

# Text Recognition and Gestures

Gestural strokes for letter (e.g., Graffiti / Unistroke Gestures)

- Map single strokes to characters



Natural Handwriting recognition (e.g., iPad Scribble, Microsoft Ink to Text)

- dictionary-based classification algorithms

The image shows a screenshot of the Microsoft Ink to Text application. At the top, there are toolbars for 'Tools' (with options like 'Type', 'Lasso Select', 'Panning Hand', and 'Eraser'), 'Shapes' (with a color & thickness selector), and 'Edit' (with a dropdown menu). The main area contains handwritten text in cursive script: 'This is a test!'. A dashed rectangular selection box surrounds the text. Below the text, the application displays the converted text: 'This is a test!'. On the right side, there is a 'Convert' section with buttons for 'Ink to Shape', 'Ink to Text', and 'Ink to Math'. A dropdown menu for 'Ink to Text' is open, showing the sub-option 'Convert handwriting to text.'

# Text Input Expert-User Input Rates

Device	Input Rates
Qwerty Desktop	80+ WPM proficient 150 WPM record (sustained for 50 minutes)
Qwerty Thumb	60 WPM typical with training
Soft Keyboards	45 WPM
T9	45 WPM possible for experts
Gestural	~ ShapeWriter claims 80 WPM (expert)
Handwriting	33 WPM
Graffiti 2	9 WPM

# ACII & Unicode

**ASCII** is a 1-byte encoding of the Latin alphabet.

**Unicode** is a *superset* of ASCII, that has replaced it in common use

- Values 0-127 have the same meaning in both (e.g., 'A'  $\leftrightarrow$  65)
- Uses multiple bytes to store character information, which greatly increases the range of values
- Denoted as UTF-xx where xx is the minimum number of bits.

**UTF-8** is the standard method of encoding characters

- Minimum 8 bits
- Capable of encoding all 1,112,064 code points in Unicode (characters, control codes, other meaningful characters)

# Text Validation

Interfaces often need to check text input typed by the user

- a required field (e.g., credit card number)
- a certain format (e.g., numeric, postal code, phone number)
- within a certain range (e.g., number between 0 and 100)
- unique (e.g., choose an unused username)

The screenshot shows a Windows application window titled "Hello CS349!". It contains three text input fields:

- Preferred Name:** The value "CS Three-Four-Nine" is entered into this field.
- Postal code:** The value "A1A 2B2" is entered into this field, highlighted with a green background.
- Phone number:** The placeholder text "e.g., (123) 123-1234" is displayed in this field.

A "Submit" button is located at the bottom of the form.

# Guidelines for Text Validation

Prevent invalid input through constant validation.

Accept data formatted in different ways

Have different levels stages of validation:

- Basic, in the View
- Intermediate, in the Model
- Thorough, in the backend

When input is invalid:

- Place error messages close to the source of the error
- Use colour to differentiate valid from invalid input

# Regular Expressions (regex)

A sequence of characters that specifies a search pattern in text

- developed from language theory and theoretical computer science
- a regex pattern describes a deterministic finite automaton (DFA)

Please refer to

- <https://regexone.com> (Regex Tutorial)
- <https://regex101.com> (Regex Testing, Explanation, Reference)

# Regular Expressions (regex)

Used in form validation to “test” if string can is correct format:

- Postal Code (upper case only, with space in between the two 3-tuples):

```
[A-Z][0-9][A-Z] [0-9][A-Z][0-9]
```

- Number (North American decimal separators required):

```
^-?[0-9]{1,3}(,[0-9]{3})*(\.\.[0-9]{1,2})?[$
```

- Phone Number (10 digit North American, with some formatting options):

```
\(?[0-9]{3}\)?[-.]?[0-9]{3}[-. ]?[0-9]{4}
```

# Text Validation

Input validation that blocks illegal inputs (via pcType) and performs a final check, including colour highlighting (via pcFinal):

```
val textInput = TextField().apply {
    promptText = "e.g., A1A 1A1"
    textFormatter = TextFormatter<String> {
        it.text = it.text.uppercase()
        val pcType = Regex("[A-Z]?[0-9]?[A-Z]? ?[0-9]?[A-Z]?[0-9]?")
        val pcFinal = Regex("[A-Z][0-9][A-Z] [0-9][A-Z][0-9]")
        if (pcType.matches(it.controlNewText)) {
            background = Background(BackgroundFill(
                if (pcFinal.matches(it.controlNewText))
                    Color.GREEN else Color.YELLOW, null, null))
            it
        } else {
            null
        }
    }
}
```

— ⌂ X

U CS 349

# Positional Input

# Properties of Positional Input Devices

## Displacement vs. Force Control

- Mouse = displacement
- (Analog) joystick = force



# Properties of Positional Input Devices

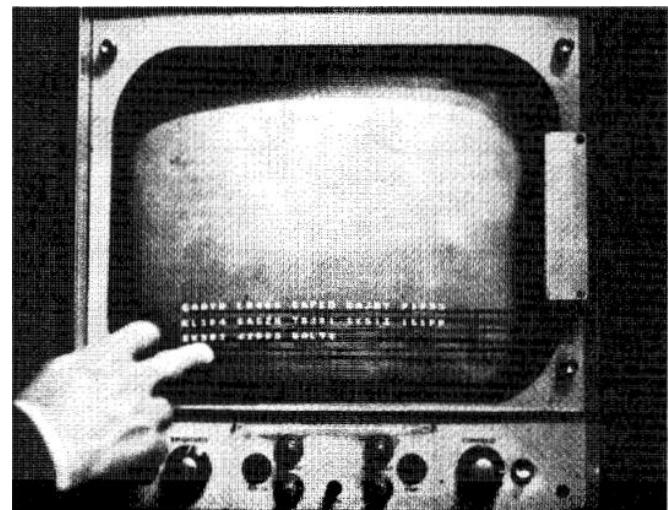
## Absolute vs. Relative Positioning

- Mouse = relative
- Touchscreen = absolute



## Direct vs. Indirect Contact

- Mouse: indirect
- Touchscreen = direct



## Degrees of Freedom (DOF): Number of (continuous) dimensions sensed:

- Mouse: 2 (x,y), 3 (x,y,scroll)
- Touchscreen: 2 (x,y), 3 (x,y,force)
- Joysticks: 2+ (x,y,z,w, ...)

# Displacement vs. Force Sensing

Isotonic devices measure displacement, e.g.,

- Mouse: optical sensor (x,y)
- Mouse: scroll wheel (scroll)



Isometric devices measure force, e.g.,

- Mouse: scroll “bar”, scroll “point” (scroll)
- Gamepad: left, right sticks



# Direct vs. Indirect Input

Indirect: input position is controlled by a cursor which is controlled by a device away from the display



Direct: input position is controlled by direct contact with the corresponding display position



# Absolute vs. Relative Position

Absolute position is a direct mapping of input device position to a display input position



Relative position maps changes in input device position to changes in cursor position



# Direct vs. Indirect vs. Absolute vs. Relative Position

Absolute



Indirect



Direct

Relative



???

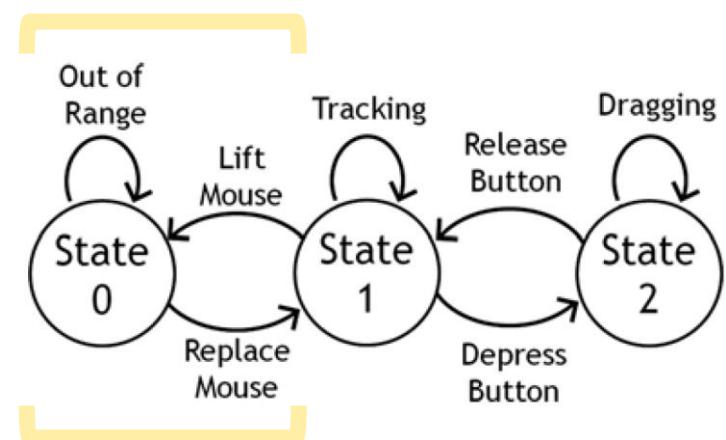
# Relative Positional Devices: Clutching

Clutching is a method to temporarily disconnect an input device from controlling cursor position

- E.g., lifting a mouse and moving it to another location

Relative positional movements will cause the device to drift and either run out of space or become out of reach.

Necessary for any kind of relative position control device



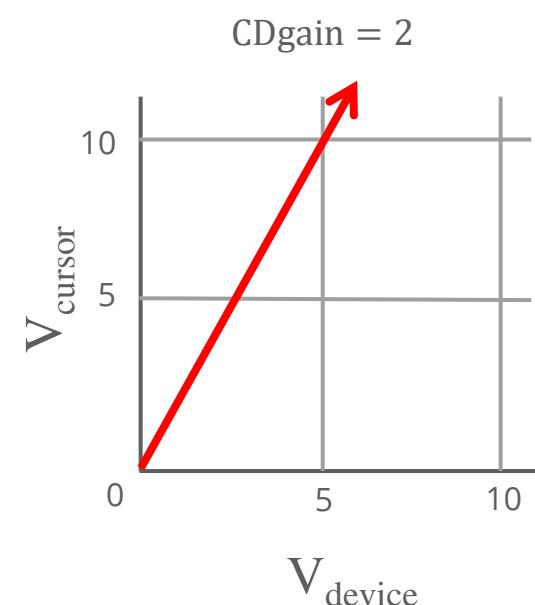
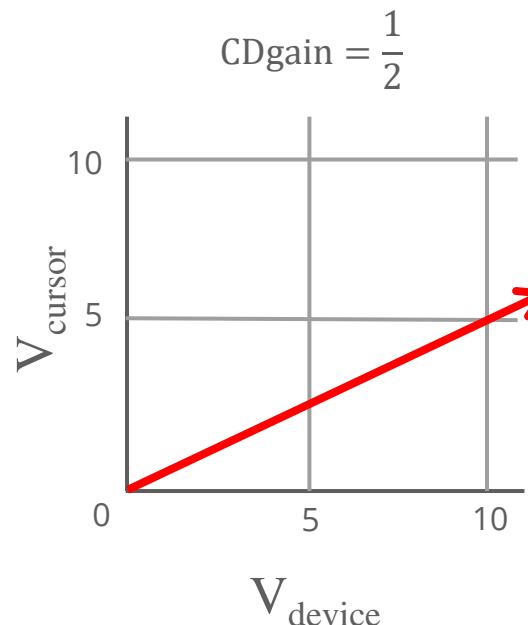
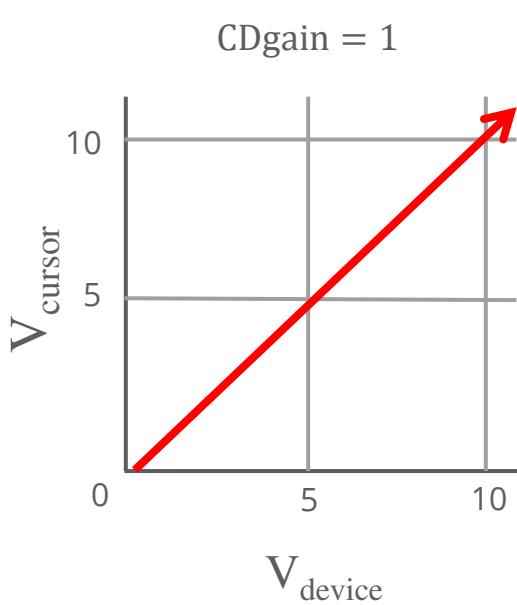
clutching

# Relative Positional Devices: Control-Display Gain

Control-Display Gain (or CD Gain) is the ratio of **display cursor** movement to **device control** movement

- The ratio is a scale factor (i.e., “gain”)
- Works for relative devices only

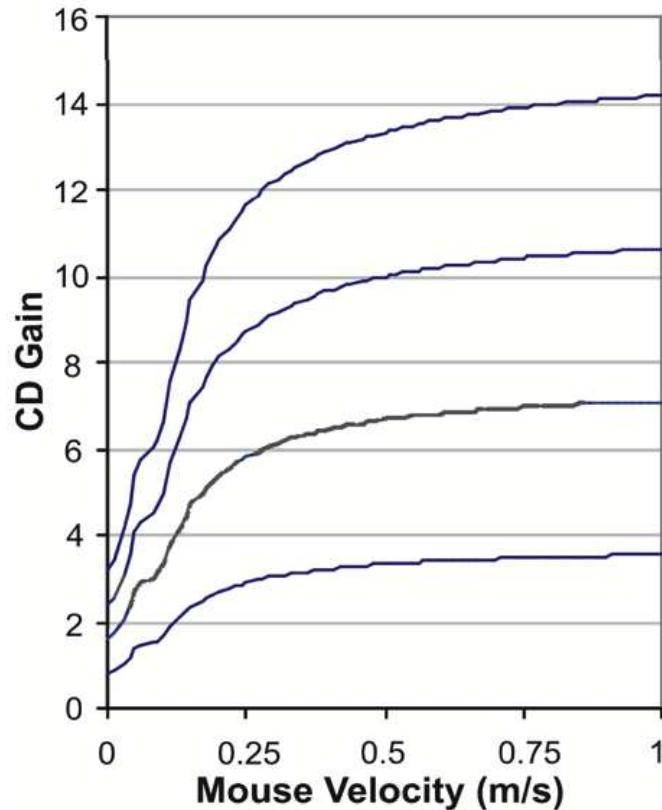
$$CDgain = \frac{v_{cursor}}{v_{device}}$$



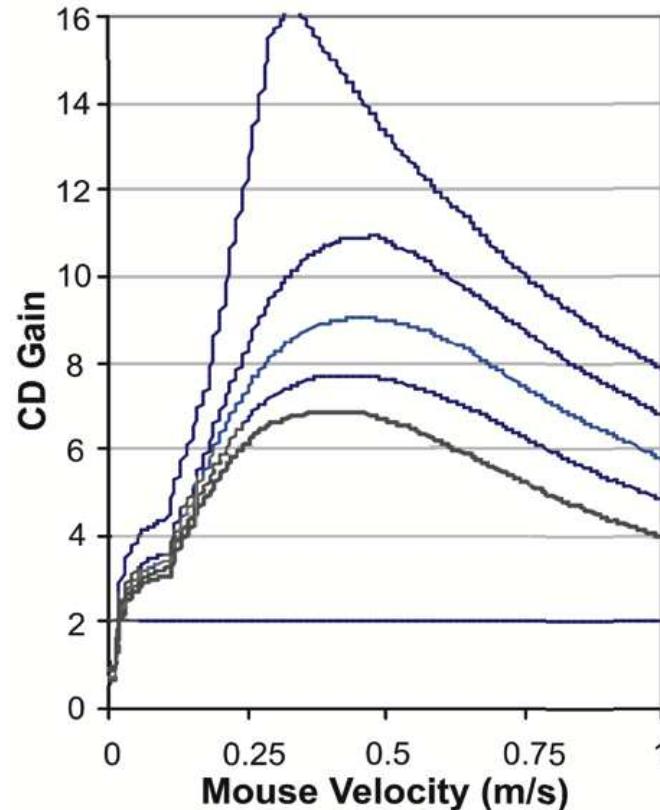
# Relative Positional Devices: Control-Display Gain

Dynamically change CD Gain based on device velocity

- can reduce the amount of clutching



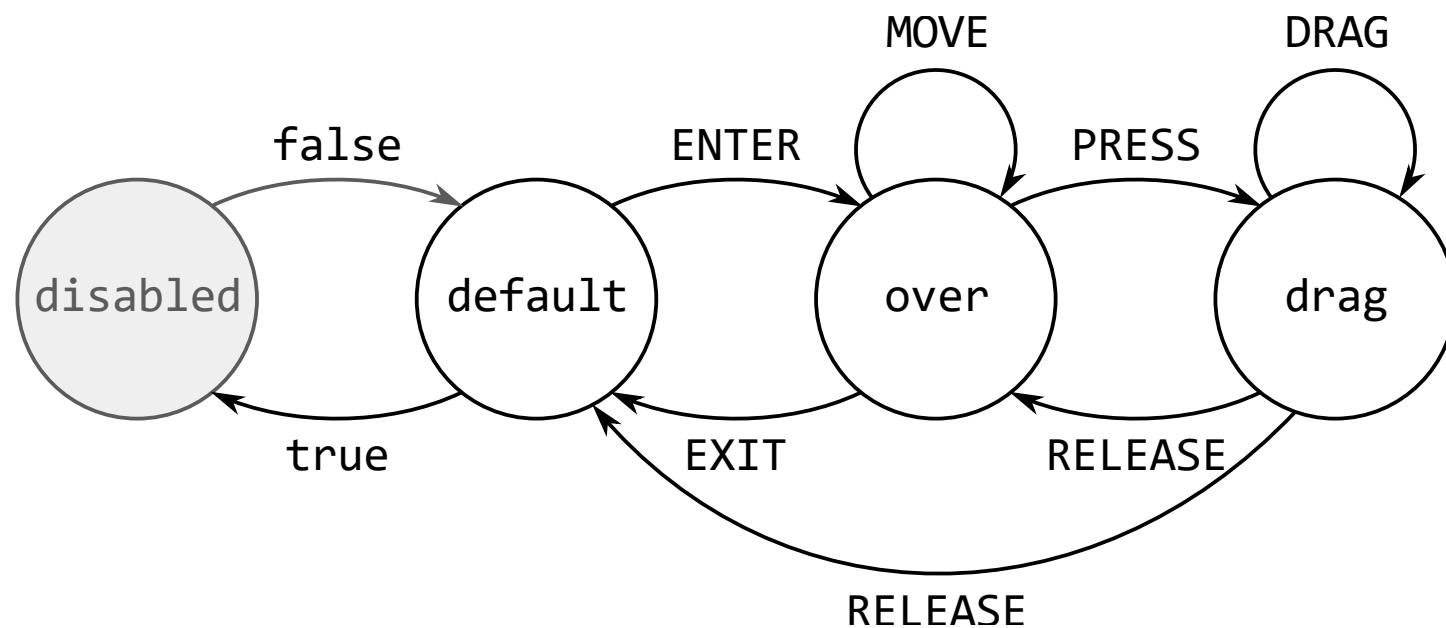
(b) Windows XP/Vista



(c) Mac OSX

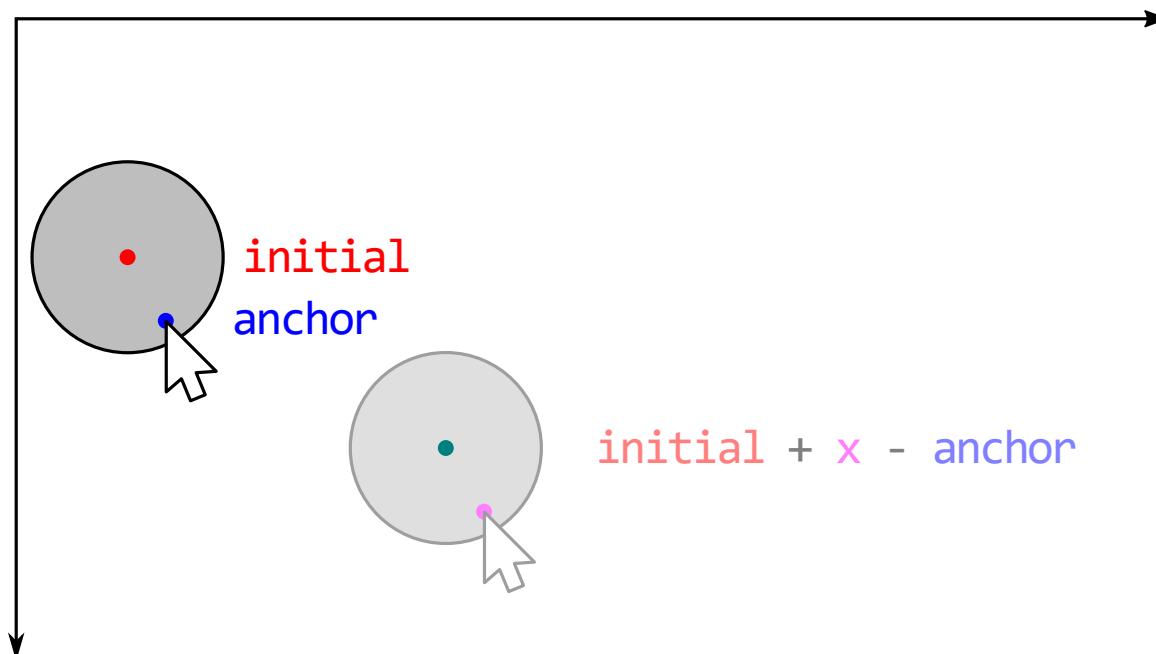
# Widget States for Positional Input

- default
- over (hovering)
- down (dragging)
- disabled



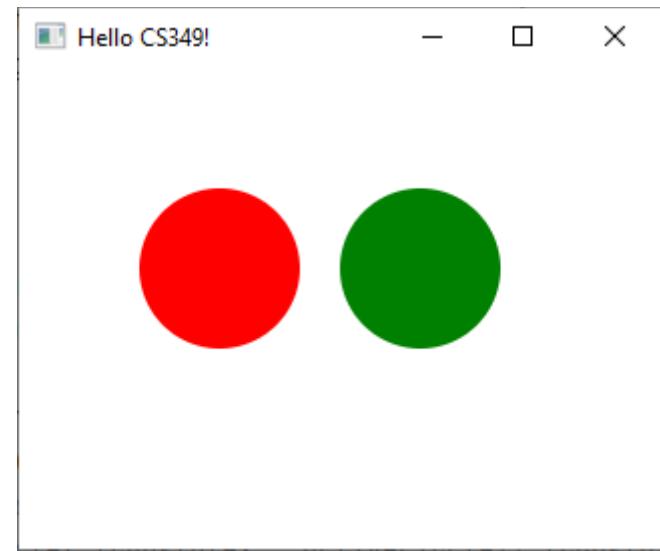
# Implementing Dragging

```
data class DragInfo(var target : Circle? = null,  
                    var anchorX: Double = 0.0,  
                    var anchorY: Double = 0.0,  
                    var initialX: Double = 0.0,  
                    var initialY: Double = 0.0)  
  
var dragInfo = DragInfo()
```



# Implementing Dragging

```
val makeCircle = { x: Double, y: Double, r: Double, col: Color ->
    Circle(x, y, r, col).apply {
        addEventFilter(MouseEvent.MOUSE_PRESSED) {
            dragInfo = DragInfo(this, it.sceneX, it.sceneY,
                translateX, translateY)
            viewOrder -= 1000.0
        }
        addEventFilter(MouseEvent.MOUSE_DRAGGED) {
            translateX = dragInfo.initialX + it.sceneX - dragInfo.anchorX
            translateY = dragInfo.initialY + it.sceneY - dragInfo.anchorY
        }
        addEventFilter(MouseEvent.MOUSE_RELEASED) {
            dragInfo = DragInfo()
            viewOrder += 1000.0
        }
    }
}
```



# Implementing Dragging

Here is an example on how to use the bubble phase of the parent to modify its child after it has processed an event:

```
val circ1 = makeCircle(100.0, 100.0, 40.0, Color.RED)
val circ2 = makeCircle(200.0, 100.0, 40.0, Color.GREEN)

val pane = Pane(circ1, circ2).apply {
    addEventHandler(MouseEvent.MOUSE_DRAGGED) {
        dragInfo.target?.translateX = dragInfo.target!!.translateX -
            (dragInfo.target!!.translateX % 10)
        dragInfo.target?.translateY = dragInfo.target!!.translateY -
            (dragInfo.target!!.translateY % 10)
    }
}
```

# End of the Chapter



Any further questions?

-



X

# Mobile Device User Interfaces

# U

CS 349

March 7

# Mobile Design Implications

More Limited resources (memory, storage, battery)

- more reliance on cloud for storage, processing

Single application model (one foreground application)

- managing state very important
- cannot easily multi-task

Small screen (different sizes, orientations such as landscape)

- layout challenges

Touch input

- multiple fingers great, but less precise
- often used one-handed

→ Big implications for UI programming



Design is about constraints (things you have to do and things you can't do) and tradeoffs (the less-than-ideal choices you make to live within the constraints)."

- Steve Krug

# Touch Input

Advantages?

- **Absolute + Direct** input (& Direct Manipulation)
- Tracks touch and movement on the screen
  - Up to 10 points of contact simultaneously!
- We can *sort-of* detect finger pressure (Apple 3D touch).
- We can combine widgets with (multi-figure) gestures.



Disadvantages?

- Your finger is not a high precision input device (e.g., try drawing)
- We cannot track which finger is being used, which limits input
- User activity (e.g., running, walking, sitting) affects data reliability.



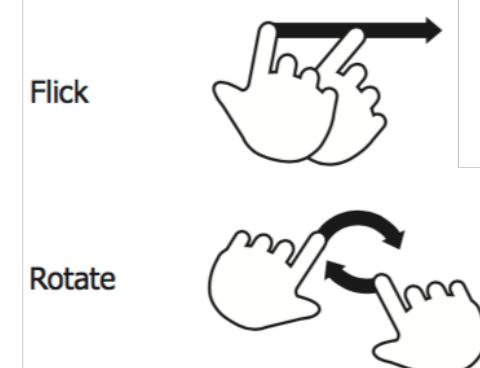
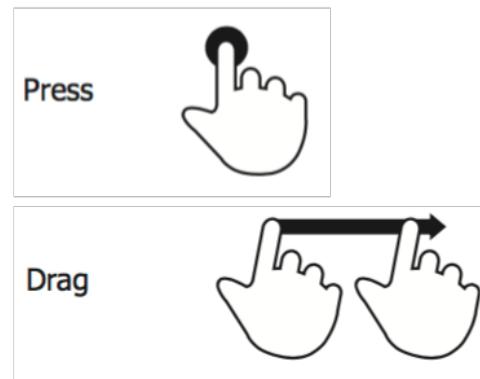
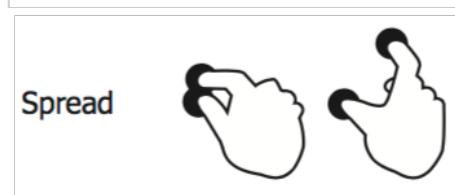
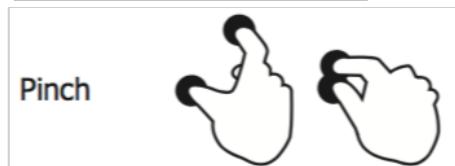
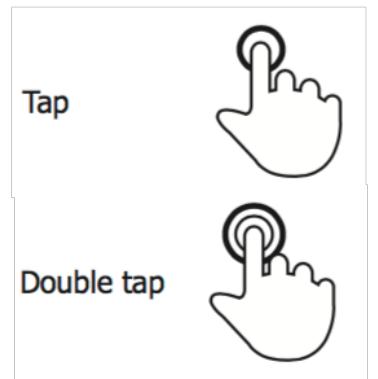
Jeff Han's Seminal Ted Talk on Multitouch Interaction (Feb 2006)

- [https://www.ted.com/talks/jeff\\_han\\_the\\_radical\\_promise\\_of\\_the\\_multi\\_touch\\_interface](https://www.ted.com/talks/jeff_han_the_radical_promise_of_the_multi_touch_interface)

# Touch Gestures

Taps and swipes with 1, 2, or more fingers

Uses various characteristics of low-level touch screen events, e.g., position, time, number of contacts



Long hold



Tap



Double Tap



Tap



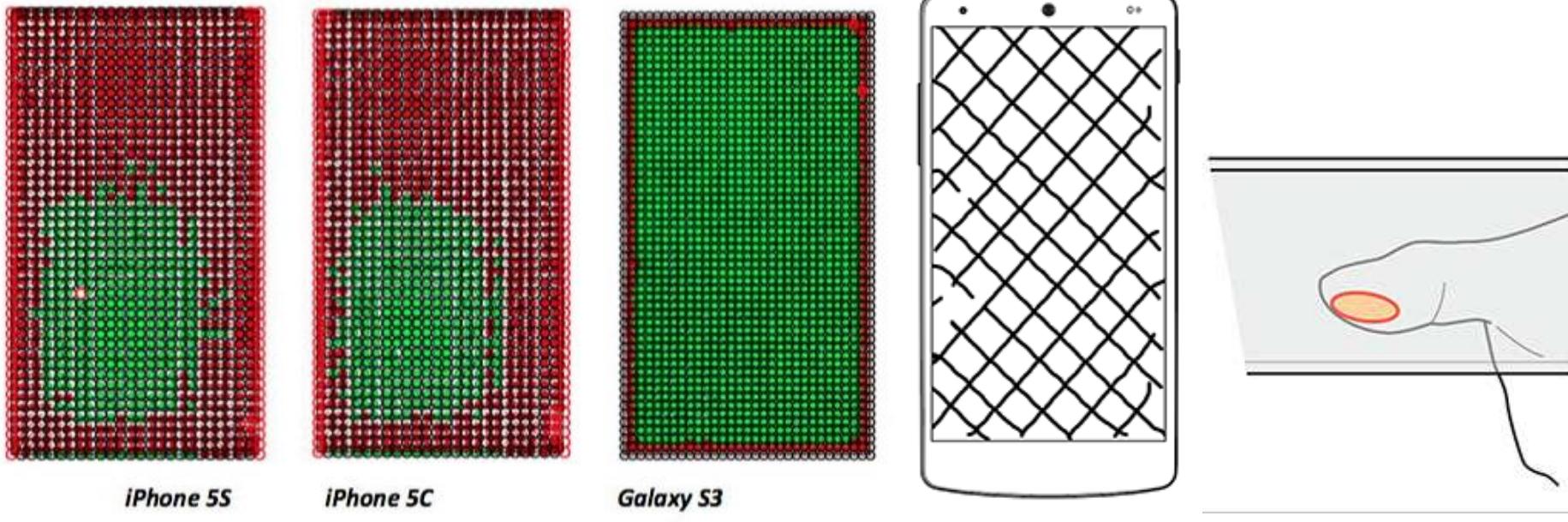
Double Tap



# Touch Sensing Accuracy

Touch screen input is noisy

- Sensors vary in their accuracy
- Estimates for “pressure” very noisy
- Large input (finger) relative to target size

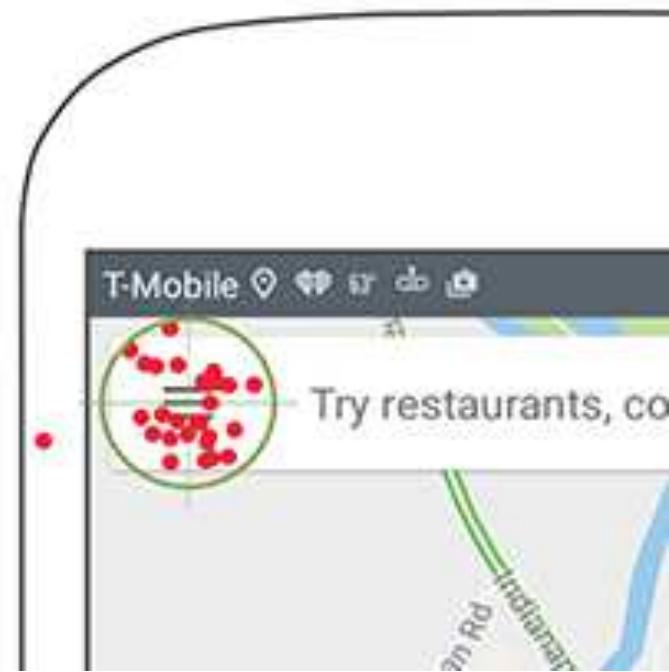
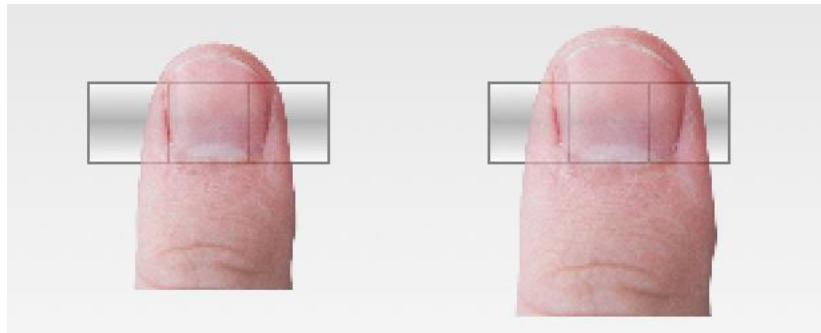


# Challenge: Human Accuracy

“Fat fingers” lead to occlusion and precision issues

Touch targets need to be large

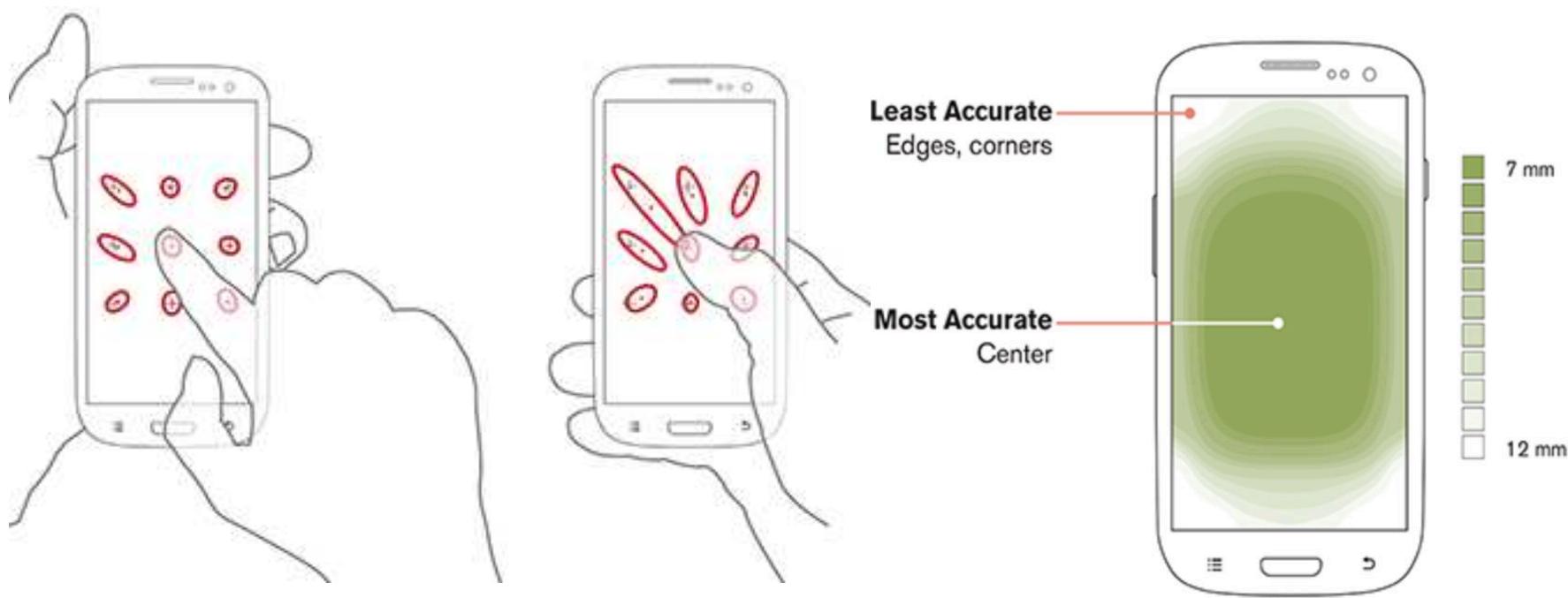
- Apple / iOS recommends 15 mm
- Google / Android recommends 9 mm (min 7 mm, min 2 mm apart)



# Challenge: Human Accuracy Varies By Position and Grip

Accuracy affected by

- Hand posture (i.e., which hand is holding, which used to interact)
- Finger vs. thumb interaction
- User activity: stationary vs. moving

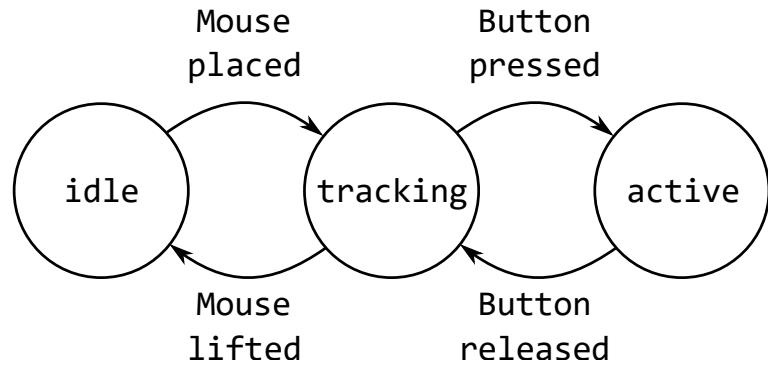


# Challenge: No Hover State in Touch

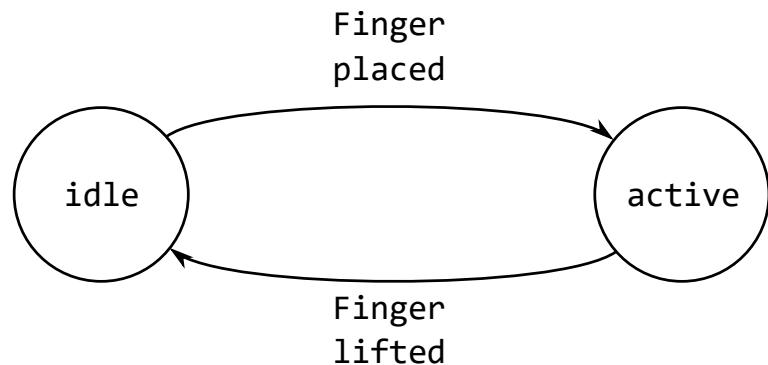
Having a middle “tracking” input state allows for hover (e.g., mouse). Users can preview an action before committing

- Mouse input typically supports 3-states (i.e., mouse off table, moving mouse cursor, dragging mouse)
- Touch input only supports 2-states (i.e., no touch, touch).

mouse input states



touch input states



“Imprecision, Inaccuracy, and Frustration: The Tale of Touch Input” by Hinckley and Wigdor

# Challenge: Multi-touch Dispatch Ambiguity

In multi-touch, multiple fingers may hit a control simultaneously, leading to ambiguity

When is click event generated? There are a number of possibilities:

- Interaction is invoked only when the last finger is lifted from the control (TOUCH\_RELEASED), or
- Interaction is invoked every time a finger presses a control (TOUCH\_PRESSED), even if another finger still makes contact, or
- over-capture: multi-touch controls captured by more than 1 contact simultaneously (e.g., selecting the thumb of a slider with two fingers can mean that it will not track directly under a single finger when moved.)

# Challenge: Physical Constraints

Touch input relies on the principle of direct manipulation, i.e., user places their fingers onto an object, moves their fingers, and the object changes its position, orientation and size to maintain the contact points.

Direct touch breaks when movement constraints are reached (e.g., moving beyond bounds, scrolling past limits). This breaks immersion, and the sense of working with a physical object.

Solution: Elastic effects that mimic physical responses (e.g., Apple iPhone scrolling past a list, “snaps” back)

# Device Characteristics: Interaction

Assume one app at a time:

- one app in the foreground
- most apps are suspended when not in the foreground

Each app has window that fills the entire screen:

- interaction is a sequence of different screens
- consistent navigation model is key

Do not expect users to switch between applications:

- difficult to lookup data in a different app

Controls need to be large to overcome occlusion and precision issues.  
They also need to be selectable while moving (walking or running).

# Standards: Interface Guidelines

Platform-specific design guidelines can provide specific usage examples and hints, beyond these basic guidelines

The image shows two side-by-side screenshots of platform-specific design guidelines. On the left is the Apple Human Interface Guidelines (HIG) website, featuring sections for 'New and updated' features like 'Charting data' and 'Live Activities', and a 'Featured' section. On the right is the Google 'Design for Android' page, which includes a sidebar with 'Quality guidelines' and a main content area with links to 'Material design guidelines', 'Material design components', and 'App quality guidelines'. Both pages include navigation bars at the top.

IOS Design Guidelines

<https://developer.apple.com/design/human-interface-guidelines/>

Android Design Guidelines

<https://developer.android.com/design>

# Desktop vs. Mobile

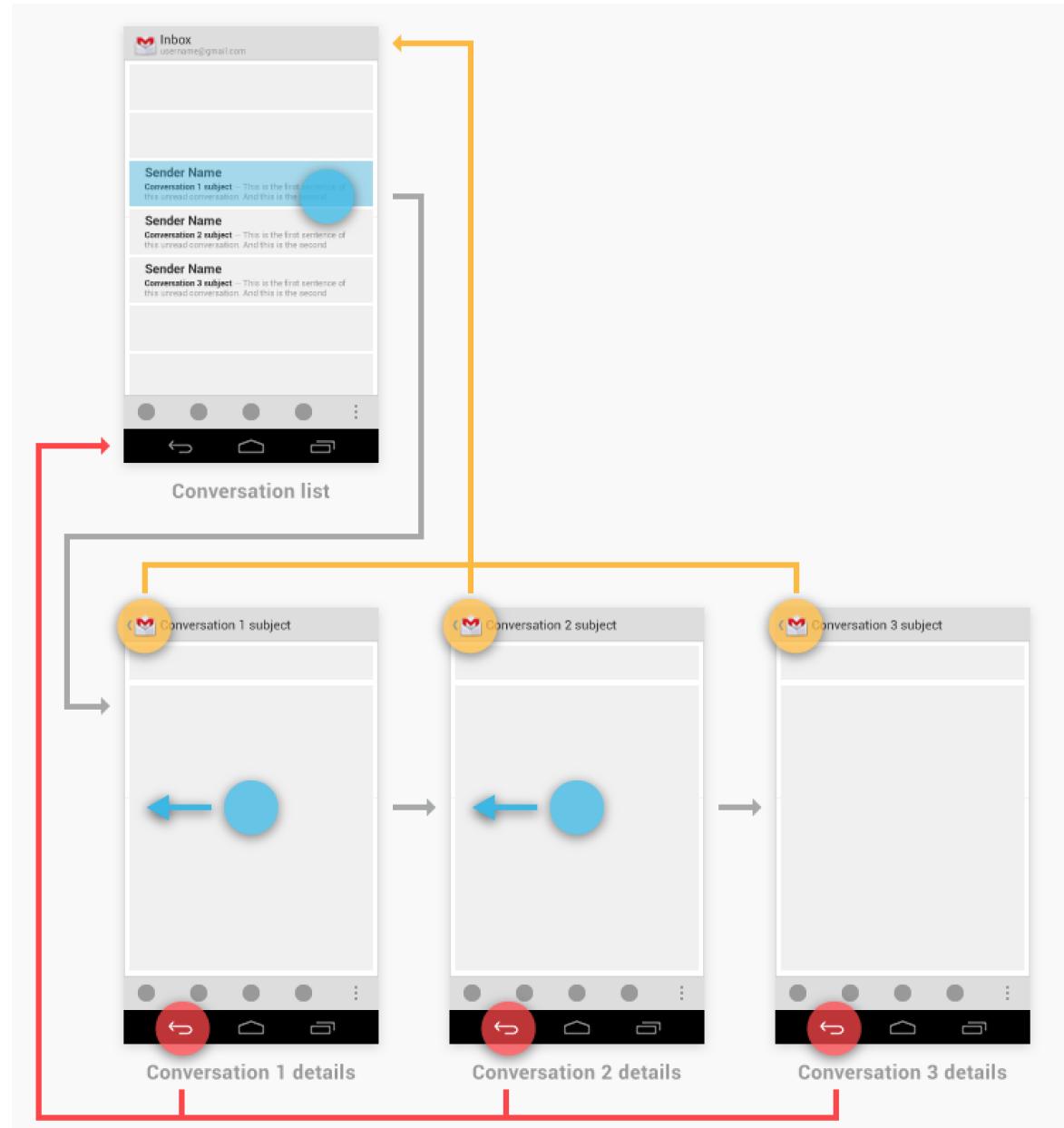
The screenshot shows the Gmail desktop interface. At the top, there's a search bar and a blue "Search" button. Below it, the "Gmail" menu is open, showing "Compose" in red. The main area displays the inbox with 7 messages. The first message is from "Google+" with the subject "Julia, a few Google+ posts you might like". The second message is from "Merced Flores" with the subject "Re: consultant for book - Hi Julia". The third message is from "Lisa Paik" with the subject "Volunteering at the Lakestone student art exhibition". The fourth message is from "Elena Casarosa" with the subject "Portrait special - We'd like to announce...". The fifth message is from "Grace Ellington" with the subject "Volunteer opportunity - I would like to...". The sixth message is from "Henri Rousseau" with the subject "Niagra falls pictures - Julia, Here's a few photos I took...". The seventh message is from "Olenna Mason" with the subject "Lakestone student art exhibition". On the left sidebar, under "Circles", there are sections for "Notes", "Personal", "Travel", and "More". At the bottom, there are sections for "Julia" and "Henri Rousseau", along with icons for "Compose", "Reply", and "Call".

The screenshot shows the Gmail mobile interface. At the top, there's a red header bar with the word "Primary" in white. Below it, there are three tabs: "Social" (with 1 new), "Promotions" (with 2 new), and "Updates" (with 1 new). The main area displays the inbox with 7 messages. The first message is from "Google+" with the subject "Julia, a few Google+ posts you might like". The second message is from "Merced Flores" with the subject "Re: consultant for book - Hi Julia". The third message is from "Lisa Paik" with the subject "Volunteering at the Lakestone student art exhibition". The fourth message is from "Elena Casarosa" with the subject "Portrait special - We'd like to announce...". The fifth message is from "Grace Ellington" with the subject "Volunteer opportunity - I would like to...". The sixth message is from "Henri Rousseau" with the subject "Niagra falls pictures - Julia, Here's a few photos I took...". The seventh message is from "Olenna Mason" with the subject "Lakestone student art exhibition". At the bottom right, there's a red circle with a white pencil icon, indicating a new edit or reply action.

# Navigation

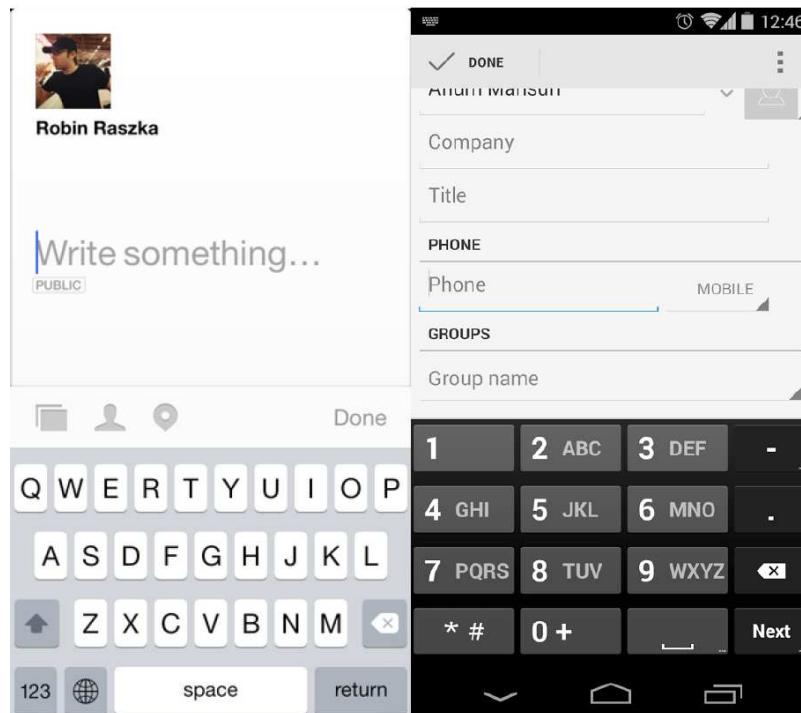
Critical due to ordering of screens in most applications.

- up
- back
- gestures

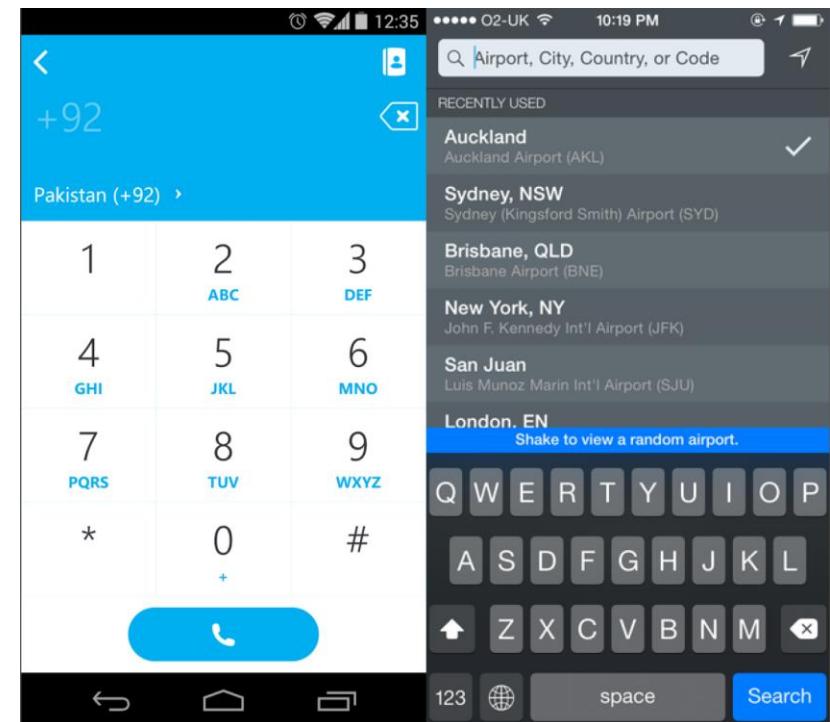


# Help Users to Enter Information

Provide the right data entry tool



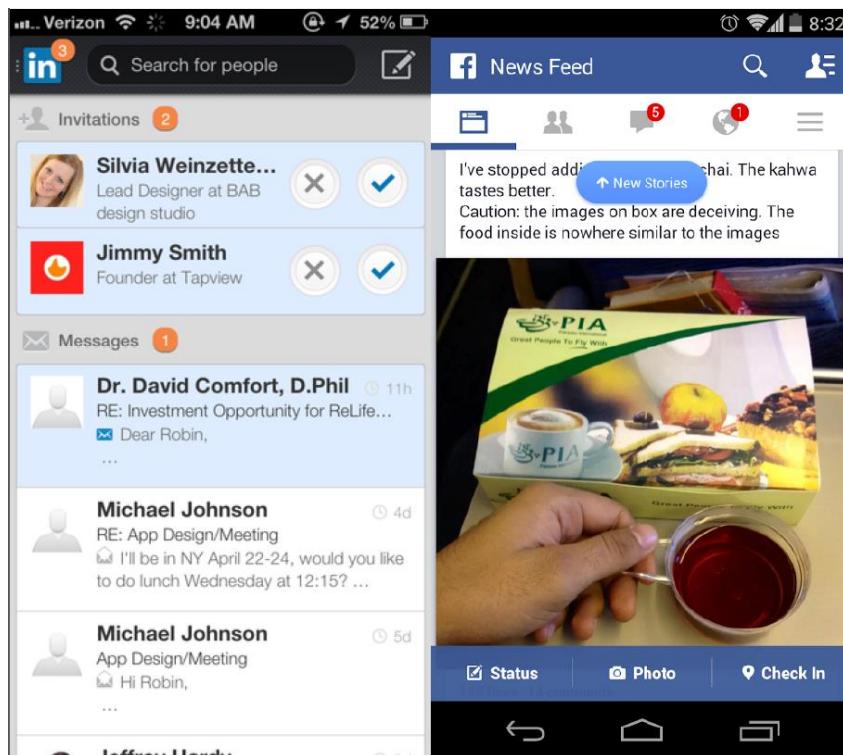
Anticipate and predict input



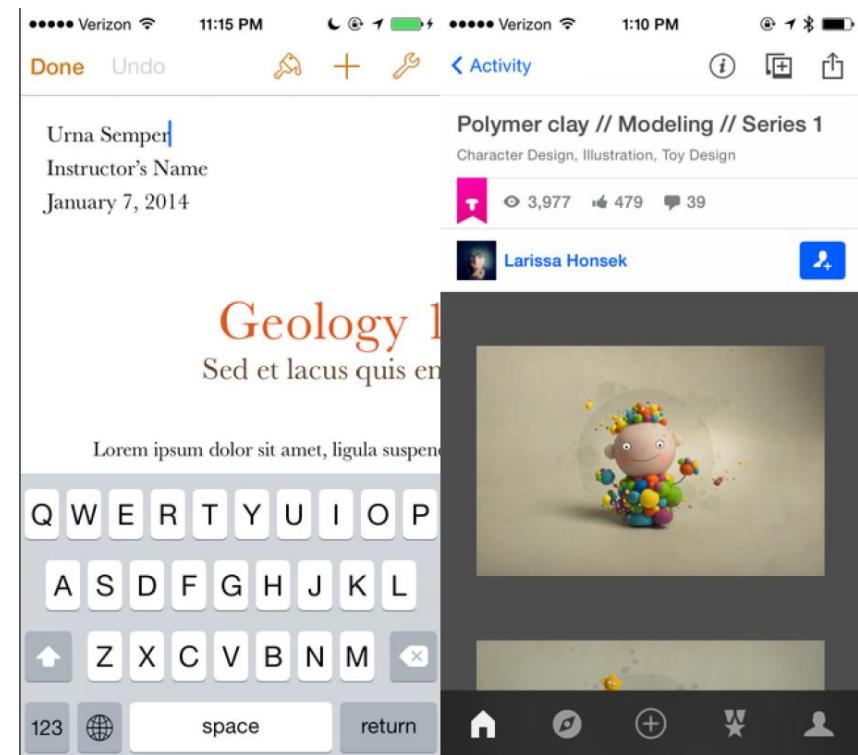
“Mobile UI Design Pattern” (Bank and Zuberi)

# Help Users Find Correct Actions

Highlight new content

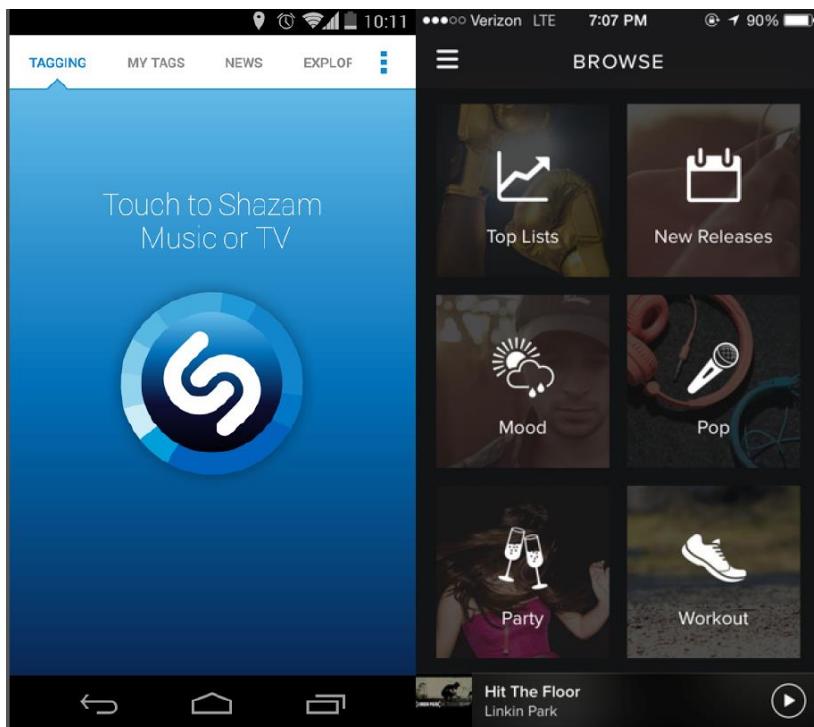


Quick access to frequent actions

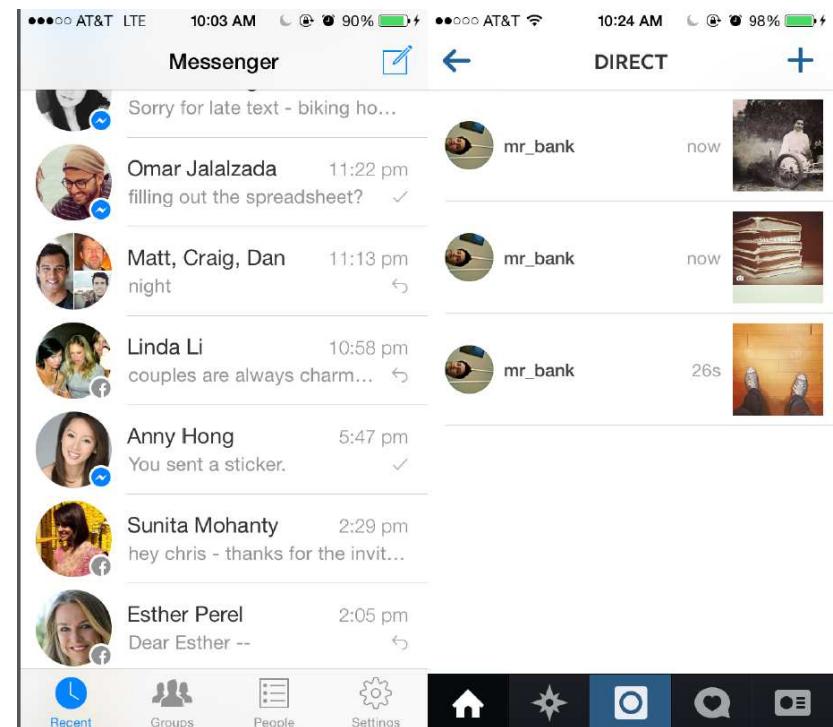


# Help Users Find Correct Actions

Make actions obvious

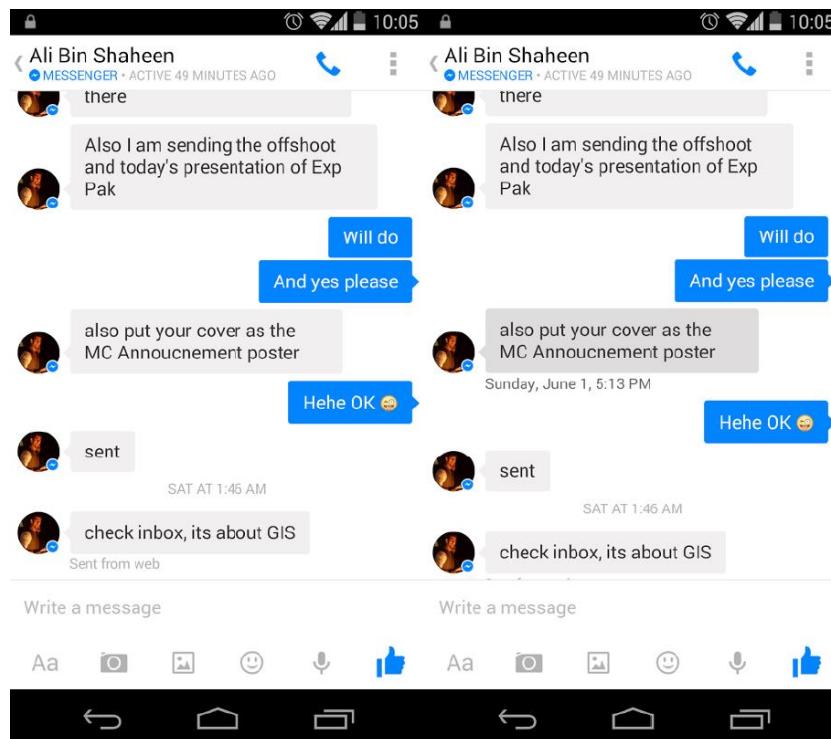


Distinguish between controls and content

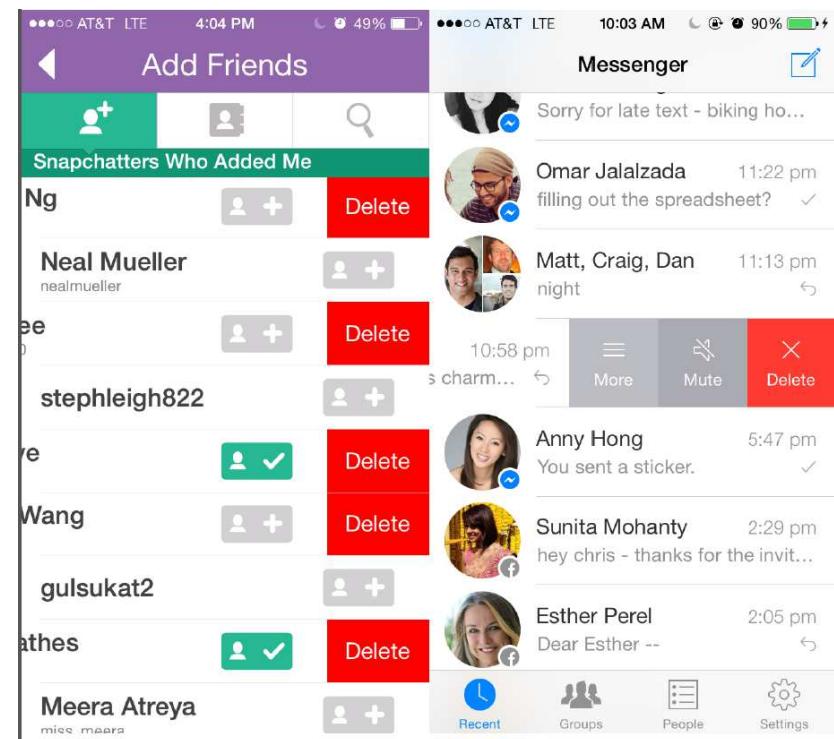


# Avoid Clutter

Hide meta-data



Hide secondary menus



# End of the Chapter



Any further questions?

-



X

# Android

## Development Fundamentals

### Layouts

### Widgets

# U

CS 349

# March 15



U

CS 349

# Development Fundamentals



# Why Android?

## Pervasive Mobile Platform

- World's most popular & frequently installed mobile OS
- Runs hundreds of millions of phones, tablets, watches, cars, ...

## Developer Friendly

- State-of the art (Kotlin) language (and Java legacy language)
- Multi-platform development support
- Open Source (by minimum-definition)

## Exposure to different UI development paradigms

- “Android Views”: *imperative* using code to build interface
- “Android Views”: *declarative* using XML description of interface
- Jetpack Compose: *declarative* using composable functions
- MVVM architecture (variation of MVC)

# Android Architecture

Android is based on Linux .

Every app has distinct user profile

- App permissions restrict access (resources, data)

Every app runs in its own process

- Apps must request access to shared resources (e.g., file system, camera)

System manages app “lifecycle”

- including terminating apps that have not been used in a while

# Development Process

Apps written in Kotlin (or Java or C++) using an Android SDK

- Applications include code + resources (media files, layouts)
- SDK libraries provide device capabilities to your source code.
- XML *manifest* file describes the application and contains information that the OS needs to install and run your application.

Compiler generates an Android Package file (APK)

- Includes manifest, code, resources, etc.

APK can be installed on a physical Android phone (with developer mode enabled), or through an online App Store.

APK can also run on an Android Virtual Device (AVD) for testing/debugging.

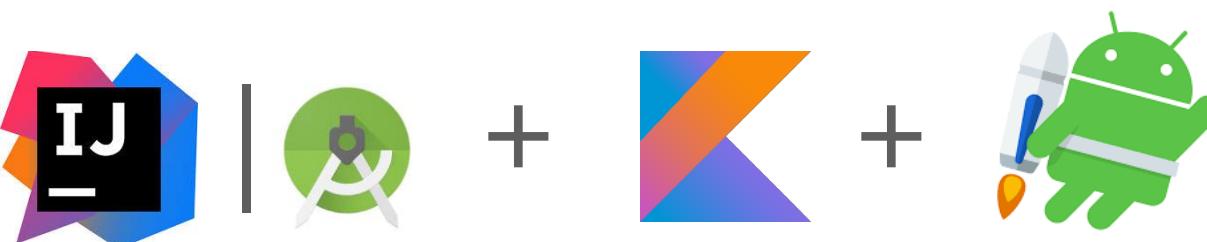
# Development Environment

IDE:

- IntelliJ (latest version), Kotlin, Java (11.0.17), and Gradle (7.5, or whatever is auto-installed)
- Android SDK: Android Tiramisu – API Level 33

Android Device Emulator (AVD)

- Pixel 4a API 33

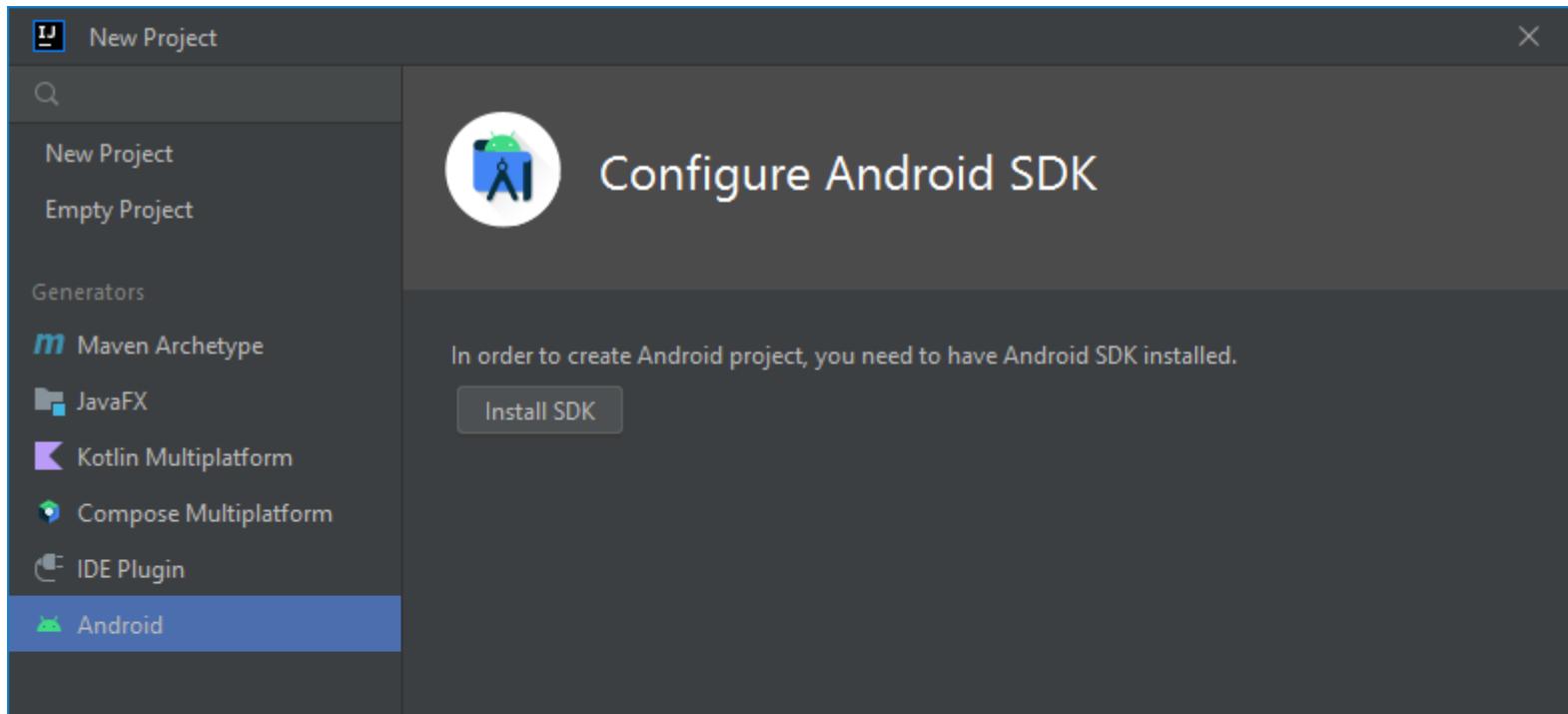


# Environment Setup

IntelliJ already lists “Android” as new project type

First time an Android project is created, it walks through installing the Android SDK and an Android Device Emulator (AVD). Follow these steps using SDK and AVD versions on previous slide:

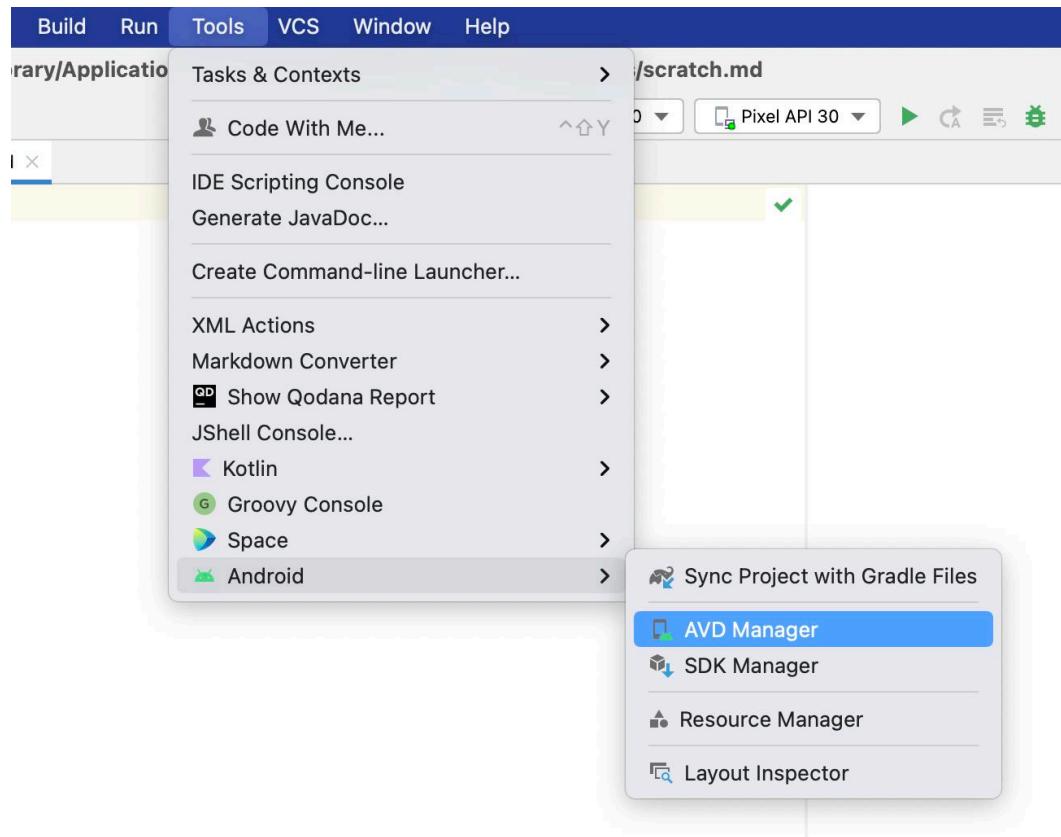
<https://www.jetbrains.com/help/idea/create-your-first-android-application.html>



# AVD and SDK Manager

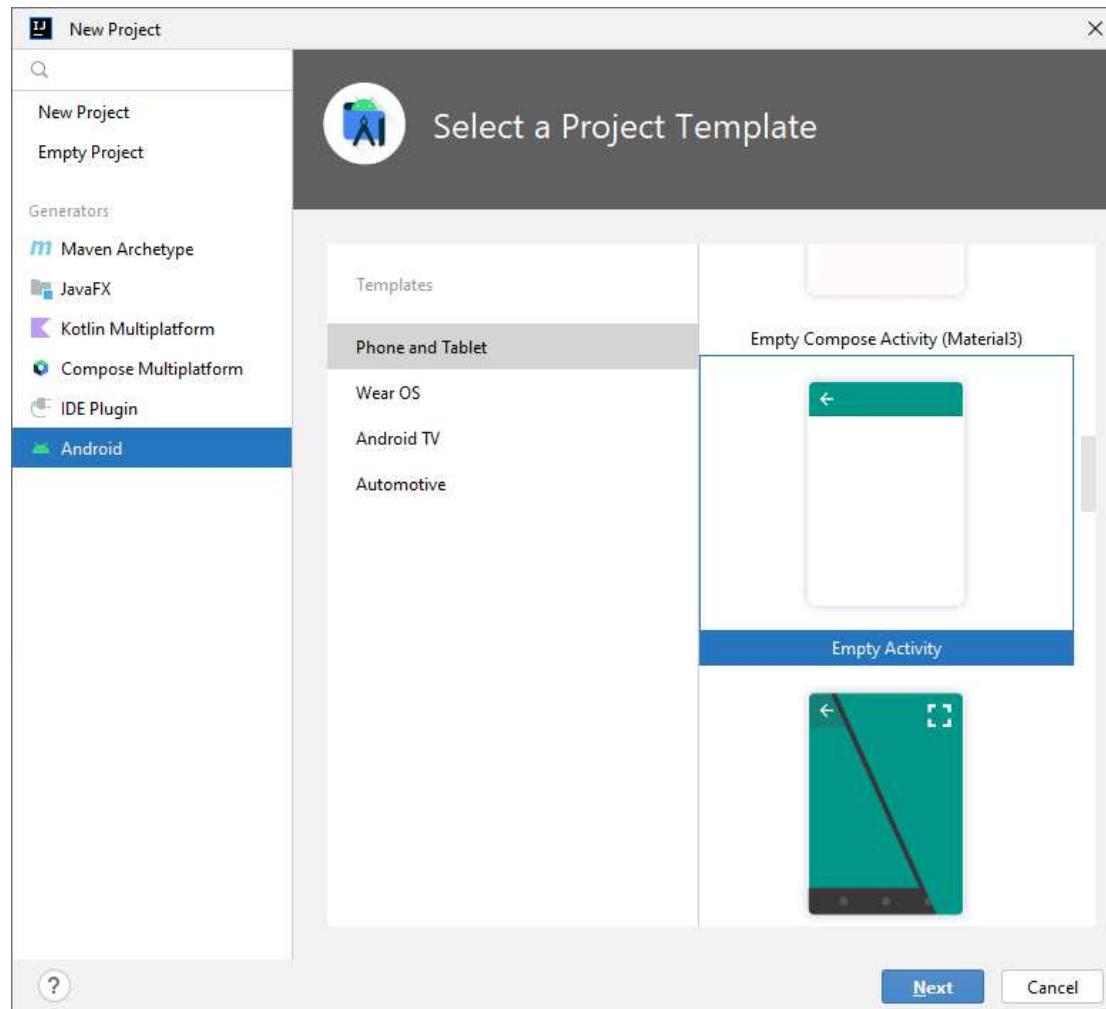
In IntelliJ “Tools/Android” menu:

- AVD Manager: add and manage Android Virtual Devices (AVDs)
- SDK Manager: add and maintain Android SDK and tools



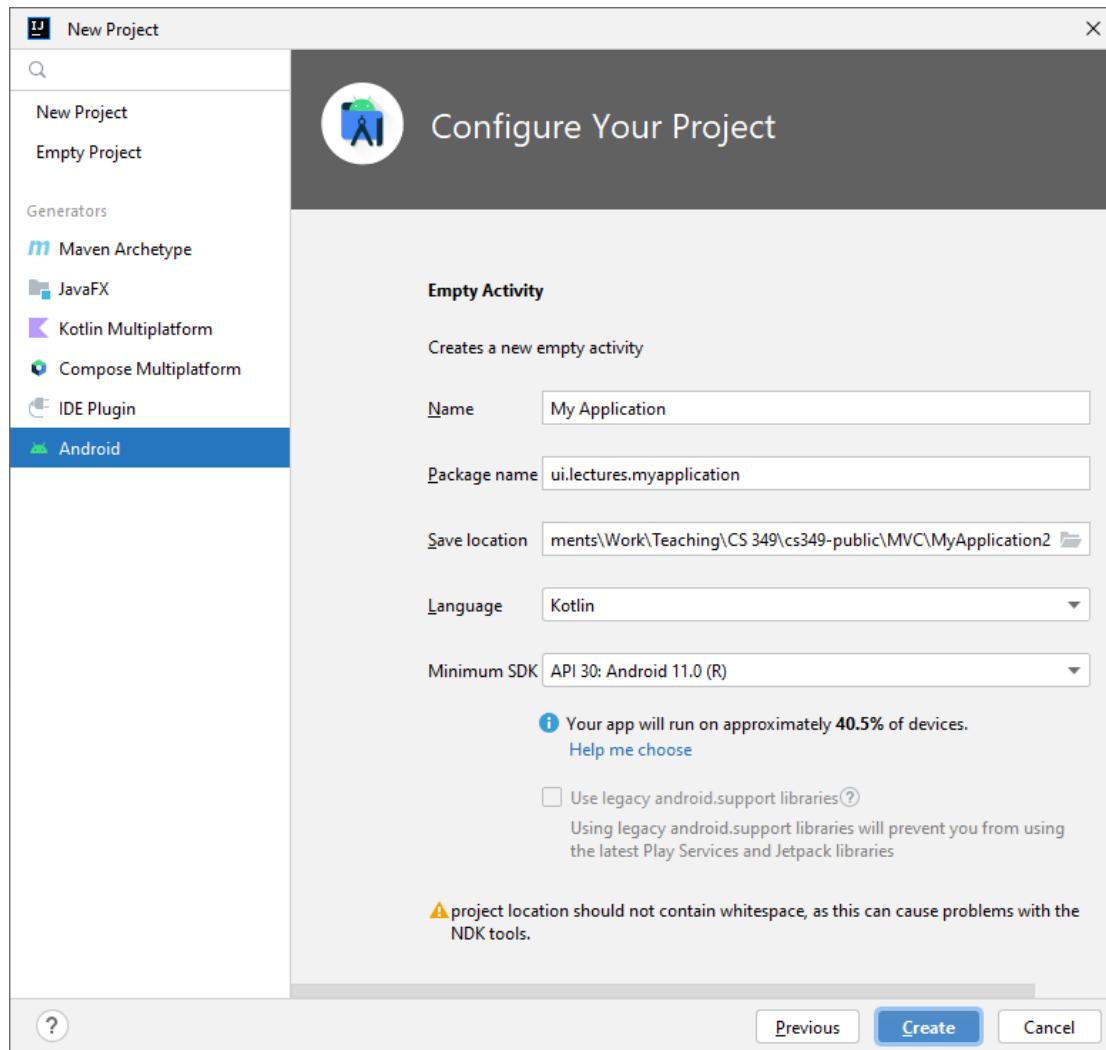
# Basic Project Walkthrough: New Android Project

Pick “Empty Activity” project template



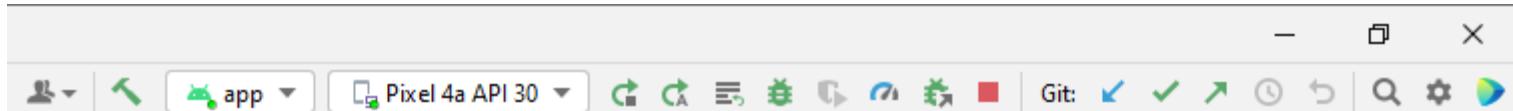
# Basic Project Walkthrough: Configure

Use Kotlin and Minimum SDK API 30: Android 11.0 (R)

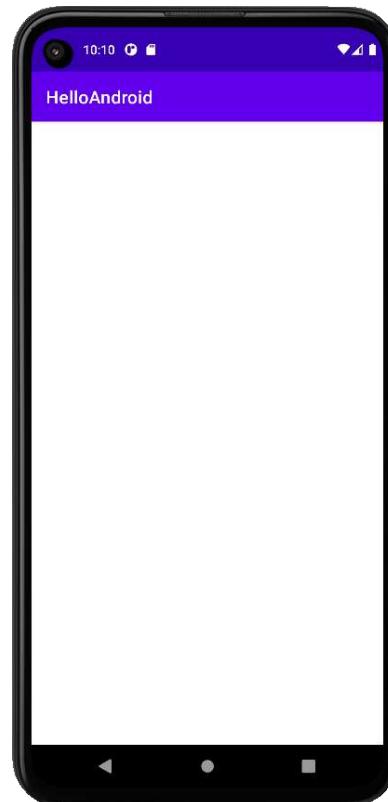


# Run The App

Verify the right Android SDK and AVD Version is selected then compile and run the default app:



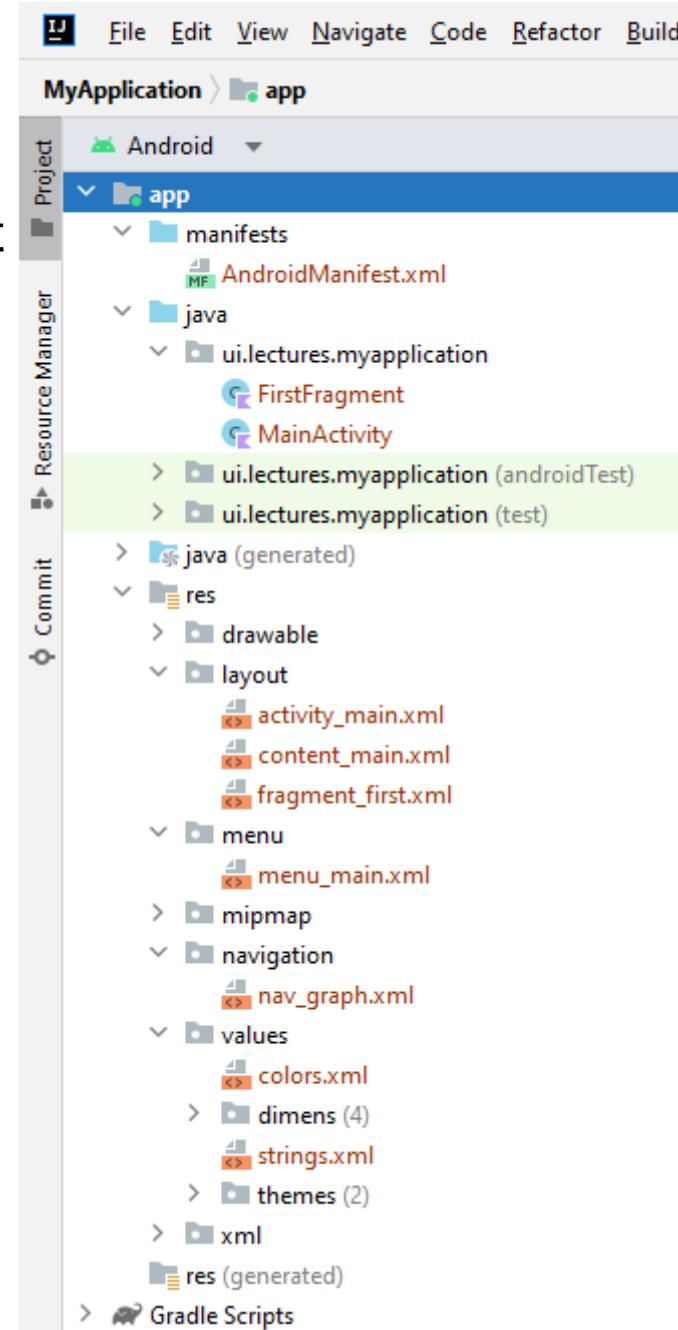
After compiling, the AVD will boot and the app will load:



# Android Project Structure

Select “Android” in Project Tab dropdown to get more optimized project view

- **Manifest** (app/manifests/): application settings (.xml)
- **Source Code** (app/java/): app behaviour (.kt)
- **Resources** (app/res/):
  - layout/: UI layout and View definitions (.xml)
  - navigation/: Navigation between activities and fragments (.xml)
  - values/: constants, e.g., strings, colours, colour schemes, ... (.xml)
  - mipmap/: raster images (.webp, .jpg, etc.)
  - drawable/: vector images (.xml, .svg, etc.)



# Manifest (AndroidManifest.xml)

Metadata about the application:

- Settings, such as, icon, name, and design theme
- Application components or “activities”
- “intent” filters: e.g., which activity to launch as main activity

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
    <application
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/Theme.MyApplication"
        tools:targetApi="33">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/Theme.MyApplication.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

# Application Components

Applications can have multiple “application components”

Each component is an entry point for a user (or the system) into some part of your application

Components	Description
Activity	Entry point for interacting with the user; a single screen with a user interface.
Service	General-purpose entry point for keeping an app running in the background
Content provider	Manages a shared set of app data that you can store in the file system
Broadcast receiver	Component that enables the system to deliver events to the app

# Activity

An **Activity** is a crucial component of an Android app and a fundamental part of the Android application model.

The Activity class creates a window to host a user interface. This window is almost always full-screen.

When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole. You implement an activity as a subclass of the Activity class.

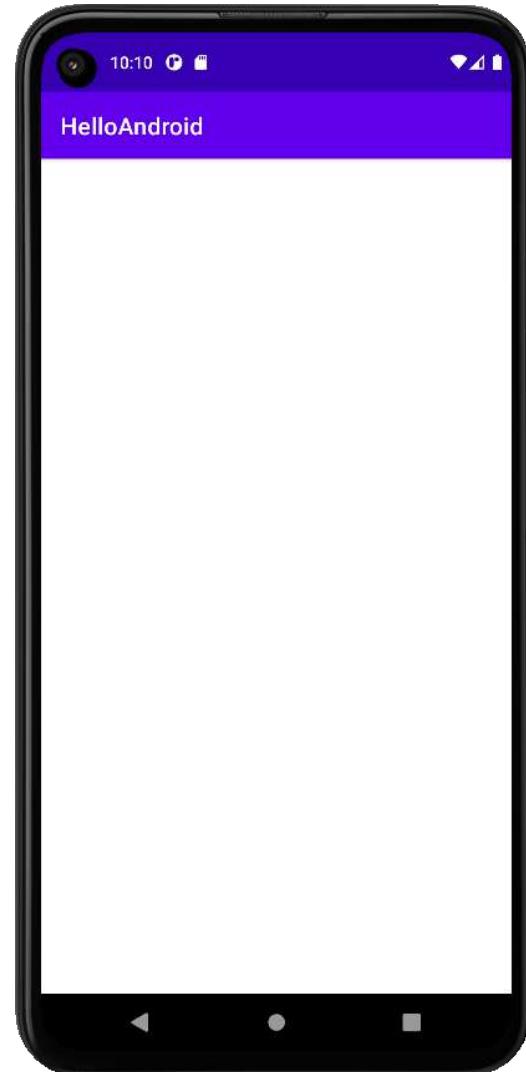


# Activity

An **Activity** is a crucial component of an Android app and a fundamental part of the Android application model.

The Activity class creates a window to host a user interface. This window is almost always full-screen.

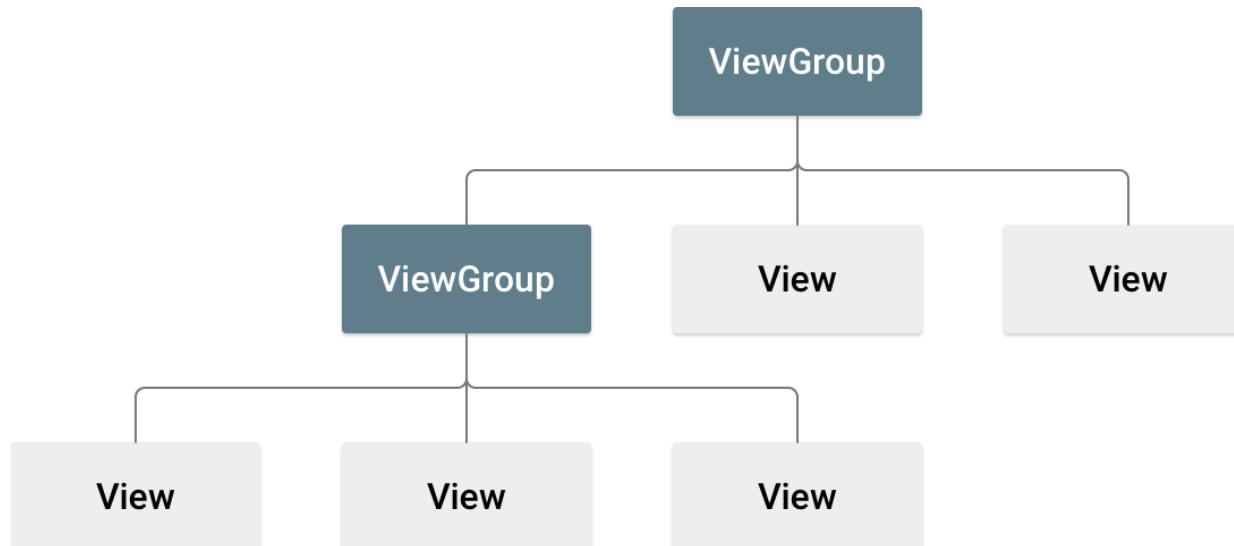
When one app invokes another, the calling app invokes an activity in the other app, rather than the app as an atomic whole. You implement an activity as a subclass of the Activity class.



# Android Scene Graph

Hierarchy of ViewGroup and View objects

- A **ViewGroup** is usually a **layout** container for child **Views** (e.g., `LinearLayout`)
- A **View** object is usually a **widget** (e.g., `Button`, `TextView`)



# Imperative UI

## MainActivity.kt

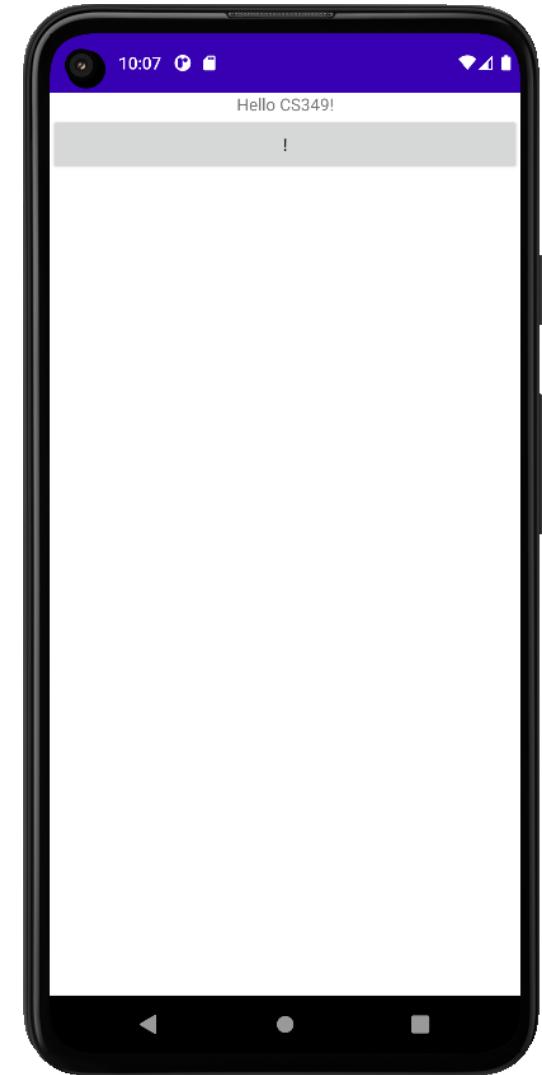
```
class MainActivity : FragmentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val layout = LinearLayout(this).apply {
            orientation = LinearLayout.VERTICAL
        }

        val myText = TextView(this).apply {
            text = "Hello CS349!"
            textAlign = View.TEXT_ALIGNMENT_CENTER
            layout.addView(this)
        }

        val button = Button(this).apply {
            text = "!"
            setOnClickListener {
                myText.text = "${myText.text}!"
            }
            layout.addView(this)
        }

        setContentView(layout)
    }
}
```



# Declarative UI

## activity\_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <TextView android:id="@+id/myText"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/greeting"/>

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/exclabutton"
        android:onClick="addExclamation"/>

</LinearLayout>
```

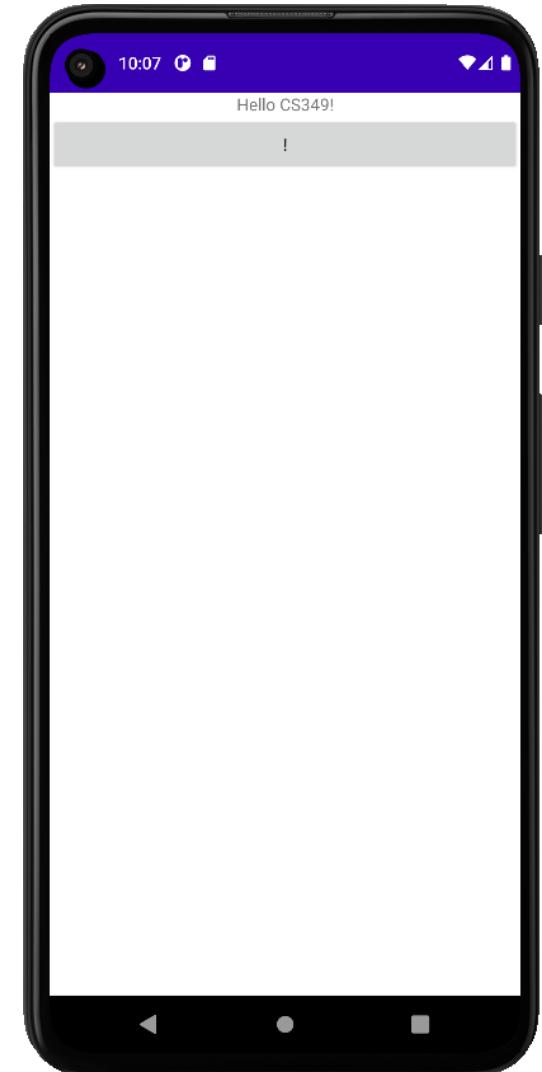
# Declarative UI

## MainActivity.kt

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
    }  
  
    fun addExclamation(view: View) {  
        val text = findViewById<TextView>(R.id.myText)  
        text.text = "${text.text}!"  
  
    }  
}
```

## strings.xml

```
<resources>  
    <string name="app_name">HelloAndroid</string>  
    <string name="greeting">Hello CS349!</string>  
    <string name="exclabutton">!</string>  
</resources>
```



# Resources and the R class

R is a static class with members generated from resources in /res

- R.layout.\* are references to xml layouts in /layouts



→ R.layout.activity\_main

- R.string.\* are refs to string elements in /values/strings.xml

```
<string name="greeting">Hello World</string>
```

→ R.string.greeting

- R.id.\* are refs to nodes in a layout with an id attribute

```
<TextView android:id="@+id/myText"
```

```
→ // the id of the view (Int)  
val viewId = R.id.myText  
// the actual view (TextView)  
val text =  
    findViewById<TextView>(R.id.myText)
```

# Example – R.values.color.xml

colors.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="math1">#FFFFBEEF</color>
    <color name="math2">#FFC60078</color>
</resources>
```

layout.xml-files:

```
<TextView android:id="@+id/myText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAlignment="center"
        android:textColor="@color/math1"
        android:text="@string/greeting"/>
```

.kt-files:

```
findViewById<TextView>(R.id.myText).apply {
    setTextColor(getColor(R.color.math2))
}
```

# Example – R.values.strings.xml

```
<resources>
    <string name="app_name">HelloCS349</string>
    <string name="greeting">Hello, %1$s %2$d!</string>
</resources>
```

## layout .xml-files:

```
<TextView android:id="@+id/myText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAlignment="center"
        android:textColor="@color/math1"
        android:text="@string/greeting"/>
```

## .kt-files:

```
findViewById<TextView>(R.id.myText).apply {
    text.text = String.format(getString(R.string.greeting), "CS", 349)
}
```

U

CS 349

# Layouts



# Android Layout Units

Device pixels-per-inch is the pixel density and measured in **dpi**

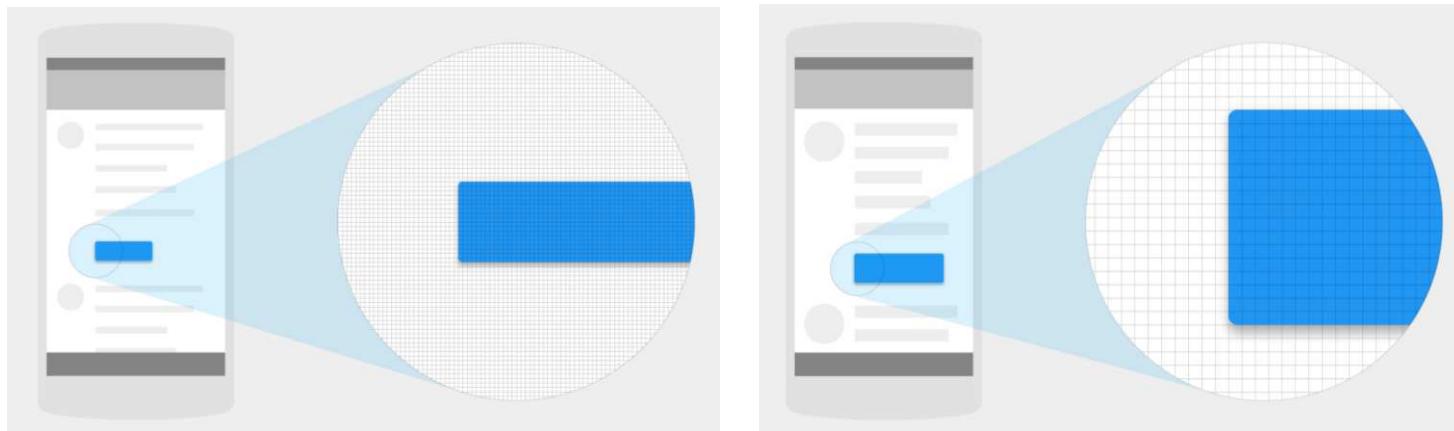
- e.g., Google Pixel 6 Phone is 441 dpi = 441 pixels-per-inch

Views typically defined in **dp** (density-independent pixels: “dips”)

- 1 dp “virtual pixel” is equal to 1 pixel on a 160dpi phone

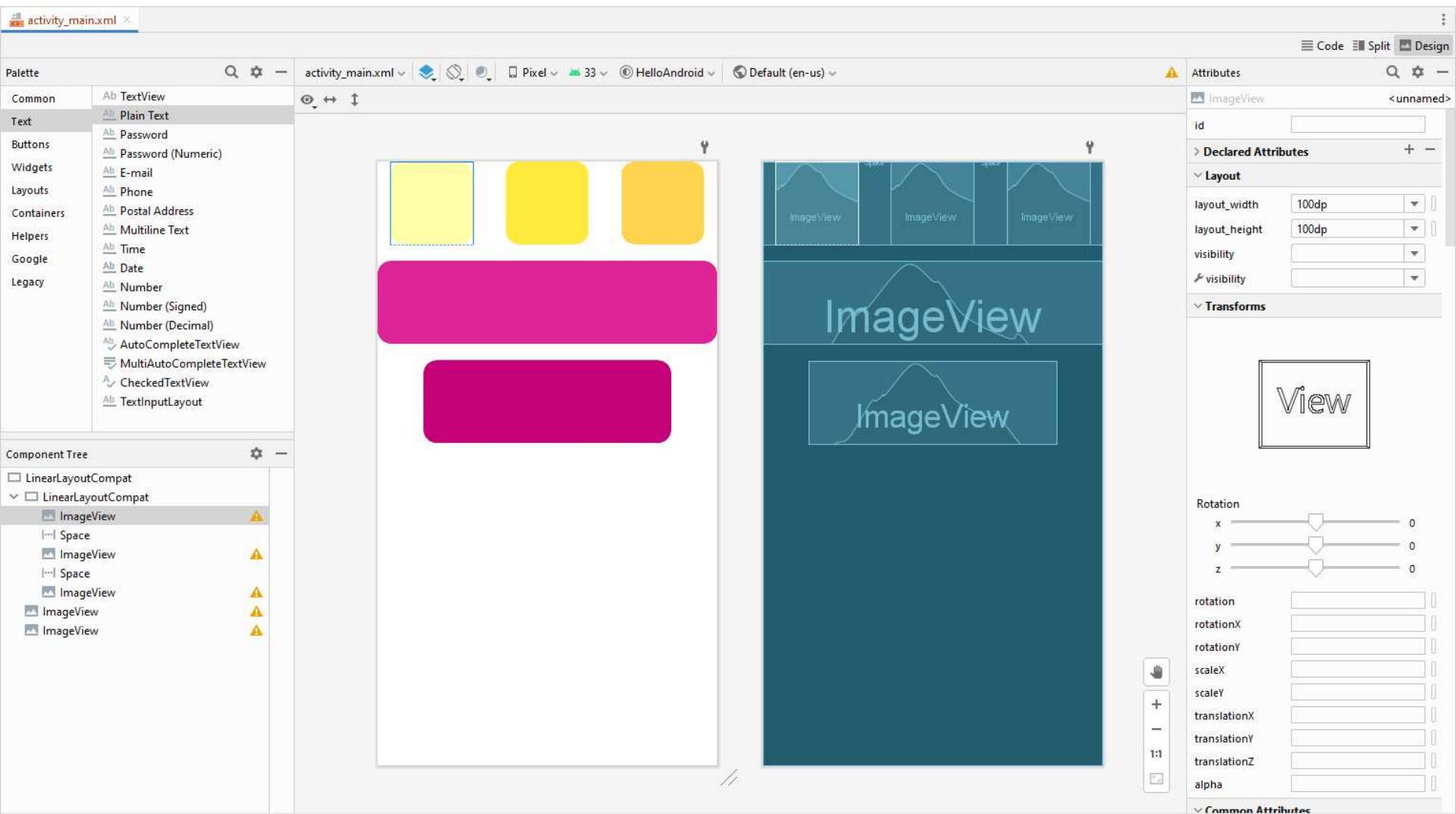
Text sizes should be defined in **sp** (scalable pixels)

- sp is the same as dp by default, but changes according to user’s preferred text size system setting
- Other units supported, e.g., **px**, **mm**, ...

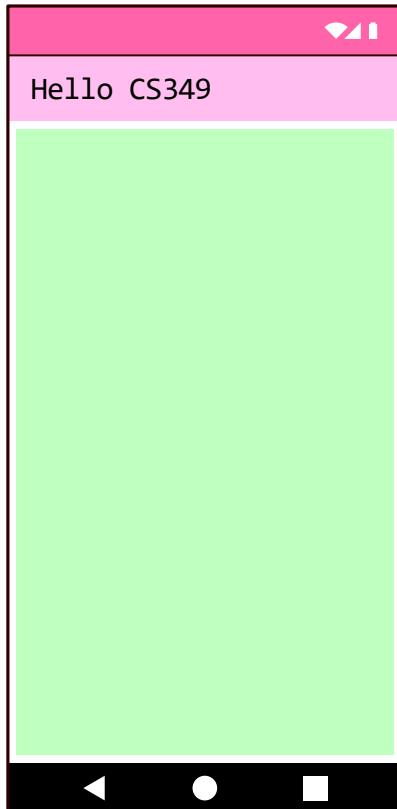


# GUI Designer

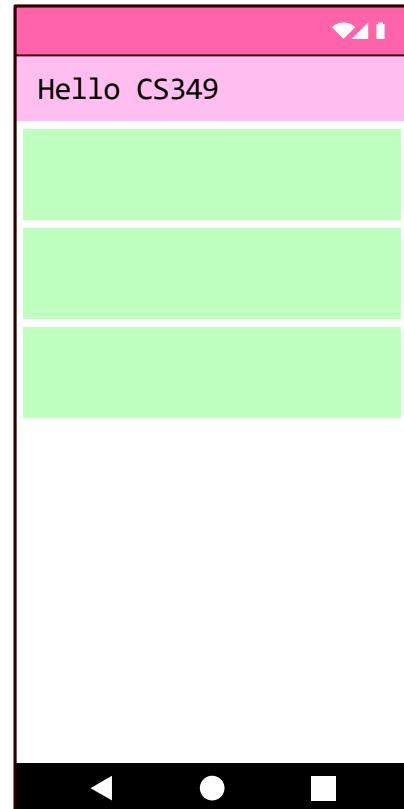
The GUI Designer allows to create a UI using a GUI instead of XML code.



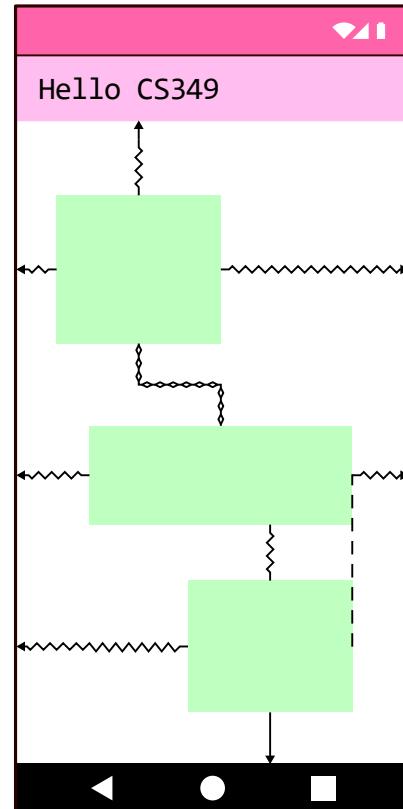
# Common Layouts



**FrameLayout**  
Displays a single View in its area.



**LinearLayout**  
Organizes its children into a single (horizontal) row or (vertical) column.



**ConstraintLayout**  
Relative positioning of children based on relationships and constraints.

# LinearLayout

```
res.drawable.rectangle.xml
```

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="..." android:shape="rectangle">
    <corners android:radius="16dp" />
</shape>
```

```
res.layout.activity_main.xml
```

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="..." xmlns:tools="..."
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
```

```
    <ImageView android:layout_height="100dp"
        android:layout_width="100dp"
        android:src="@drawable/rectangle"
        app:tint="@color/math1" />
```

```
    <ImageView android:layout_height="100dp"
        android:layout_width="200dp"
        android:src="@drawable/rectangle"
        app:tint="@color/math2" />
```

```
    <ImageView android:layout_height="100dp"
        android:layout_width="300dp"
        android:src="@drawable/rectangle"
        app:tint="@color/math3" />
```

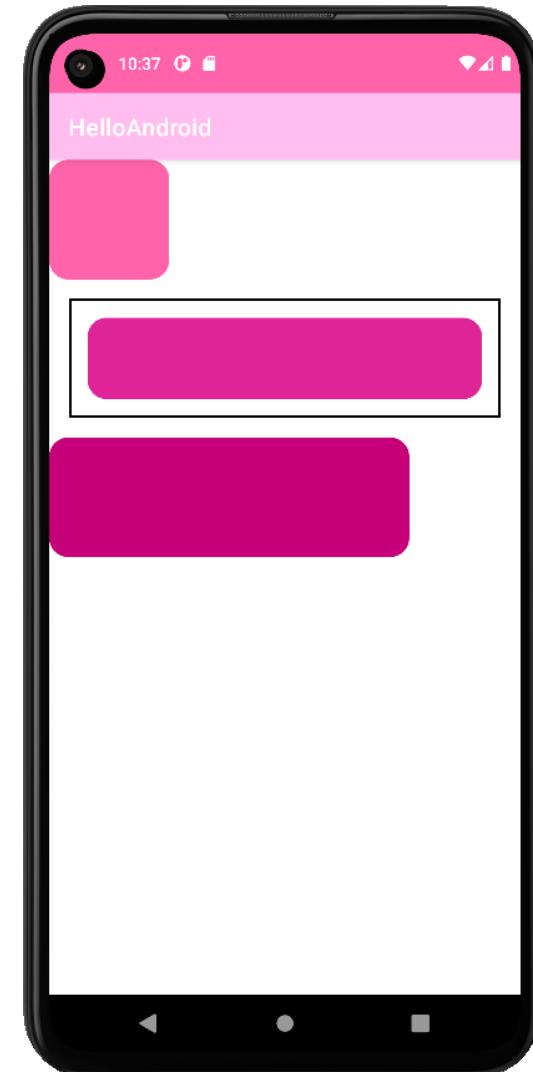
```
</LinearLayout>
```



# LinearLayout – Children's Attributes

```
<ImageView android:layout_height="100dp"
           android:layout_width="100dp"
           android:src="@drawable/rectangle"
           app:tint="@color/math1" />
<ImageView android:layout_height="100dp"
           android:layout_width="match_parent"
           android:src="@drawable/rectangle"
           app:tint="@color/math2"
           android:padding="16dp"
           android:layout_margin="16dp" />
<ImageView android:layout_height="100dp"
           android:layout_width="300dp"
           android:src="@drawable/rectangle"
           app:tint="@color/math3"
           android:onClick="rectTouched" />
```

- `layout_width="match_parent"`: view dimension expands to match parent (also `layout_height`)
- `layout_width="wrap_content"`: view dimension matches size of internal content
- `padding="16dp"`: add space inside the boundary
- `layout_margin="16dp"`: add space outside the boundary
- `onClick="rectTouched"`: function called on touch

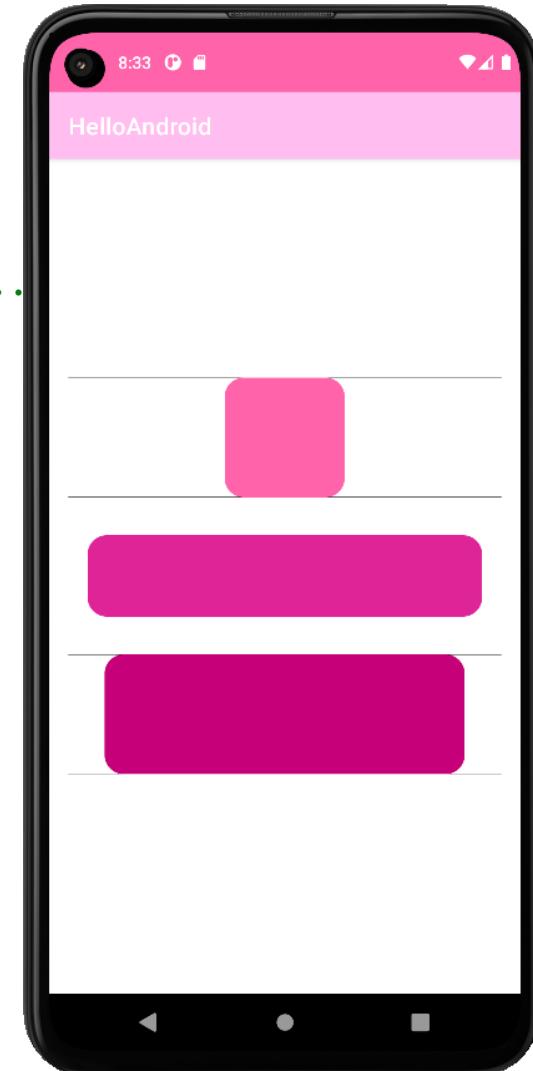


# LinearLayout – Attributes

```
res.layout.activity_main.xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="..." xmlns:app="..." xmlns:tools="..."
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    app:divider="@color/black"
    app:dividerPadding="16dp"
    app:showDividers="end|middle|beginning"
    tools:context=".MainActivity">

    ...
</LinearLayout>
```

- `orientation="vertical"`: shows content as a column  
(also `"horizontal"`)
- `gravity="center"`: how children are positioned
- `showDividers="end|middle|beginning"`: shows dividers between children



# LinearLayout – Spacing

LinearLayout has no attribute for spacing between child views.

Workarounds include:

- Adding margins to each child nodes:

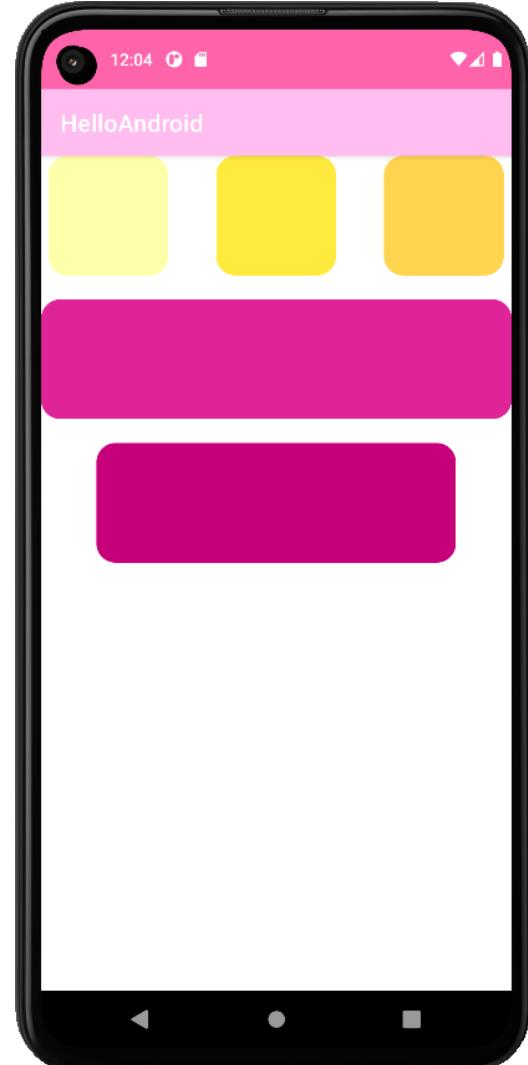
```
    android:layout_marginVertical="10dp"
```

- Inserting Space views between child nodes:

```
<Space android:layout_width="match_parent"  
      android:layout_height="20dp" />
```

# LinearLayout – Nesting Layouts

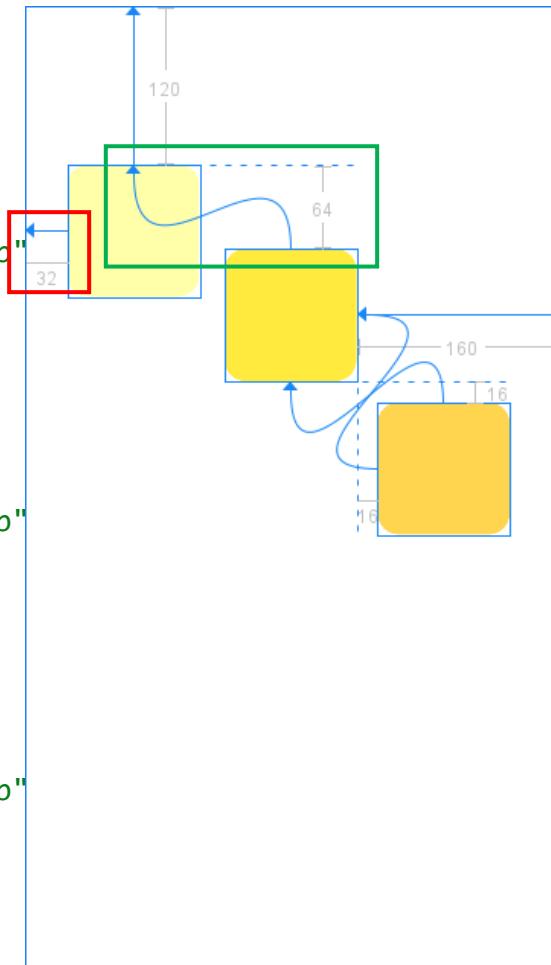
```
<LinearLayout  
    xmlns:android="..." xmlns:app="..." xmlns:tools="..."  
    android:layout_width="match_parent" android:layout_height="match_parent"  
    android:orientation="vertical" android:gravity="center_horizontal"  
    app:divider="@drawable/divider" app:showDividers="middle"  
    tools:context=".MainActivity">  
  
<LinearLayout  
    android:layout_width="match_parent" android:layout_height="wrap_content"  
    android:orientation="horizontal" android:gravity="center_horizontal">  
  
    <ImageView android:layout_height="100dp" android:layout_width="100dp"  
              android:src="@drawable/rectangle" app:tint="@color/uw1" />  
    <Space android:layout_height="0dp" android:layout_width="40dp" />  
    <ImageView android:layout_height="100dp" android:layout_width="100dp"  
              android:src="@drawable/rectangle" app:tint="@color/uw2" />  
    <Space android:layout_height="0dp" android:layout_width="40dp" />  
    <ImageView android:layout_height="100dp" android:layout_width="100dp"  
              android:src="@drawable/rectangle" app:tint="@color/uw3" />  
/</LinearLayout>  
  
<ImageView android:layout_height="100dp" android:layout_width="match_parent"  
          android:src="@drawable/rectangle" app:tint="@color/math3" />  
  
<ImageView android:layout_height="100dp" android:layout_width="300dp"  
          android:src="@drawable/rectangle" app:tint="@color/math4" />  
  
</LinearLayout>
```



# ConstraintLayout – Absolute Positioning

Absolute positioning, relative to **parent** or **another node**:

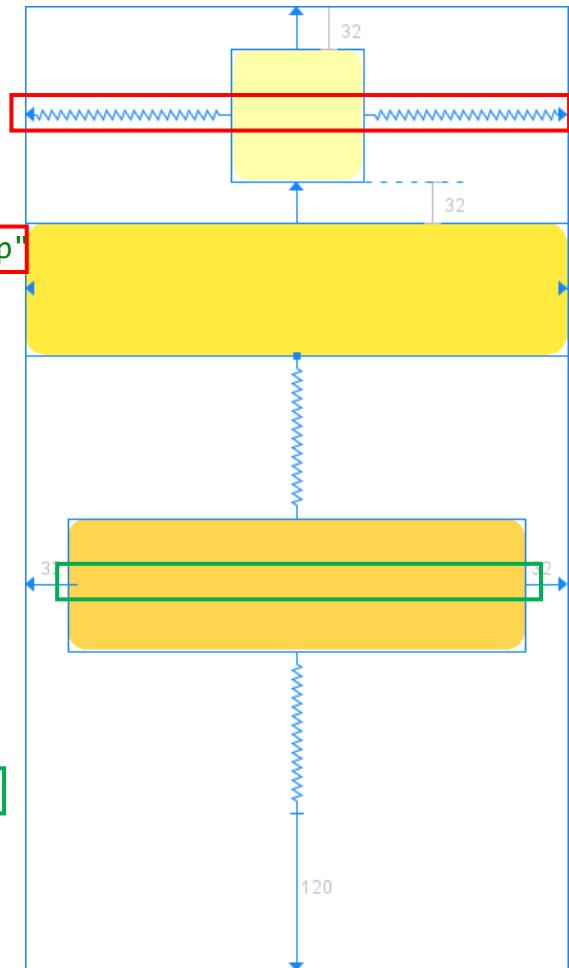
```
<ImageView android:id="@+id/imageView1"
    android:layout_height="100dp" android:layout_width="100dp"
    android:src="@drawable/rectangle" app:tint="@color/uw1"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_marginStart="32dp"
    android:layout_marginTop="120dp" />
<ImageView android:id="@+id/imageView2"
    android:layout_height="100dp" android:layout_width="100dp"
    android:src="@drawable/rectangle" app:tint="@color/uw2"
    app:layout_constraintTop_toTopOf="@+id/imageView1"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_marginEnd="160dp"
    android:layout_marginTop="60dp" />
<ImageView
    android:layout_height="100dp" android:layout_width="100dp"
    android:src="@drawable/rectangle" app:tint="@color/uw3"
    app:layout_constraintTop_toBottomOf="@+id/imageView2"
    app:layout_constraintStart_toEndOf="@+id/imageView2"
    android:layout_marginTop="16dp"
    android:layout_marginStart="16dp" />
```



# ConstraintLayout – Centered Positioning

Positioning between two points with **fixed** or **calculated** node dimension.

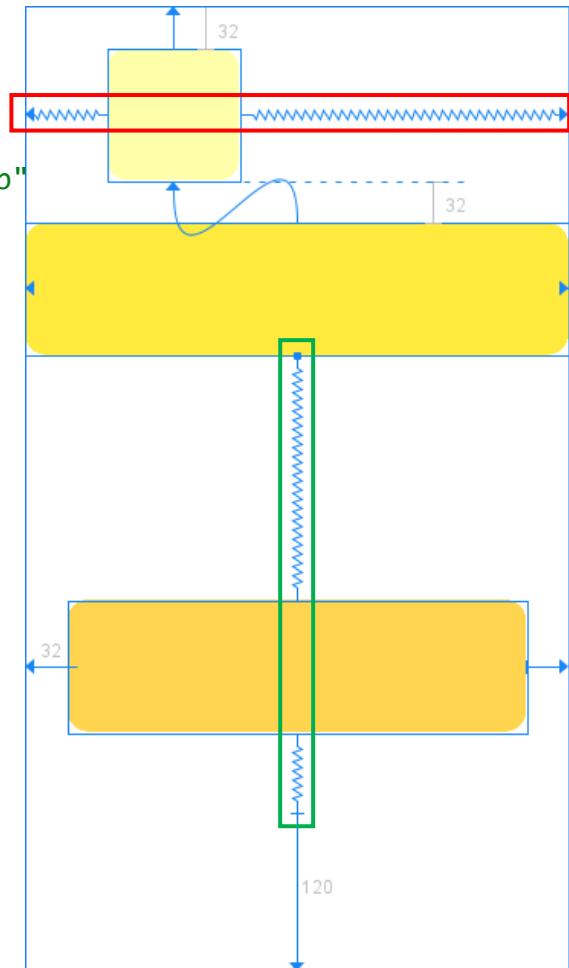
```
<ImageView android:id="@+id/imageView1"
    android:layout_height="100dp" android:layout_width="100dp"
    android:src="@drawable/rectangle" app:tint="@color/uw1"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_marginTop="32dp"/>
<ImageView android:id="@+id/imageView2"
    android:layout_height="100dp" android:layout_width="0dp"
    android:src="@drawable/rectangle" app:tint="@color/uw2"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/imageView1"
    android:layout_marginTop="32dp"/>
<ImageView
    android:layout_height="100dp" android:layout_width="0dp"
    android:src="@drawable/rectangle" app:tint="@color/uw3"
    app:layout_constraintTop_toBottomOf="@+id/imageView2"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    android:layout_marginBottom="120dp"
    android:layout_marginStart="32dp"
    android:layout_marginEnd="32dp"/>
```



# ConstraintLayout – Centered Positioning

## Offsetting (“biasing”) nodes

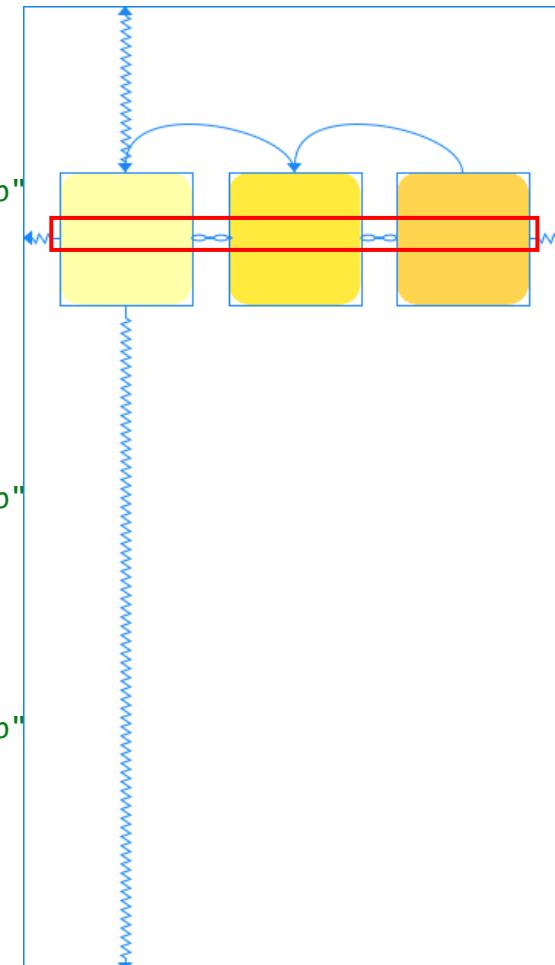
```
<ImageView android:id="@+id/imageView1"
    android:layout_height="100dp" android:layout_width="100dp"
    android:src="@drawable/rectangle" app:tint="@color/uw1"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.2"
    app:layout_constraintTop_toTopOf="parent"
    android:layout_marginTop="32dp"/>
<ImageView android:id="@+id/imageView2"
    android:layout_height="100dp" android:layout_width="0dp"
    android:src="@drawable/rectangle" app:tint="@color/uw2"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/imageView1"
    android:layout_marginTop="32dp"/>
<ImageView
    android:layout_height="100dp" android:layout_width="0dp"
    android:src="@drawable/rectangle" app:tint="@color/uw3"
    app:layout_constraintTop_toBottomOf="@+id/imageView2"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintVertical_bias="0.75"
    android:layout_marginBottom="120dp"
    android:layout_marginStart="32dp"
    android:layout_marginEnd="32dp"/>
```



# ConstraintLayout – Chains

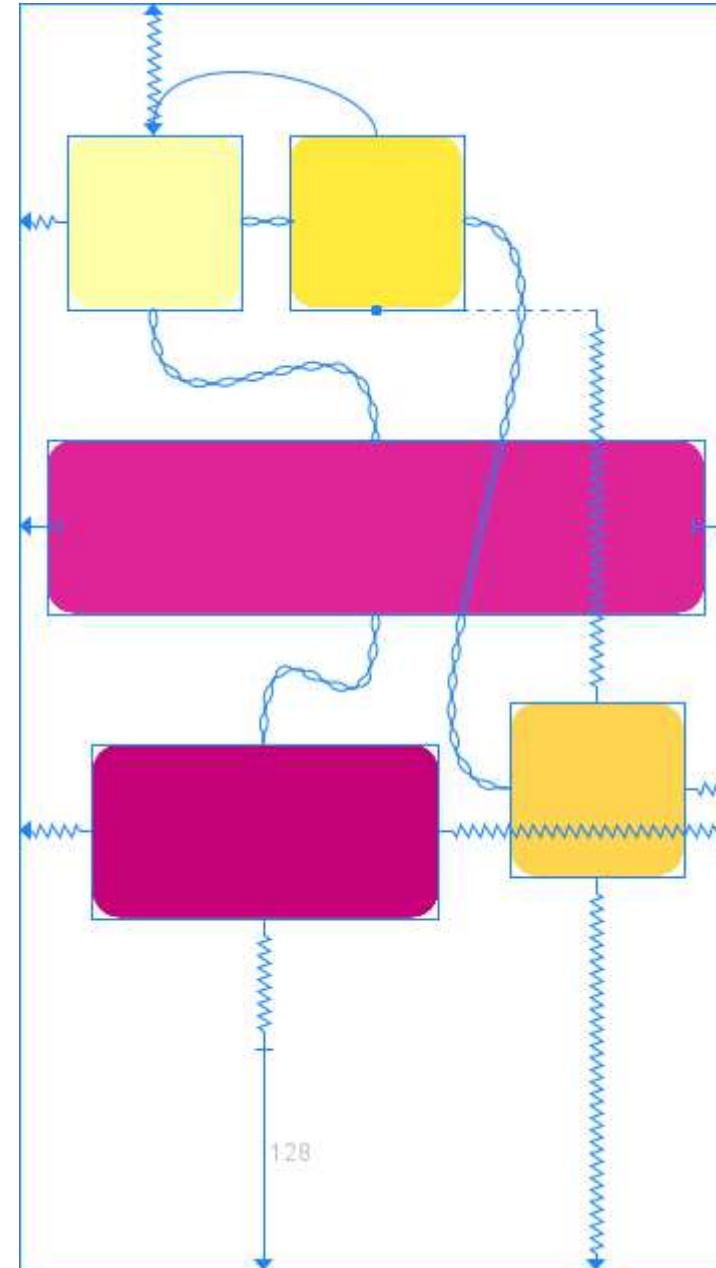
Chains group nodes together in one dimension:

```
<ImageView android:id="@+id/imageView1"
    android:layout_height="100dp" android:layout_width="100dp"
    android:src="@drawable/rectangle" app:tint="@color/uw1"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/imageView2"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintVertical_bias="0.2" />
<ImageView android:id="@+id/imageView2"
    android:layout_height="100dp" android:layout_width="100dp"
    android:src="@drawable/rectangle" app:tint="@color/uw2"
    app:layout_constraintStart_toEndOf="@+id/imageView1"
    app:layout_constraintEnd_toStartOf="@+id/imageView3"
    app:layout_constraintTop_toTopOf="@+id/imageView1" />
<ImageView android:id="@+id/imageView3"
    android:layout_height="100dp" android:layout_width="100dp"
    android:src="@drawable/rectangle" app:tint="@color/uw3"
    app:layout_constraintStart_toEndOf="@+id/imageView2"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="@+id/imageView2" />
```



# ConstraintLayout – Chains

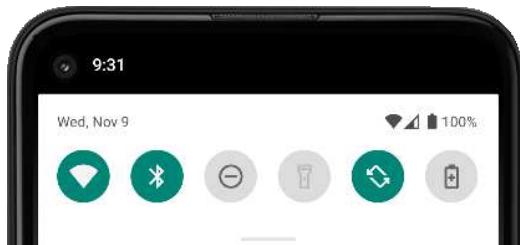
Combining multiple chains...



# Layouts and Device Rotation

By default, Android handles device rotation by rotating the content of the app. Part of this process includes recreating the app, i.e., calling `onDestroy()` and `onCreate()`. (We will learn more about the Android app life-cycle later.)

When testing, make sure AVD has device rotation ON:

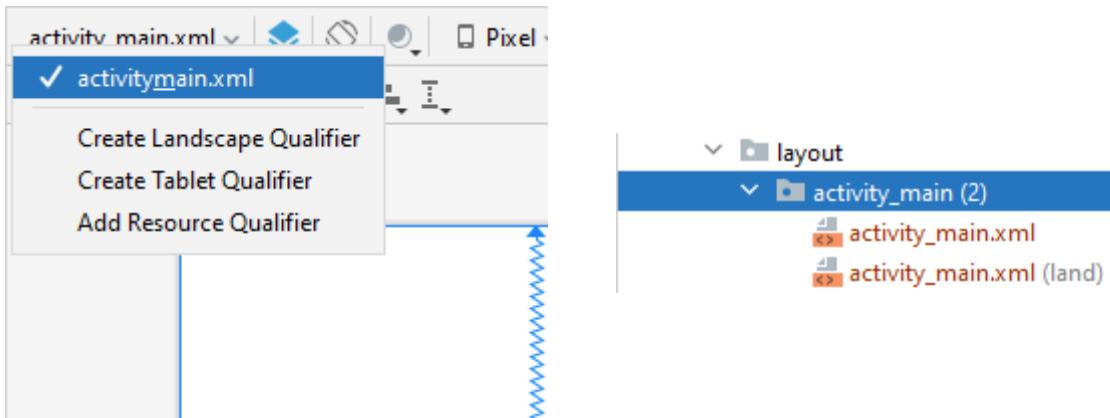


If the default behaviour is not sufficient, it is possible to handle device rotation within the app.

# Layouts and Device Rotation

Rotating the device causes a *configuration change*

You may specify separate layouts for landscape and portrait orientation or use the same for both.



# Layouts and Device Orientation

To react programmatically to orientation change:

- Add this to your main activity in `AndroidManifest.xml`:

```
    android:configChanges="orientation|screenSize"
```

- Add this to your main activity, e.g., in `MainActivity.kt`:

```
override fun onConfigurationChanged(newConfig: Configuration) {
    super.onConfigurationChanged(newConfig)
    when (newConfig.orientation) {
        Configuration.ORIENTATION_LANDSCAPE -> {
            Log.i("OrientationChange", "Landscape")}
        Configuration.ORIENTATION_PORTRAIT -> {
            Log.i("OrientationChange", "Portrait") }
        else -> {
            Log.e("OrientationChange", "Whaaat...") }
    }
}
```

# Layouts and Device Rotation

For more fine-grain information about the rotation angle:

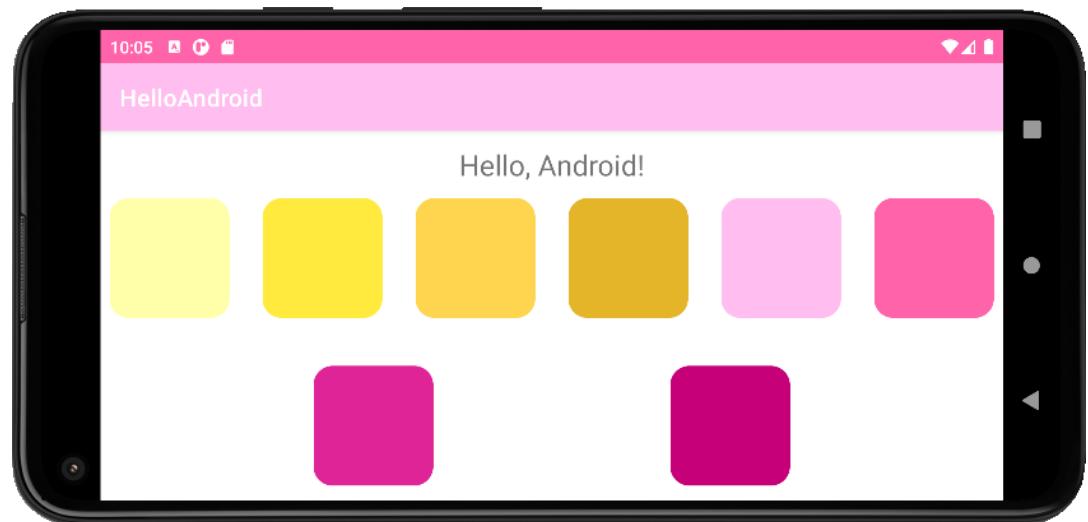
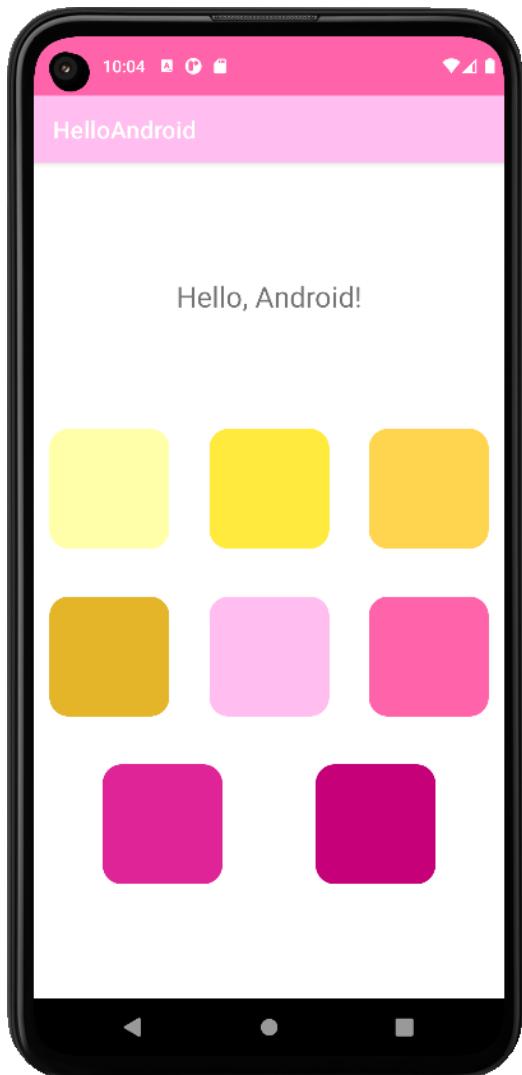
- Add this to your main activity in `AndroidManifest.xml`:

```
android:configChanges="orientation|screenSize"
```

- Add this to your `.kt` file, e.g., `MainActivity.kt`:

```
val oel = object : OrientationEventListener(applicationContext) {  
    override fun onOrientationChanged(degrees: Int) {  
        Log.d("OrientationEvent",  
            "New device orientation is $degrees degrees.")  
    }  
}  
oel.enable()
```

# Layouts and Device Rotation



U  
CS 349

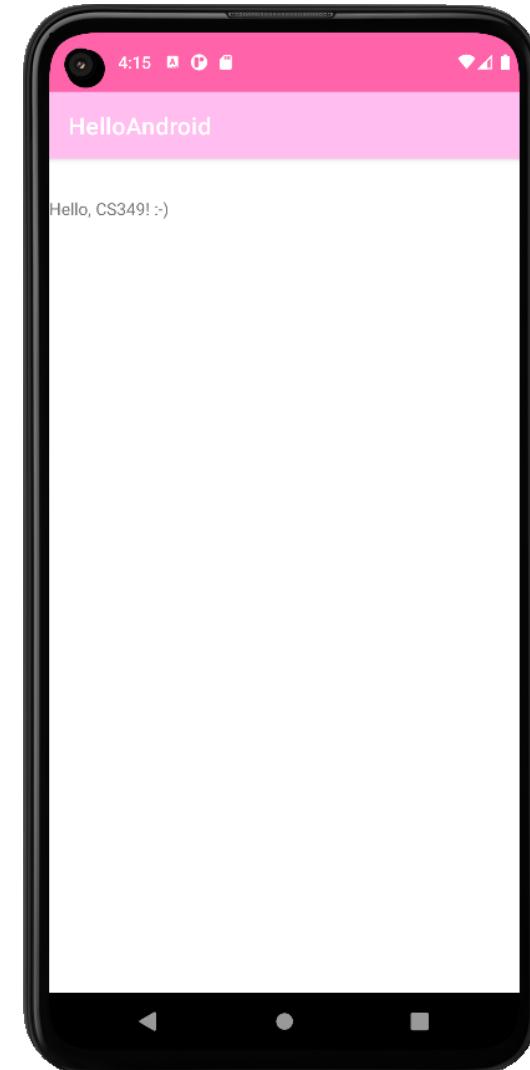
# Android Widgets



# TextView

TextView display text to the user.

```
<TextView android:id="@+id/myTextView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/greeting"
    android:layout_marginTop="32dp" />
```



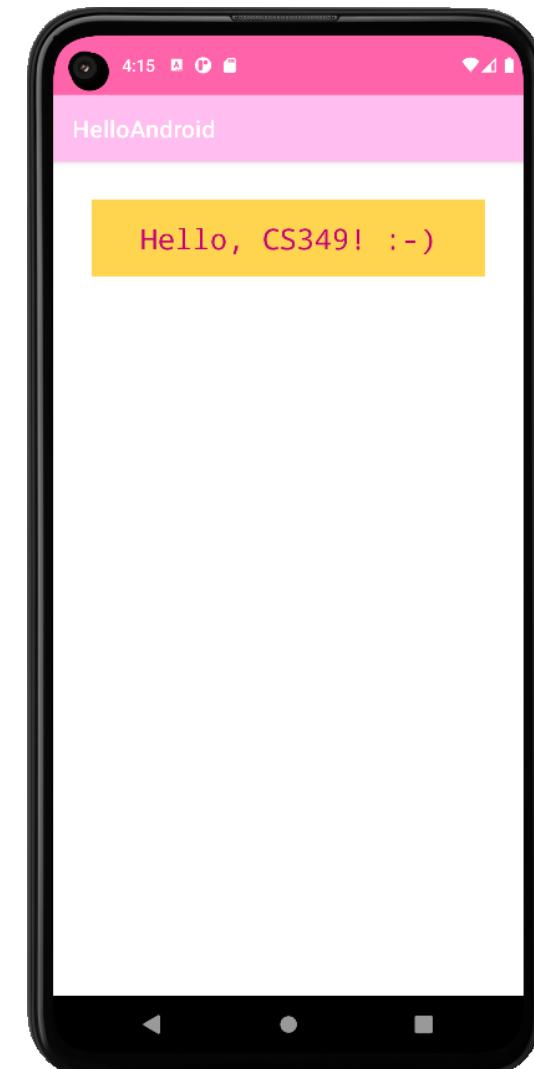
# TextView – Common Properties

.xml-file:

```
<TextView android:id="@+id/textView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/greeting"
    android:textSize="24sp"
    android:typeface="monospace"
    android:textAlignment="center"
    android:textColor="@color/math4"
    android:background="@color/uw3"
    android:layout_margin="@dimen/viewMargin"
    android:padding="16dp" />
```

.kt-file:

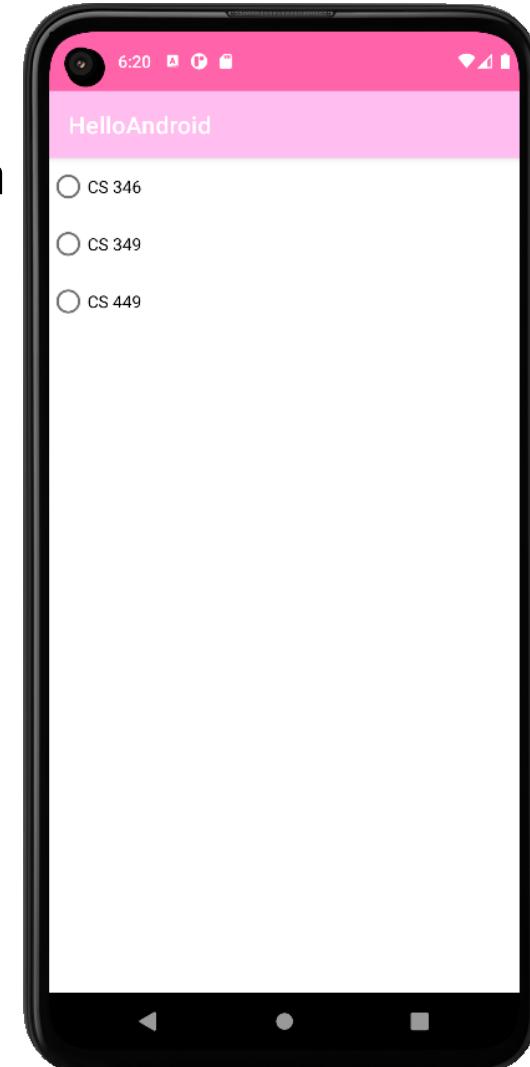
```
fun textViewTap(view: View) {
    findViewById<TextView>(R.id.textView).apply {
        text = "$text!"
    }
    // alternative implementation using casting
    (view as TextView).apply {
        text = "$text!"
    }
}
```



# RadioButton

Radio buttons allow the user to select one option from a set. Use radio buttons if you think that the user needs to see all available options side-by-side.

```
<RadioGroup android:layout_width="wrap_content"
            android:layout_height="wrap_content">
    <RadioButton android:id="@+id/radBtn346"
                android:text="@string/radBtnLbl346"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
    <RadioButton android:id="@+id/radBtn349"
                android:text="@string/radBtnLbl346"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
    <RadioButton android:id="@+id/radBtn449"
                android:text="@string/radBtnLbl449"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
</RadioGroup>
```



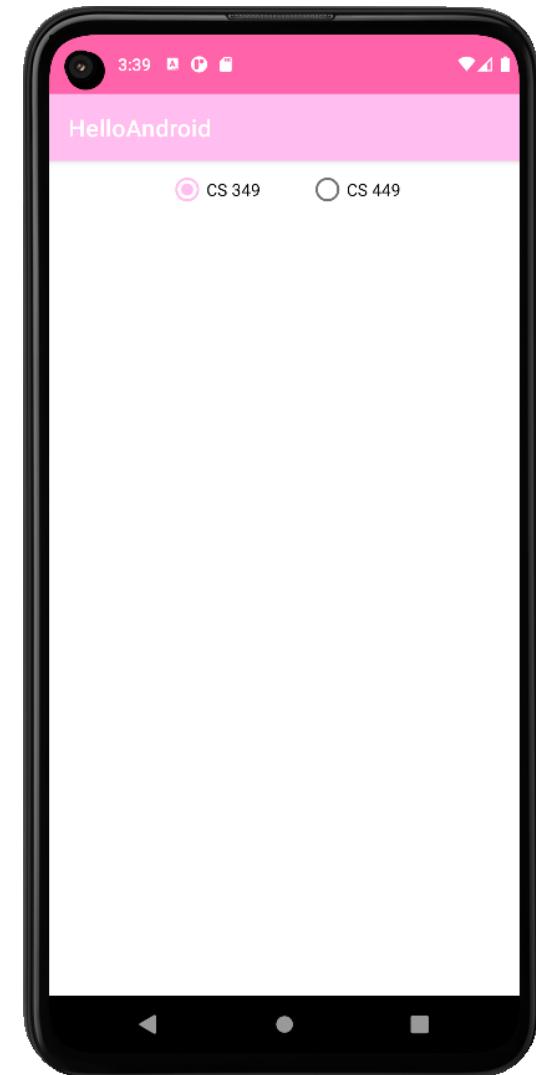
# RadioButton – Common Properties

.xml-file:

```
<RadioGroup  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:orientation="horizontal" android:gravity="center"  
    android:checkedButton="@+id/rb349">  
    <RadioButton android:id="@+id/rb349"  
        android:text="@string/rbl349"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:onClick="radioGroupClicked" />  
    <Space android:layout_width="40dp"  
        android:layout_height="wrap_content" />  
    <RadioButton android:id="@+id/rb449"  
        android:text="@string/rbl449"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:onClick="radioGroupClicked" />  
</RadioGroup>
```

.kt-file:

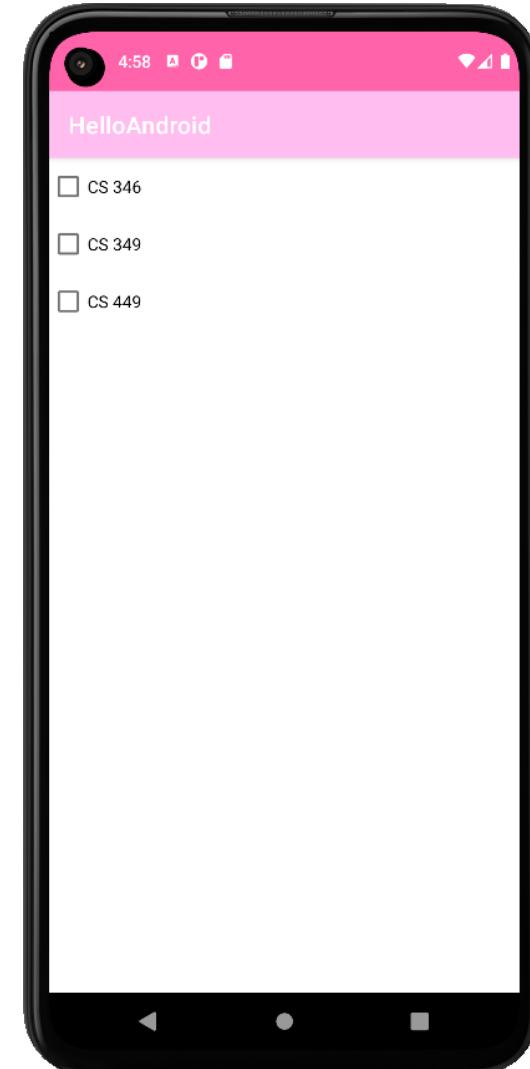
```
fun radioGroupClicked(view: View) {  
    Log.i("RADIO", view.id.toString())  
    when (view.id) {  
        R.id.rb349 -> {}  
        R.id.rb449 -> {}  
        else -> {}  
    }  
}
```



# CheckBox

Checkboxes allow the user to select one or more options from a set. Typically, they should be presented in a vertical list.

```
<CheckBox android:id="@+id/cb346"
          android:text="@string/cbl346"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content" />
<CheckBox android:id="@+id/cb349"
          android:text="@string/cbl349"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content" />
<CheckBox android:id="@+id/cb449"
          android:text="@string/cbl449"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content" />
```



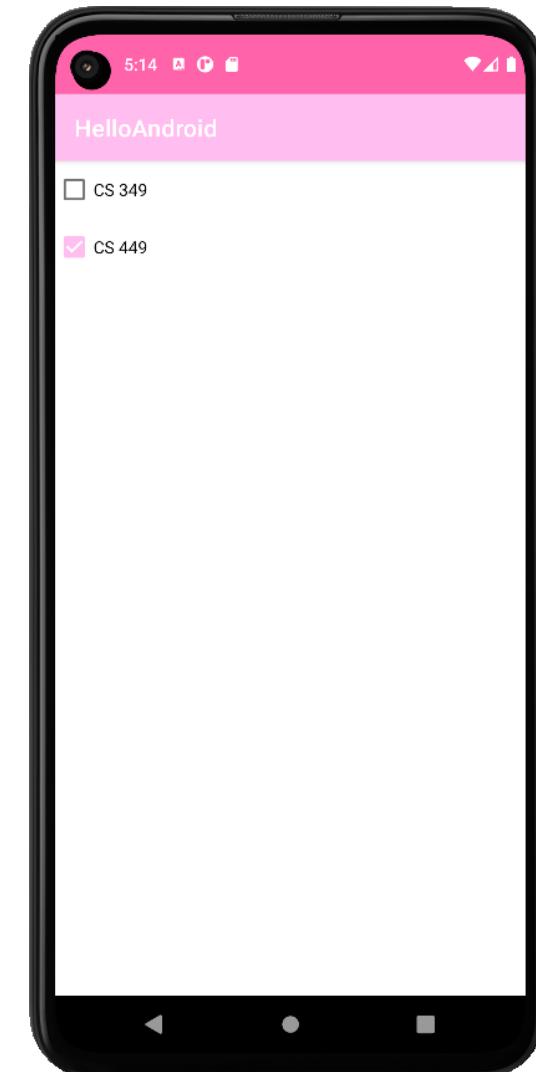
# CheckBox – Common Properties

## .xml-file

```
<CheckBox android:id="@+id/cb349"
    android:text="@string/cb1349"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="checkBoxClicked" />
<CheckBox android:id="@+id/cb449"
    android:text="@string/cb1449"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:checked="true"
    android:onClick="checkBoxClicked" />
```

## .kt-file

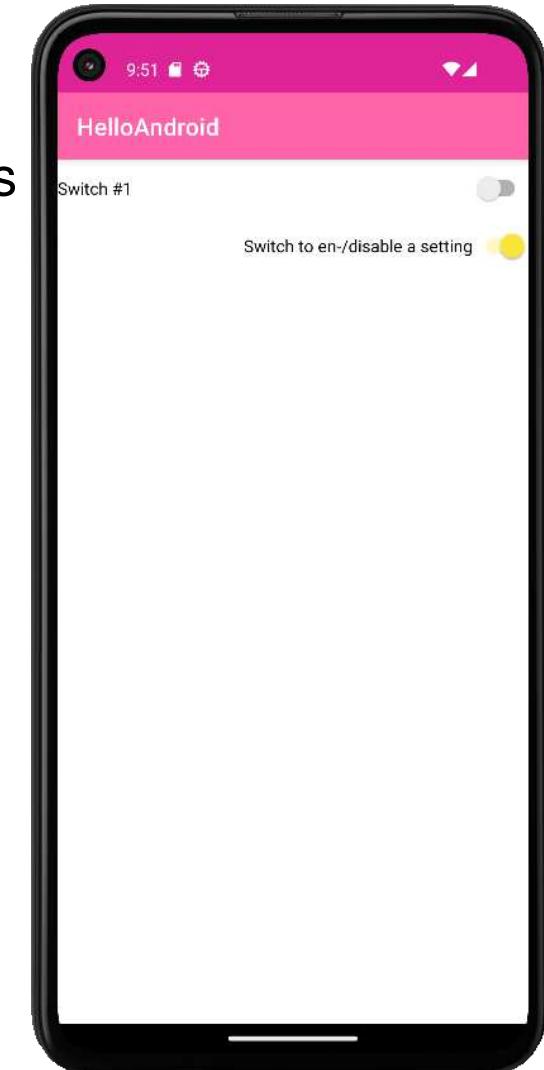
```
fun checkBoxClicked(view: View) {
    view as CheckBox
    Log.i("CHECK", "${view.id}:${view.isChecked}")
    when (view.id) {
        R.id.cb349 -> { }
        R.id.cb449 -> { }
        else -> { }
    }
}
```



# Switch

Switches allow the user to chose between two options from a single property.

```
<Switch  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="Switch #1" />  
  
<Switch  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Switch to en-/disable a setting"/>
```



# Switch – Custom Styling

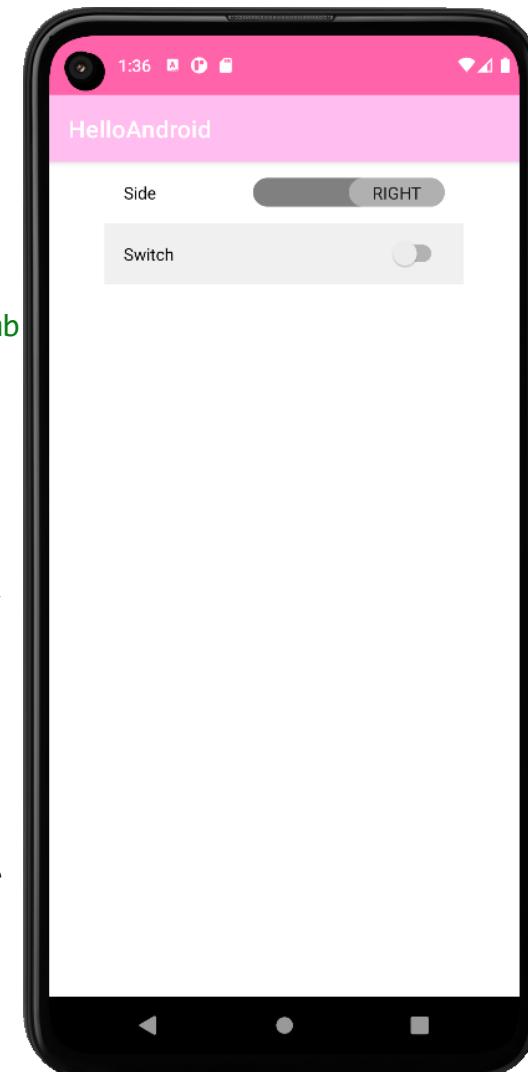
.xml-file:

```
<Switch android:layout_width="300dp"
        android:layout_height="wrap_content"
        android:checked="true" android:showText="true"
        android:track="@drawable/track" android:thumb="@drawable/thumb"
        android:textOn="Right" android:textOff="Left"
        android:text="Side" android:padding="@dimen/viewPadding" />
<Switch android:id="@+id/colorSwitch"
        android:layout_width="300dp"
        android:layout_height="wrap_content"
        android:background="#FFF0F0F0"
        android:text="Switch" android:padding="@dimen/viewPadding" />
```

.kt-file:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    findViewById<Switch>(R.id.colorSwitch).setOnCheckedChangeListener
    { view, isChecked -
        view as Switch
        view.text = String.format(getString(R.string.switchText),
                                  isChecked)
        if (isChecked)
            view.thumbDrawable.setTint(getColor(R.color.white))
        else
            view.thumbDrawable.setTint(getColor(R.color.black))
    }
}
```



# SeekBar

SeekBar allows users to select a continuous value for a property.

```
<SeekBar android:id="@+id/seekBar"  
        android:layout_width="match_parent"  
        android:layout_height="32dp" />
```



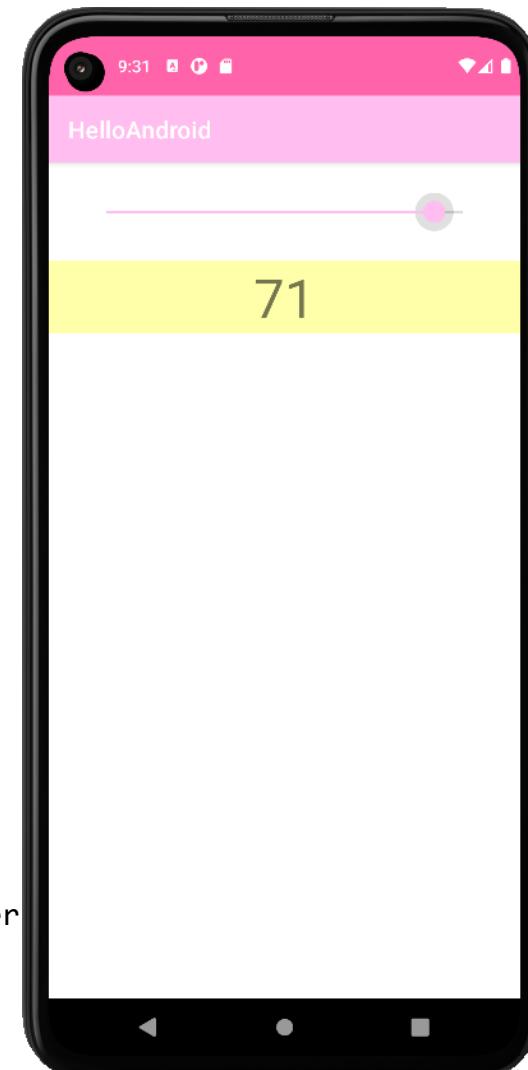
# SeekBar – Custom Styling

.xml-file:

```
<SeekBar android:id="@+id/seekBar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:min="25" android:max="75"
    android:layout_margin="@dimen/viewMargin" />
<TextView android:id="@+id/seekBarValue"
    android:layout_width="match_parent"
    android:layout_height="60dp"
    android:autoSizeTextType="uniform"
    android:textAlignment="center"
    android:background="@color/uw1" />
```

.kt-file:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    val seekBar = findViewById<SeekBar>(R.id.seekBar)
    val seekText = findViewById<TextView>(R.id.seekBarValue)
    seekBar.setOnSeekBarChangeListener(object: OnSeekBarChangeListener {
        override fun onProgressChanged(view: SeekBar?, value: Int,
                                       fromUser: Boolean) {
            seekText.text = value.toString()
        }
        override fun onStartTrackingTouch(p0: SeekBar?) { }
        override fun onStopTrackingTouch(p0: SeekBar?) { }
    })
}
```



# Button, ImageButton

A button consists of text or an icon (or both text and an icon), which should communicate what action occurs when the user touches it.

```
<Button android:id="@+id/myBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/btnText" />
```

```
<ImageButton android:id="@+id/myImgBtn"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:src="@mipmap/ic_launcher" />
```



# Button, ImageButton – Common Properties

.xml-file:

```
<Button android:id="@+id/myBtn"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/viewMargin"
        android:text="@string/btnText"
        android:drawableLeft="@mipmap/ic_launcher"
        android:onClick="btnClicked"
        app:strokeWidth="4dp"
        app:strokeColor="@color/math2"
        app:cornerRadius="4dp" />

<ImageButton android:id="@+id/myImgBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@mipmap/ic_launcher"
        android:backgroundTint="@color/transparent"
        android:onClick="btnClicked" />
```

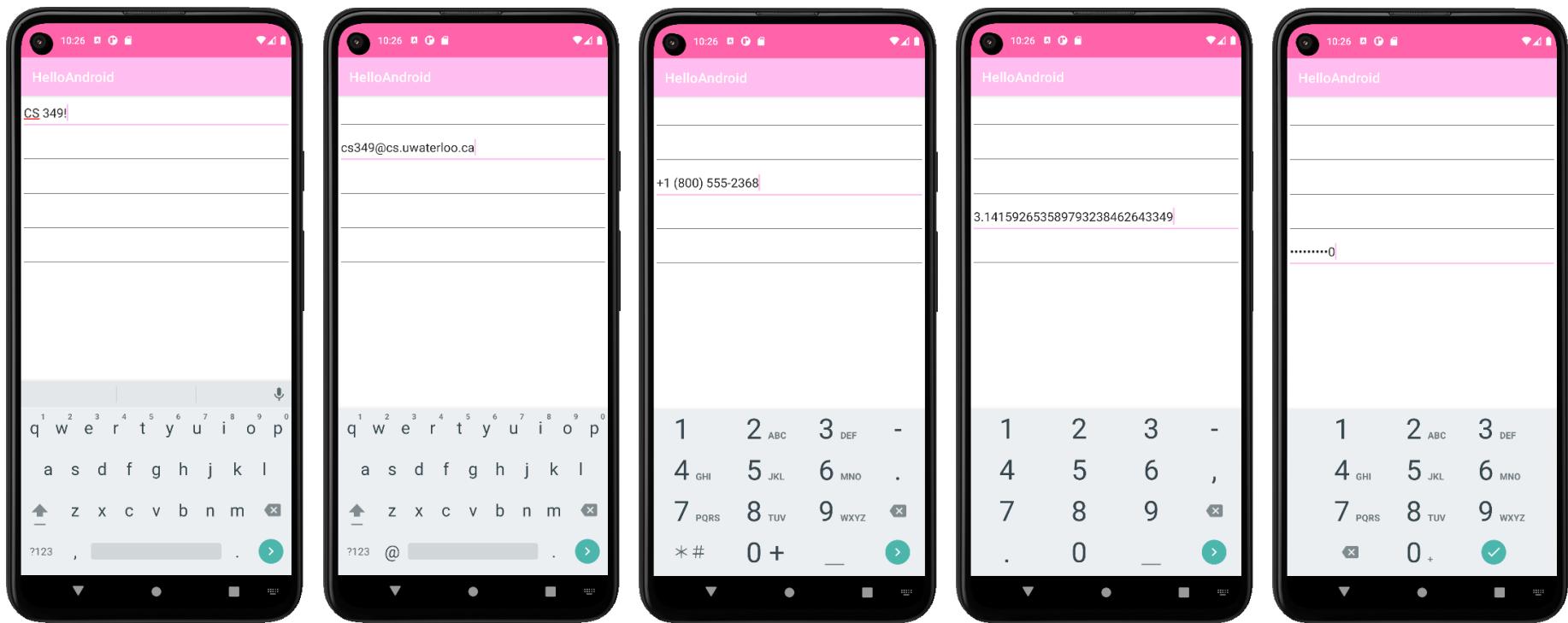
.kt-file:

```
fun btnClicked(view: View) {
    when(view.id) {
        R.id.myBtn -> { Log.i("BUTTON", "myBtn clicked") }
        R.id.myImgBtn -> { Log.i("BUTTON", "myIMgBtn clicked") }
    }
}
```



# EditText

```
<EditText android:inputType="text"
          android:layout_width="match_parent" android:layout_height="50dp" />
<EditText android:inputType="textEmailAddress"
          android:layout_width="match_parent" android:layout_height="50dp" />
<EditText android:inputType="phone"
          android:layout_width="match_parent" android:layout_height="50dp" />
<EditText android:inputType="numberDecimal"
          android:layout_width="match_parent" android:layout_height="50dp" />
<EditText android:inputType="numberPassword"
          android:layout_width="match_parent" android:layout_height="50dp" />
```



# EditText – Validation

## .xml-file:

```
<EditText android:id="@+id/email"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:inputType="textEmailAddress"
    android:maxLength="@integer/emailMaxLen" />
<LinearLayout
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    android:gravity="end">
    <Button android:id="@+id	btnSubmit"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Submit" android:onClick="btnSubmit"
        android:enabled="false" />
    <Button android:id="@+id	btnClear"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Clear" android:onClick="btnClear"
        android:enabled="false" />
</LinearLayout>
```



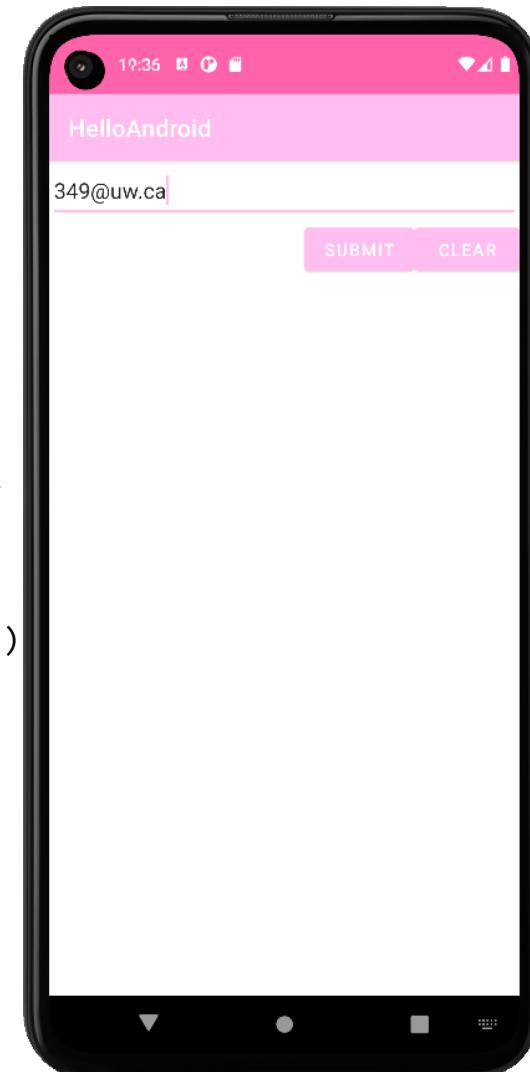
# EditText – Validation

.kt-file:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    // ...
    findViewById<EditText>(R.id.email).addTextChangedListener(
        object : TextWatcher {
            override fun beforeTextChanged(p0: CharSequence?,
                                         p1: Int, p2: Int, p3: Int) { }
            override fun onTextChanged(seq: CharSequence?,
                                      p1: Int, p2: Int, p3: Int) {
                val emailRegex =
                    Regex("^[a-zA-Z0-9+]+@[a-zA-Z0-9]{2,}\.\.(?:ca|com)\$")
                findViewById<Button>(R.id.btnSubmit).isEnabled =
                    seq?.matches(emailRegex) ?: false
                findViewById<Button>(R.id.btnClear).isEnabled =
                    seq.isNullOrEmpty().not()
            }
            override fun afterTextChanged(p0: Editable?) { }
        })
}

fun btnSubmit(view: View) {
    Log.i("BUTTON",
          "Submitting ${findViewById<EditText>(R.id.email).text}.")
}

fun btnClear(view: View) {
    findViewById<EditText>(R.id.email).text.clear()
}
```

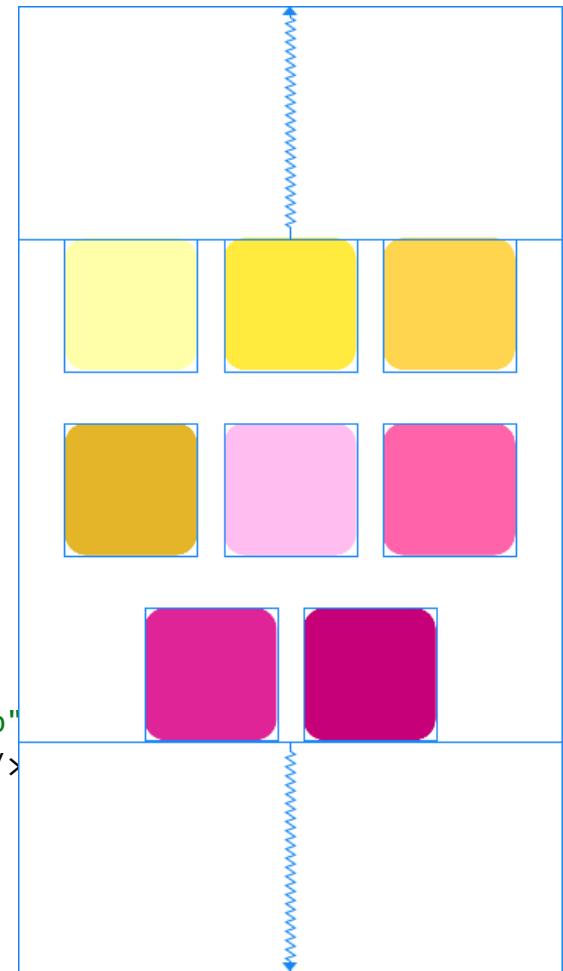


# ConstraintLayout – Flow

Create a grid-like layout with nodes wrapping to next line:

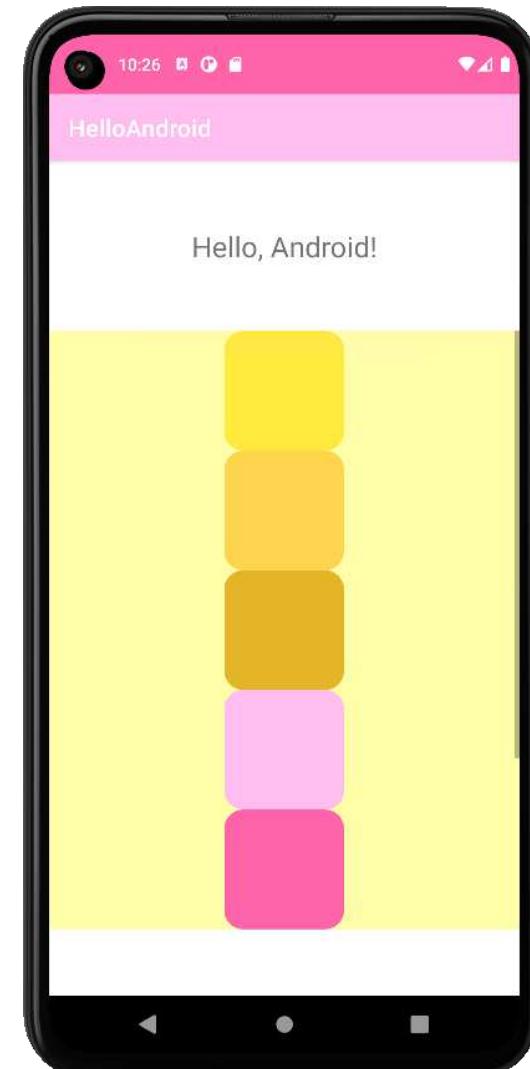
```
<androidx.constraintlayout.helper.widget.Flow
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:flow_wrapMode="aligned"
    app:flow_horizontalStyle="packed"
    app:flow_horizontalGap="20dp"
    app:flow_verticalGap="40dp"
    app:constraint_referenced_ids="imageView1,..."
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"/>

<ImageView android:id="@+id/imageView1"
    android:layout_height="100dp" android:layout_width="100dp"
    android:src="@drawable/rectangle" app:tint="@color/uw1" />
...
...
```



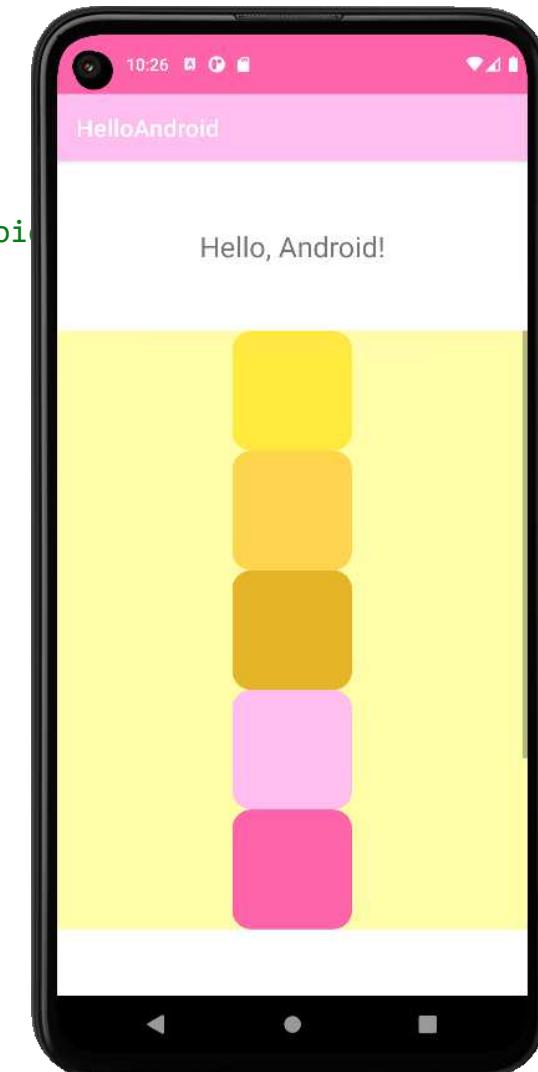
# Layouts and Scrolling

```
<TextView android:id="@+id/textView" android:text="@string/greeting"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="24sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toTopOf="@+id/scrollView2" />
<ScrollView android:id="@+id/scrollView"
    android:layout_width="match_parent"
    android:layout_height="500dp"
    app:layout_constraintTop_toBottomOf="@+id/textView"
    app:layout_constraintBottom_toBottomOf="parent">
    <LinearLayout android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:gravity="center"
        android:backgroundTint="@color/uw1">
        <ImageView android:layout_height="100dp"
            android:layout_width="100dp"
            android:src="@drawable/rectangle"
            app:tint="@color/uw2" />
        ...
    </LinearLayout>
</ScrollView>
```



# Layouts and Scrolling – Including Layouts

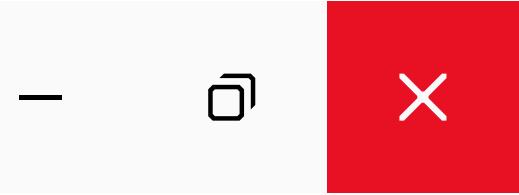
```
rectangle_scroll_view.xml:  
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:orientation="vertical"  
    android:gravity="center"  
    android:backgroundTint="@color/uw1">  
    <ImageView android:layout_height="100dp"  
        android:layout_width="100dp"  
        android:src="@drawable/rectangle"  
        app:tint="@color/uw2"/>  
    ...  
</LinearLayout>  
  
.xml-file:  
<ScrollView android:id="@+id/scrollView2"  
    android:layout_width="match_parent"  
    android:layout_height="500dp"  
    app:layout_constraintTop_toBottomOf="@+id/textView"  
    app:layout_constraintBottom_toBottomOf="parent"  
    tools:layout_editor_absoluteX="0dp">  
    <include layout="@layout/rectangle_scroll_view" />  
</ScrollView>
```



# End of the Chapter



Any further questions?



# Android 2

App Lifecycle  
Fragments

U

CS 349

March 22



U

CS 349

# App Lifecycle

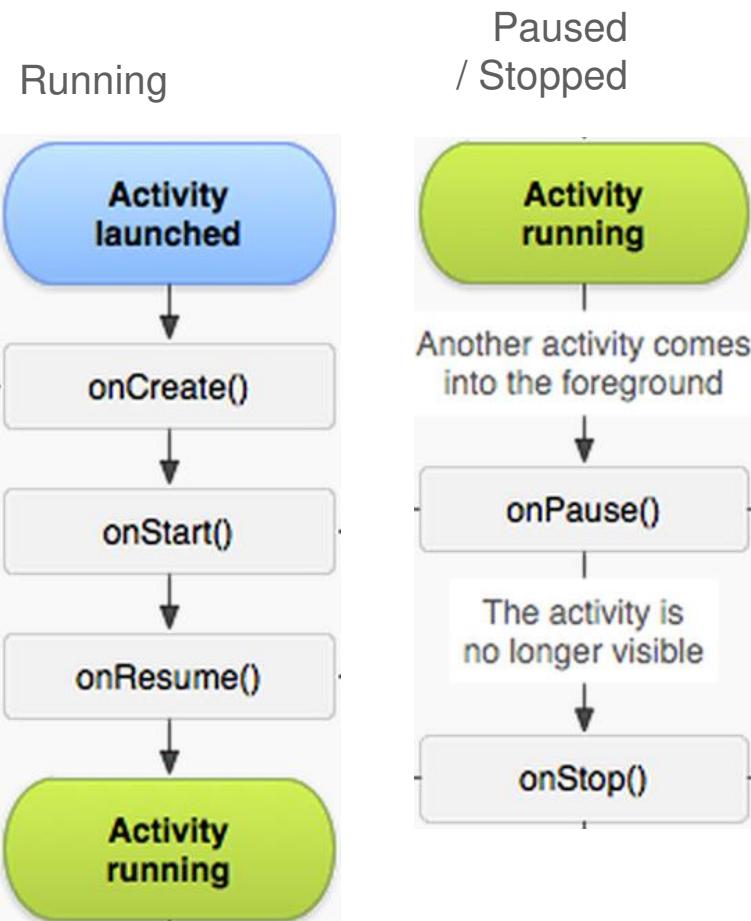


# Activity Lifecycle

Activities transition through a sequence of lifecycle states.

When a new state is entered, a corresponding callback method is called:

- e.g., `onCreate()` is called when Activity is first launched
- e.g., `onPause()` is called when another Activity comes into the foreground



# Activity Destruction and Application State

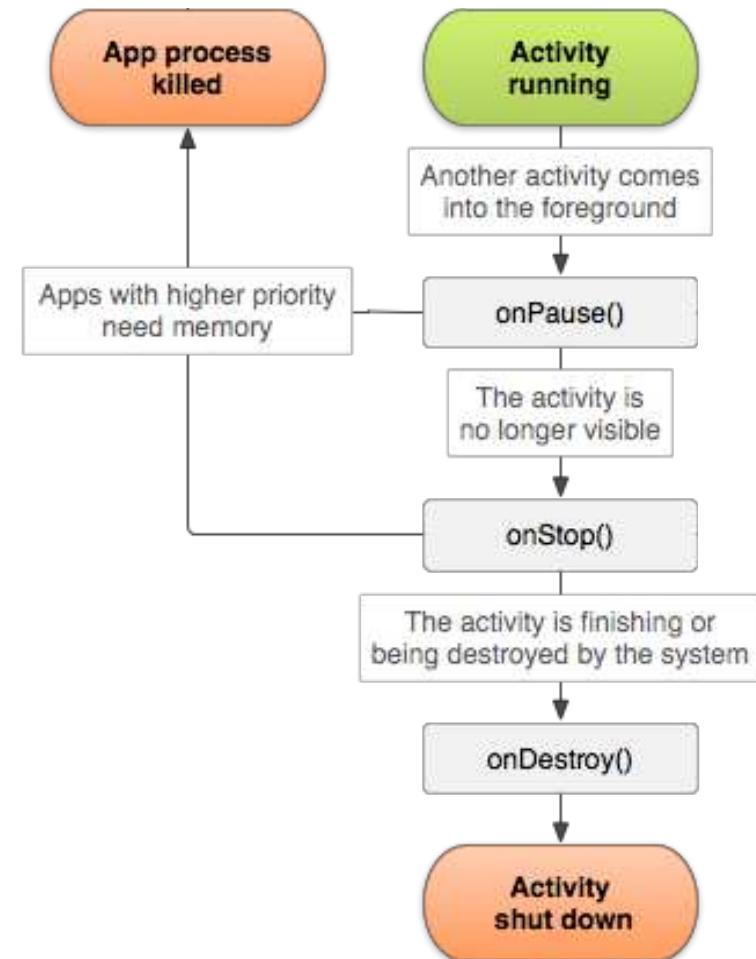
Activity can stop at any time:

- due to user-initiated actions, e.g.,
  - pressing back button
  - “configuration change”, like rotate device
- due to system-initiated actions, e.g.,
  - reclaim resources

Returning to a destroyed Activity causes it to be created again.

If an app's Activities are destroyed, the system may kill Application state.

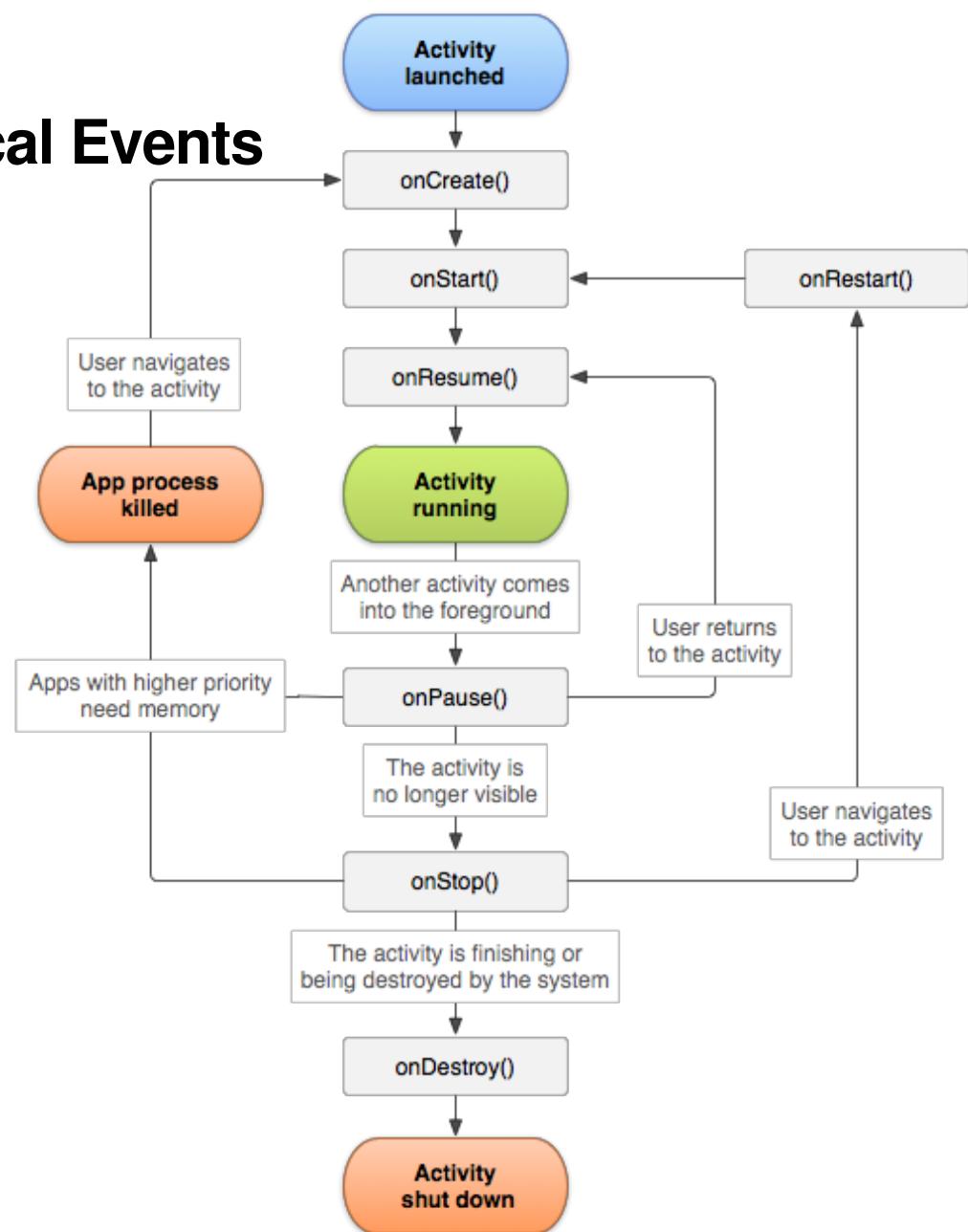
Persisting application state can be challenging.



# Activity Lifecycle and Typical Events

Core callback functions:

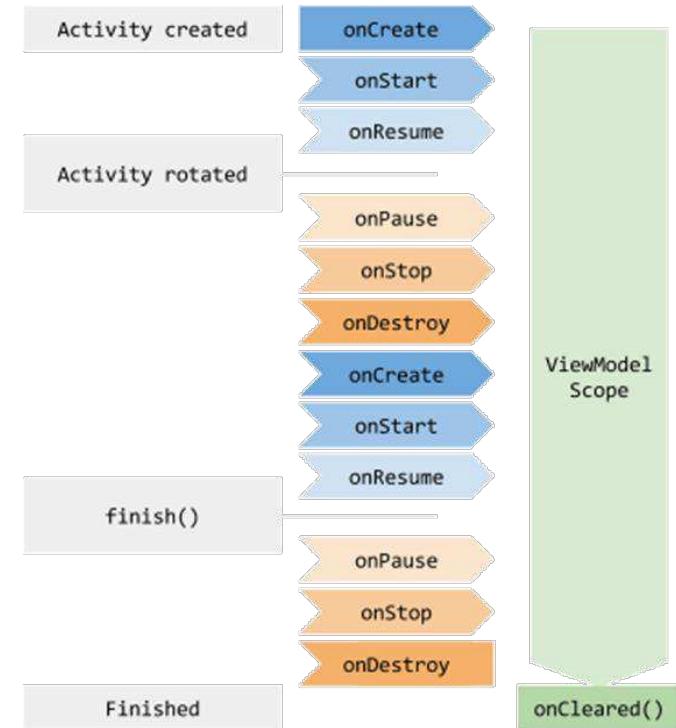
- `onCreate()`: app created or launching
- `onStart()`: becomes visible to user
- `onResume()`: prior to user interaction
- `onPause()`: loses focus or background
- `onStop()`: no longer visible to user
- `onDestroy()`: being recycled and freed



# Options for Preserving State

## ViewModel

- retains data while user is actively using app
- saved in memory



## Saved Instance State

- to recreate activity destroyed by system
- save data to “disk” (using lightweight “bundle”)

## Persistent Storage

- retains data as long as app is installed
- save data to “disk” (using heavyweight storage, database, etc.)

# Options for Preserving State – MVVM

## MyViewModel.kt:

```
class MyViewModel : ViewModel() {

    data class Contact(val name: String, val cell: String)
    private var myData = MutableLiveData<Contact>()

    init {
        // get data from model
        myData.value = Contact("Adrian Reetz", "+1 (800) 555-2368") // faking model
    }

    fun getContact() : LiveData<Contact> {
        return myData
    }

    fun setContact(cell: String) {
        if (myData.value != null) {
            // submit data to model
            Log.i("MVVM", myData.value!!.toString())
            myData.value = Contact("${myData.value!!.name}z", cell) // faking model
        }
    }
}
```

# Options for Preserving State – MVVM

activity\_main.xml:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView android:id="@+id/contact_name"
        android:layout_width="match_parent"
        android:layout_height="@dimen/viewHeight"
        android:autoSizeTextType="uniform" />

    <EditText android:id="@+id/contact_cell"
        android:layout_width="match_parent"
        android:layout_height="@dimen/viewHeight"
        android:inputType="phone" />

    <Button android:id="@+id/action"
        android:layout_width="match_parent"
        android:layout_height="@dimen/viewHeight"
        android:text="Confirm"/>
</LinearLayout>
```

# Options for Preserving State – MVVM

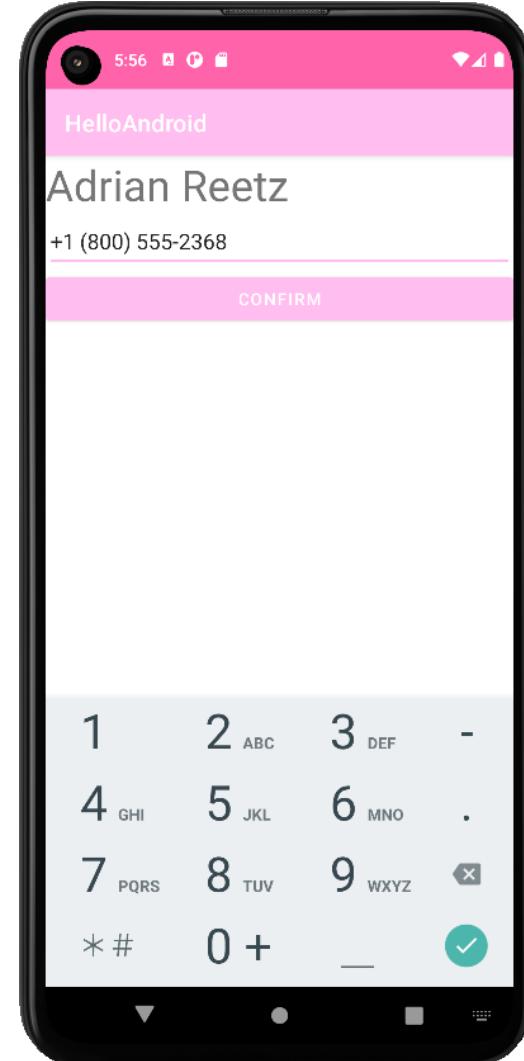
MainActivity.kt:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val myVM =
        ViewModelProvider(this)[MyViewModel::class.java]

    myVM.getContact().observe(this) {
        findViewById<TextView>(R.id.contact_name).text =
            it.name
        findViewById<TextView>(R.id.contact_cell).text =
            it.cell
    }

    findViewById<Button>(R.id.action).setOnClickListener {
        myVM.setContact(findViewById<TextView>(R.id.contact_cell).text.toString())
    }
}
```



U

CS 349

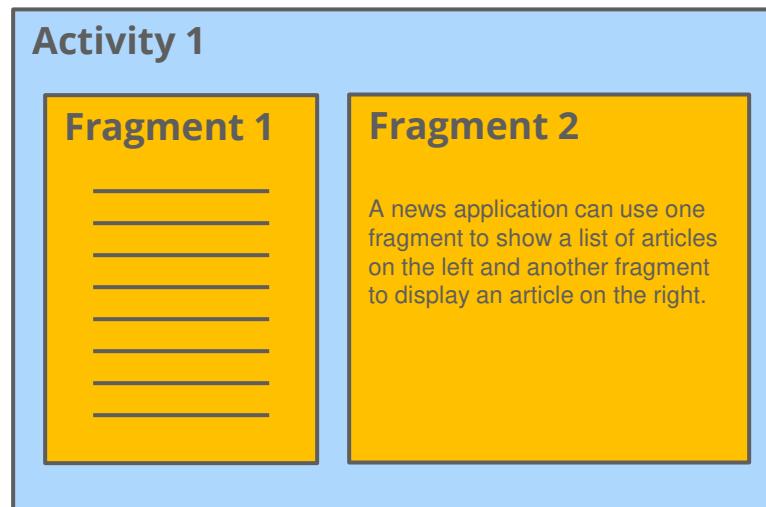
# Fragments



# Fragments

Fragments are a re-usable portions of a UI. They have their own lifecycle, and their own layout.

An activity can contain (and manage) multiple fragments. Fragments can be used like multiple Activities by swapping out root layout with different Fragments to navigate



# Tooling Configuration for Fragments

You might have to manually add these dependencies to `build.gradle`:

```
dependencies {  
    ...  
    implementation 'androidx.fragment:fragment-ktx:1.5.4'  
}
```

# Fragment Definition

A Fragment is a layout (.xml + .kt-file):

## fragment\_new.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".BlankFragment">

    <TextView android:text="@string/hello_blank_fragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@color/uw1" />

</FrameLayout>
```

## NewFragment.kt

```
class NewFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                             savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.fragment_new, container, false)
    }
}
```

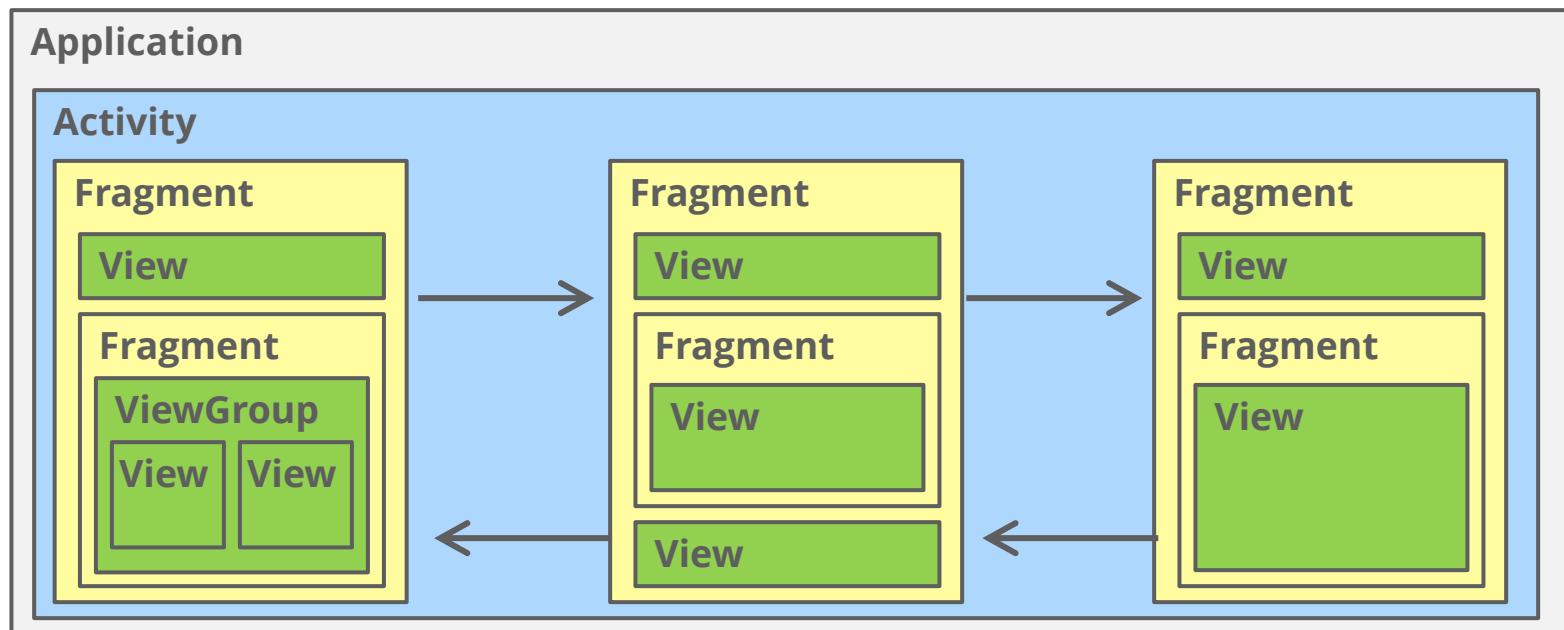
# Inserting a Fragment

```
<androidx.fragment.app.FragmentContainerView  
    android:id="@+id/fragmentContainerViewBlank"  
    android:layout_width="0dp"  
    android:layout_height="0dp"  
    android:name="ui.lectures.helloandroid.BlankFragment"  
    ... />  
  
<androidx.fragment.app.FragmentContainerView  
    android:id="@+id/fragmentContainerViewOther"  
    android:layout_width="0dp"  
    android:layout_height="0dp"  
    android:name="ui.lectures.helloandroid.OtherFragment"  
    ... />
```



# Multiple Fragment Navigation

A common way to architect Android apps is to navigate between different Fragments hosted in a single Activity. This is called “Single-Activity Architecture”:

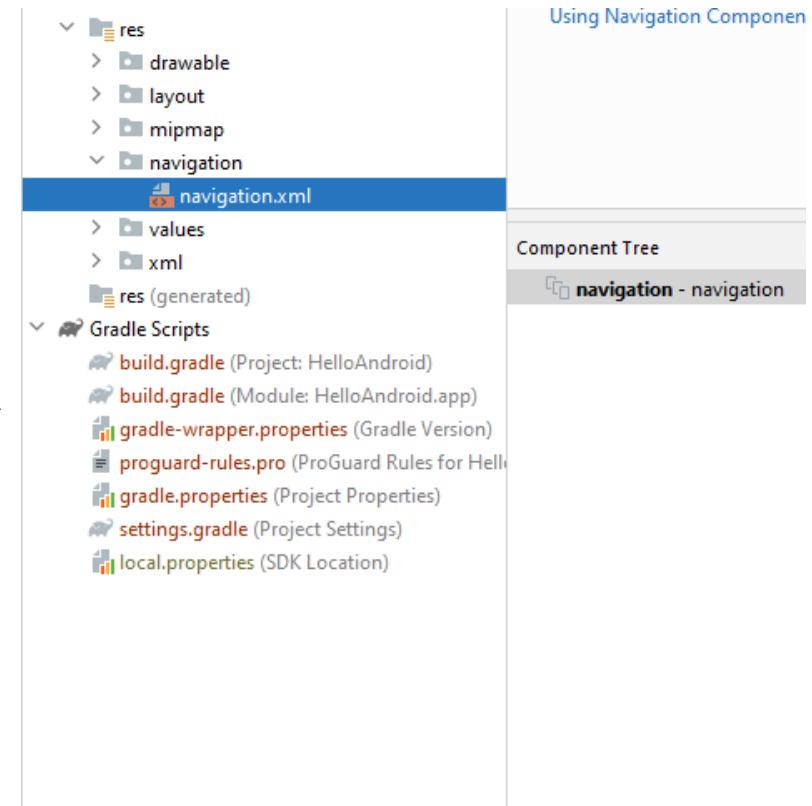
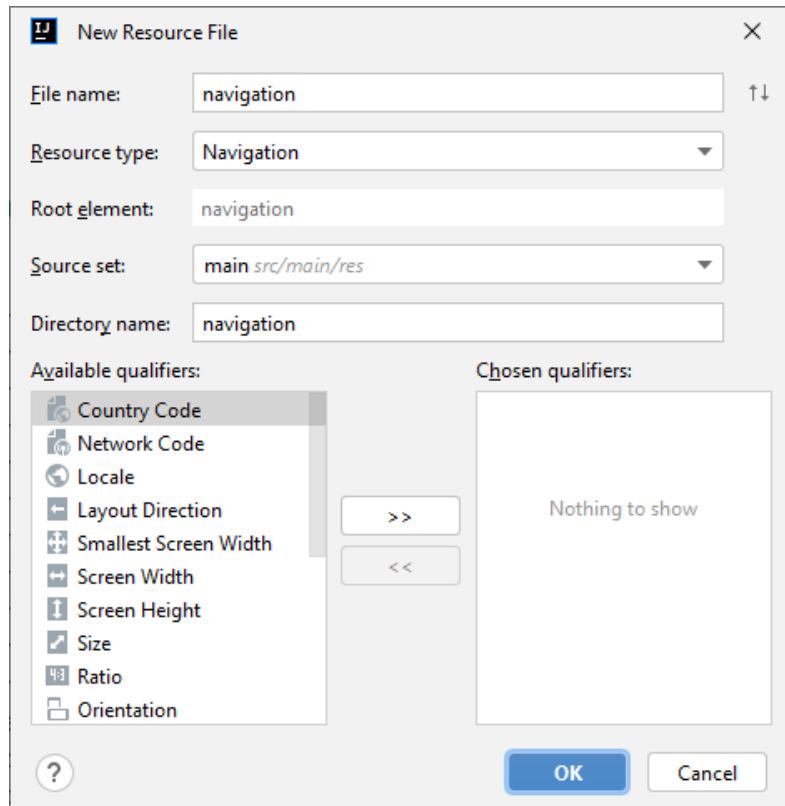


# Tooling Configuration for Fragment Navigation

You might have to manually add these dependencies to build.gradle:

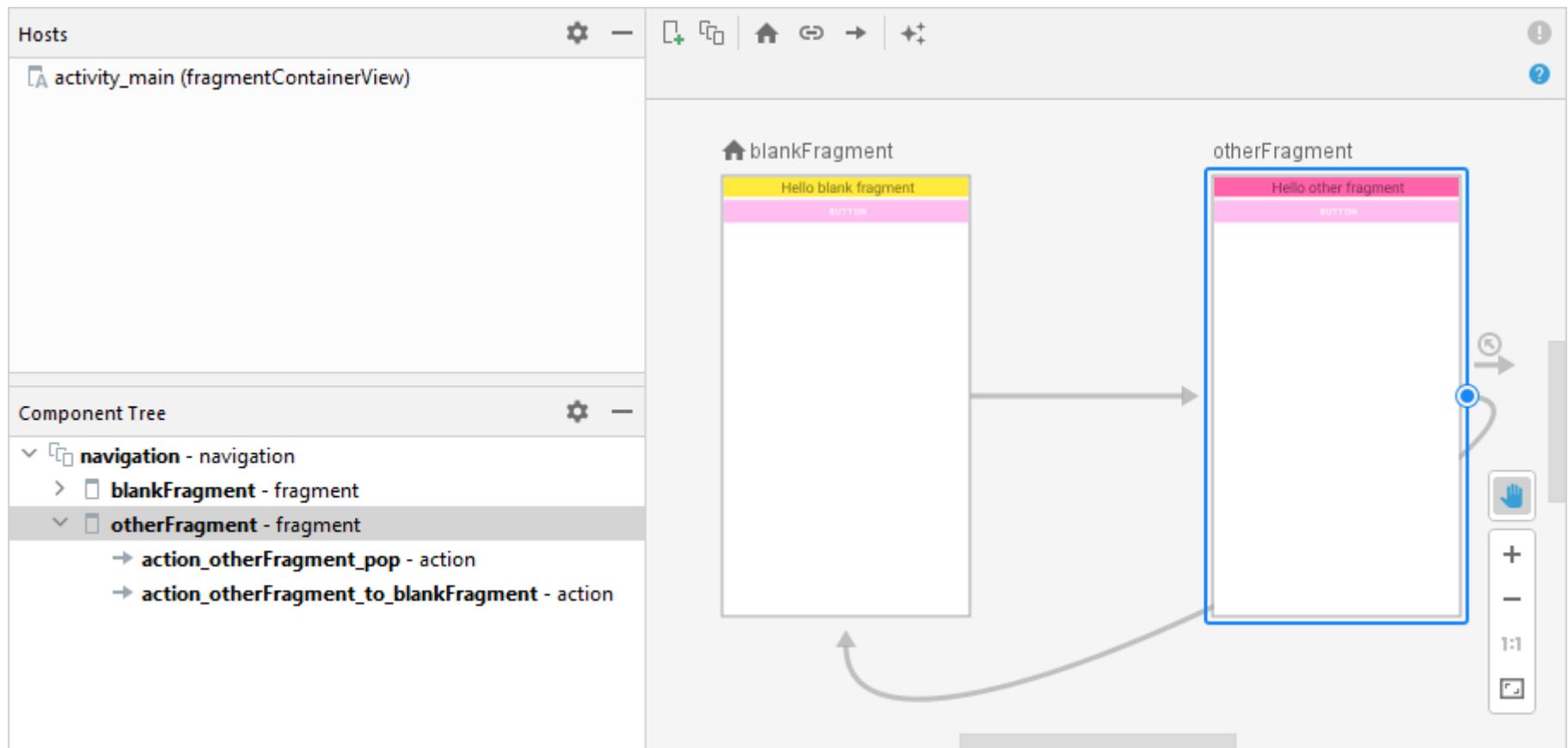
```
dependencies {  
    ...  
    implementation 'androidx.navigation:navigation-ui-ktx:2.5.3'  
    implementation 'androidx.navigation:navigation-fragment-ktx:2.5.3'  
}
```

# Add navigation Resource File



# Define Navigation

- right-click to add navigation actions
- drag navigation paths between fragments



# Define Navigation

## navigation.xml

```
<?xml version="1.0" encoding="utf-8"?>
<navigation android:id="@+id/navigation"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    app:startDestination="@+id/blankFragment">

    <fragment android:id="@+id/blankFragment"
        android:name="ui.lectures.helloandroid.BlankFragment"
        android:label="fragment_blank"
        tools:layout="@layout/fragment_blank">
        <action android:id="@+id/action_blankFragment_to_otherFragment"
            app:destination="@+id/otherFragment" />
    </fragment>
    <fragment android:id="@+id/otherFragment"
        android:name="ui.lectures.helloandroid.OtherFragment"
        android:label="fragment_other"
        tools:layout="@layout/fragment_other">
        <action android:id="@+id/action_otherFragment_to_blankFragment"
            app:destination="@+id/blankFragment"/>
        <action android:id="@+id/action_otherFragment_pop"
            app:popUpTo="@+id/otherFragment"
            app:popUpToInclusive="true"/>
    </fragment>
</navigation>
```

# Define Navigation

The property `app:destination` navigates to a new fragment:

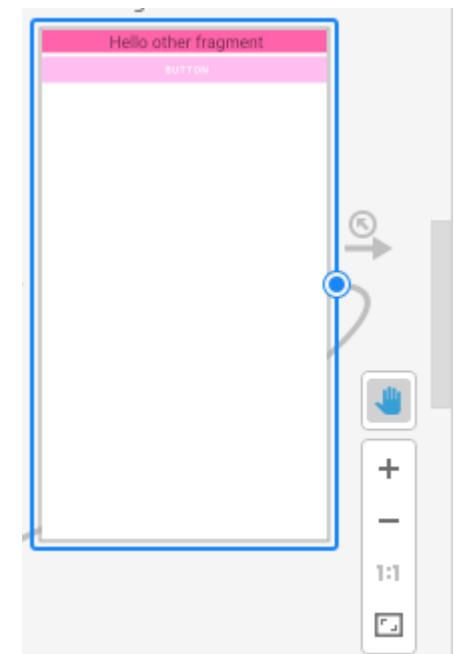
```
<action android:id="@+id/action_otherFragment_to_blankFragment"  
        app:destination="@+id/blankFragment" />
```

- calling Fragment is saved in back stack
- back button remembers path and will return to calling Fragment

The property `app:popUpTo` navigates to the previous fragment:

```
<action android:id="@+id/action_otherFragment_pop"  
        app:popUpTo="@+id/otherFragment"  
        app:popUpToInclusive="true" />
```

- calling Fragment is "popped off" the back stack
- back button will not return to calling Fragment



# Use Navigation

Instead of including a single fragment via `FragmentContainerView`, include a `NavHostFragment`, which will add the necessary fragment container.

## `activity_main.xml`

```
<FrameLayout android:layout_width="match_parent"
    android:layout_height="match_parent"
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragmentContainerView"/>
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:navGraph="@navigation/navigation"
    app:defaultNavHost="true"
</FrameLayout>
```

# Use Navigation

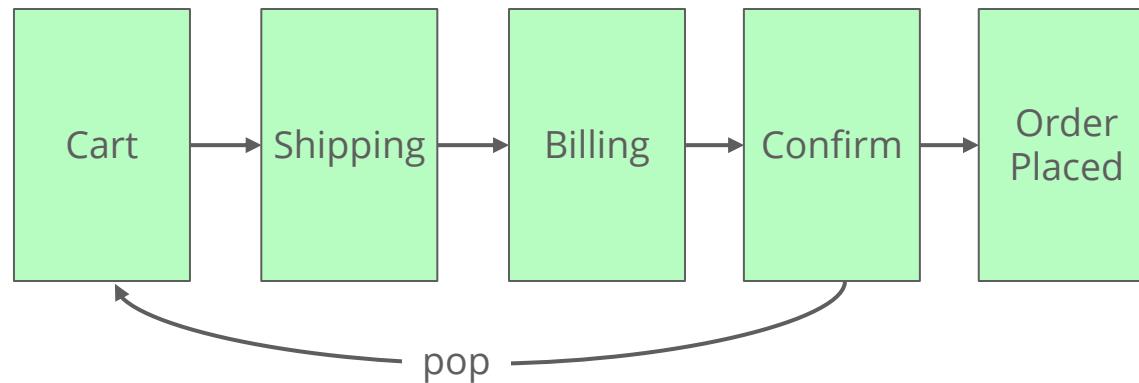
Instead of including a single fragment via `FragmentContainerView`, include a `NavHostFragment`, which will add the necessary fragment container.

## BlankFragment.kt

```
override fun onCreateView(..): View? {
    val root = inflater.inflate(R.layout.fragment_main, container, false)
    root.findViewById<Button>(R.id.goToOtherButton).apply {
        setOnClickListener {
            findNavController().navigate(R.id.otherFragment)
        }
    }
    return root
}
```

# Navigation and the "Back Stack"

By default, Android's back / up buttons returns to previous Fragment stored on a back stack, which tracks Fragment navigation:



Some app structures need to control back stack behaviour, e.g., cancelling in the middle of an ordering process

Solution: set "Pop Behavior" on actions, e.g., navigate to cart but "pop off" past ordering fragments.

# Exchanging Information Between Fragments – MVVM

The ViewModel can be used to transfer information between fragments. The ViewModel belongs to an activity but can be shared amongst fragments.

## ViewModel.ks

```
class MyViewModel : ViewModel() {  
    // ...  
}
```

In an activity (e.g., MainActivity), the ViewModel can be retrieved using:

```
val myVM = ViewModelProvider(this)[MyViewModel::class.java]
```

In a fragment, the ViewModel can be retrieved using:

```
val myVM = ViewModelProvider(requireActivity())[MyViewModel::class.java]
```

# Exchanging Information Between Fragments – MVVM

## BlankFragment.kt

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                           savedInstanceState: Bundle?): View? {
    val view = inflater.inflate(R.layout.fragment_blank, container, false)

    view.findViewById<Button>(R.id.goToOther).setOnClickListener {
        findNavController().navigate(R.id.action_blankFragment_to_otherFragment)
    }

    val myVM = ViewModelProvider(requireActivity())[MyViewModel::class.java]
    myVM.getContact().observe(viewLifecycleOwner) {
        view.findViewById<TextView>(R.id.viewer).text =
            "${it.name}: ${it.cell}"
    }

    return view
}
```

# Exchanging Information Between Fragments – MVVM

## OtherFragment.kt

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                           savedInstanceState: Bundle?): View? {
    val view = inflater.inflate(R.layout.fragment_other, container, false)

    val myVM = ViewModelProvider(requireActivity())[MyViewModel::class.java]

    view.findViewById<Button>(R.id.submit).setOnClickListener {
        myVM.setContact(view.findViewById<EditText>(R.id.editor).text.toString())
    }

    myVM.getContact().observe(viewLifecycleOwner) {
        view.findViewById<EditText>(R.id.editor).text = it.cell
    }

    return view
}
```

# Exchanging Information Between Fragments – Bundles

Bundles are mappings between String keys and other data:

## BlankFragment.kt

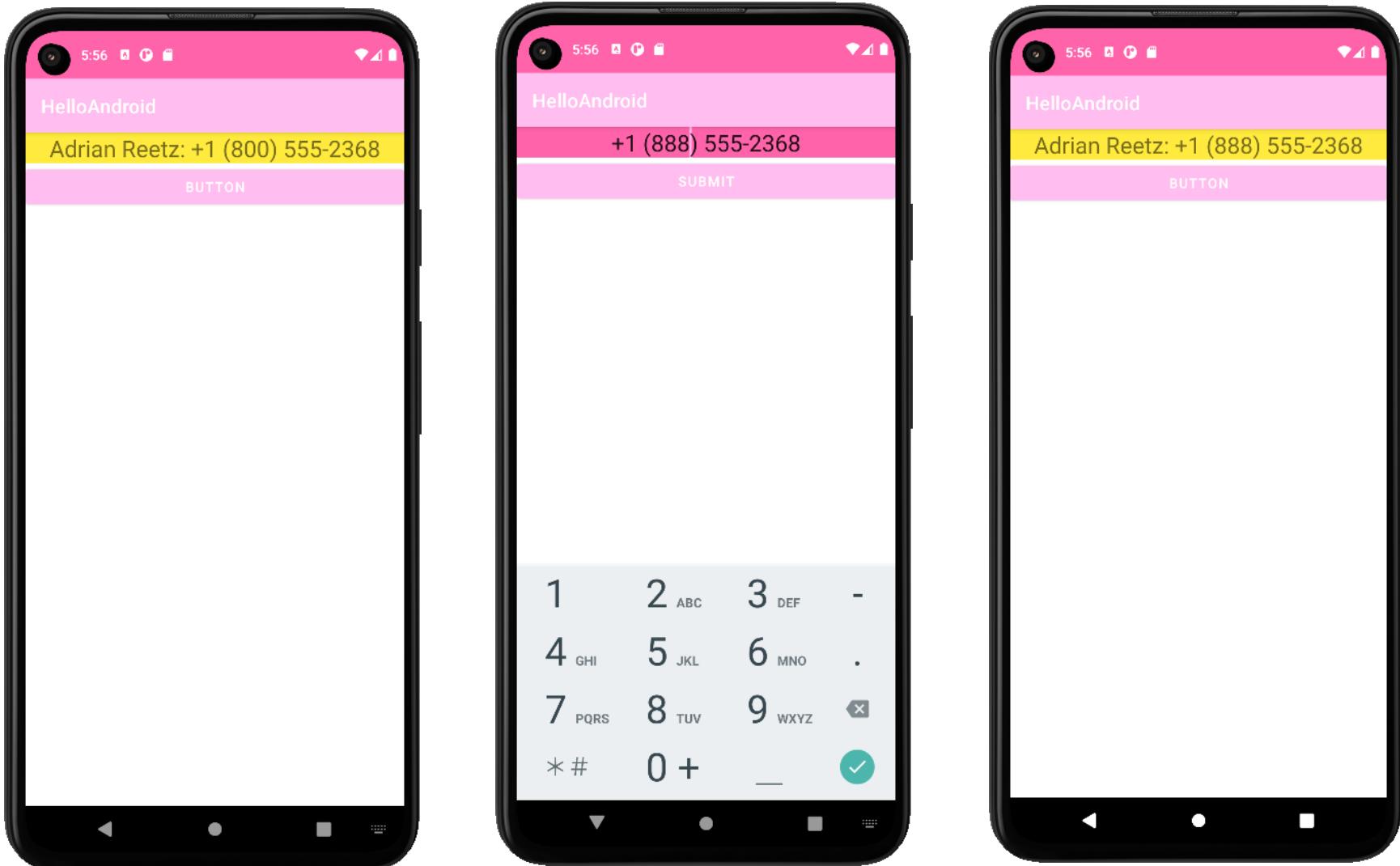
```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,  
                         savedInstanceState: Bundle?): View? {  
    val fragment = inflater.inflate(R.layout.fragment_blank, container, false)  
  
    fragment.findViewById<Button>(R.id.goToOther).setOnClickListener {  
        findNavController().navigate(R.id.fragment_other,  
            Bundle().apply {  
                putString("key1", "value")  
               .putInt("key2", 0)  
            })  
    }  
    return fragment  
}
```

# Exchanging Information Between Fragments – Bundles

## OtherFragment.kt

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,  
                         savedInstanceState: Bundle?): View? {  
    val view = inflater.inflate(R.layout.fragment_other, container, false)  
    val key1 = requireArguments().getString("key1") } // String?  
    val key2 = requireArguments().getInt("key2") } // Int  
  
    return view  
}
```

# Exchanging Information Between Fragments – MVVM



# End of Section



- How to implement MVVM in Android.
  - Use `ViewModel`
  - Use `LiveData / MutableLiveData`
  - Prevent data mutation from outside the model
- How to navigate between fragments



Any further questions?

-



# Responsiveness

Human Perception and Expectations

Performing CPU-intensive Tasks

U

CS 349

March 27



U | CS 349

# Human Perception and Expectations



Steve Souders' , "The Illusion of Speed" (Velocity 2013)

<https://www.youtube.com/watch?v=bGYgFYG2Ccw>

# Design Implications

Time to prepare for conscious cognition task: Display a fake version of an application interface, or image of document on last save, while the real one loads in less than 10s

Meet The Canon Cat.  
A new breed of office machine.

The Canon Cat is the world's first Work Processor. It is a simple but powerful office machine.

The Canon Cat is not a typewriter, electronic or otherwise. Yet minutes after you plug it in you can be typing on The Cat like a veteran.

The Canon Cat is not a word processor. Yet it will let you write and edit as well as the most sophisticated word processor.

The Canon Cat is not a personal computer. Yet it will let you do calculations right in the text, store information and communicate with other office machines.

What is The Cat? As we said, it's the world's first Work Processor. It can help you write and edit, communicate and calculate. It'll even dial your phone.

**The great leap forward.**

The Canon Cat is the brainchild of the man who originated The Macintosh Computer. This time he wanted to make, not a computer, but an office appliance so simple that anyone could plug it in and use it—like a toaster.

He analyzed how people (particularly office people) think and work. Then he designed The Cat to work the way people think. Which makes work easier.

For example, if you look over the keyboard on this page you'll see that it is just like a traditional typewriter keyboard. No fancy computer keys with confusing names like Access and Control. It's familiar and easy to use.

See these royal Leap keys<sup>™</sup>. They are The Cat's most fascinating feature. Just press one down, type in a few letters you remember from a document—and you're where you want to be—instantly. No menus. No files. No mouse.

A most productive pet.

If you're ready to move up from typewriters to the world of microchips and screens, The Canon Cat is for you.

It's from Canon, home of a long tradition of office innovations, including personal copiers and desktop laser beam printers. The Cat has been designed to work especially well with Canon Printers including The Cat180 Daisy Wheel Printer and the Canon Laser Beam Printer.

The Cat is so easy to learn that you or your employees won't have to disappear for days into often frustrating training sessions.

Anyone can become an expert on The Cat in just a few hours.

The Cat is most affordable.

The Canon Cat will make mountains of work disappear faster and easier than ever before.

That's why we call it the world's first WORK Processor.

Macintosh is a trademark of Apple Computer, Inc.

**Canon Cat**  
The Advanced WORK Processor

# Responsive User Interfaces

A responsive UI delivers feedback to the user in a timely manner

- It does not make the user wait any longer than necessary
- It communicates state of long tasks to the user

We can make a UI responsive in two ways:

- Designing to meet human expectations and perceptions
- Loading data efficiently while maintaining interactivity

# What factors affect responsiveness?

## User Expectations

- how quickly a system “should” react to complete some task
- expectations for technology (e.g., web app vs. native desktop)

## Application and Interface Design

- the interface keeps up with user actions
- the interface informs the user about application status
- the interface doesn’t make users wait unexpectedly

Responsiveness is the most important factor in determining user satisfaction, more so than ease of learning, or ease of use.

Responsiveness is not just system performance!

# **Responsive User Interfaces**

Slow Performance can still be responsive:

- Provide feedback to confirm user actions
- Provide feedback about progress
- Allow users to perform other tasks while waiting
- Perform low-priority system tasks in the background

# Responsive User Interfaces

Provide feedback to confirm user actions, e.g.:

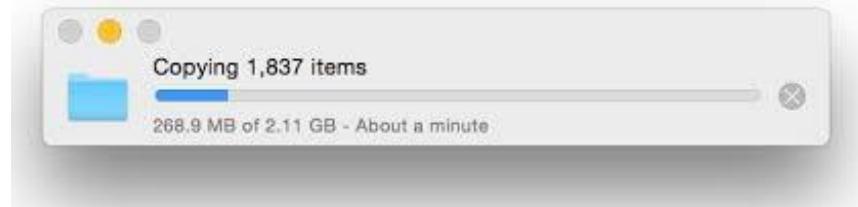
- Busy indicators

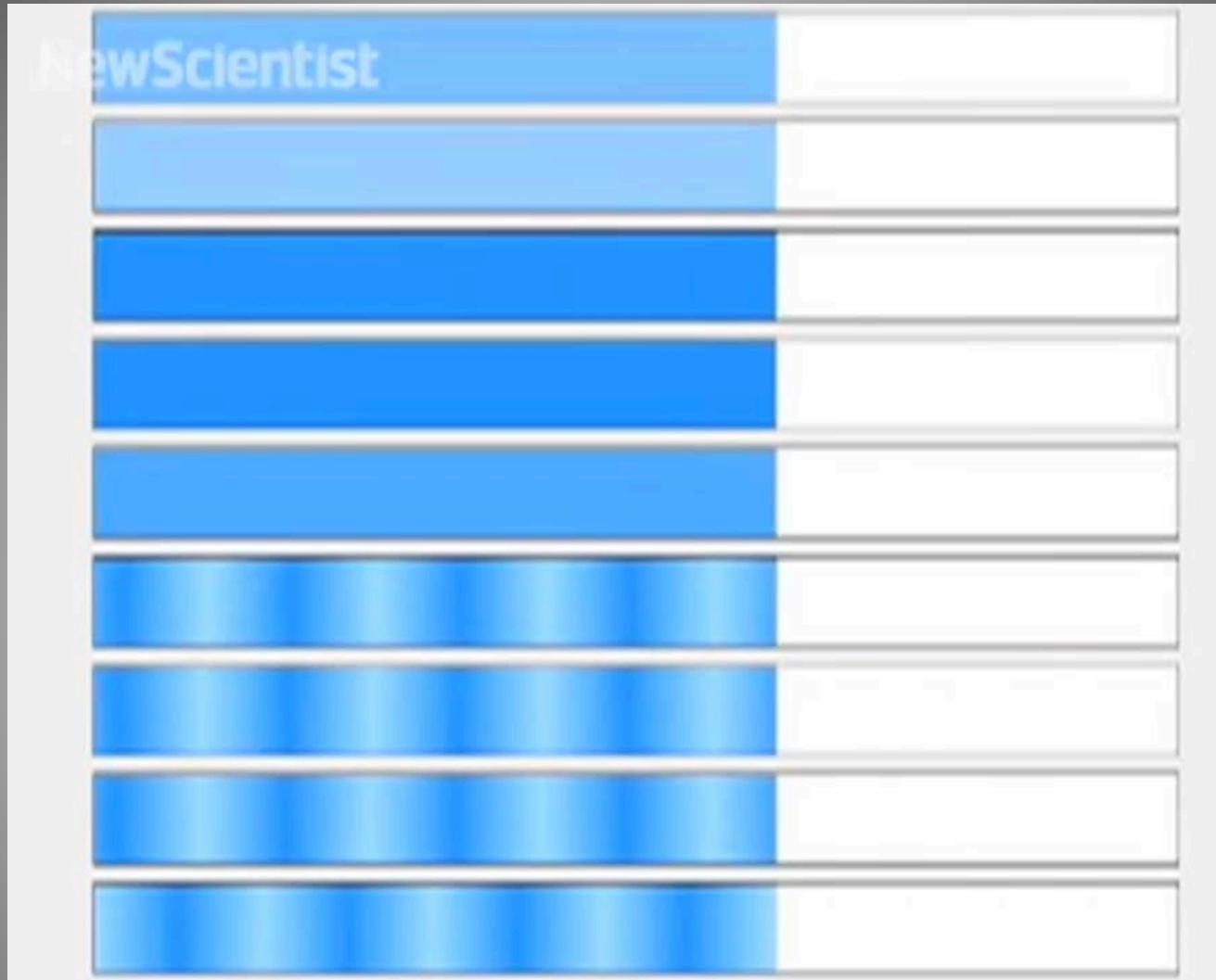


# Responsive User Interfaces

Provide feedback about progress, e.g., progress bars:

- Show work remaining, not work completed
- Show total progress when multiple steps, not only step progress
- Display finished state (e.g., 100%) very briefly at the end
- Show smooth progress, not erratic bursts
- Use human precision, not computer precision: “74.5 seconds remaining” (bad), “About a minute” (good)



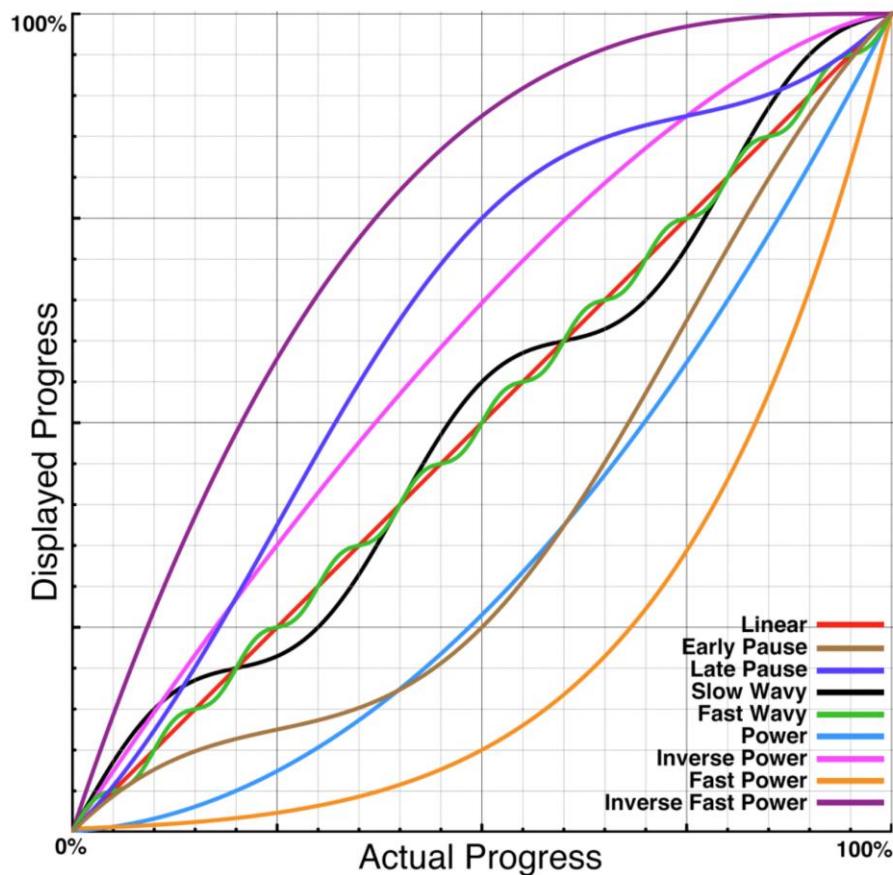


Harrison et al. Faster Progress Bars (2010)

<https://www.newscientist.com/article/dn18754-visual-tricks-can-make-downloads-seem-quicker/>

# Changing Perception of Progress Bars

Change how actual progress maps to displayed progress



positive or negative  
perception of time  
measured in user study

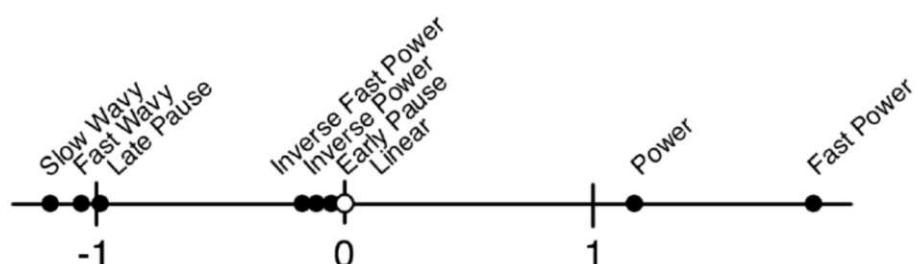


Figure 4: Number line showing relative distances from linear, which is centered at 0. Values generated from logistic regression model.

# Responsive User Interfaces

Allow users to perform other tasks while waiting, e.g.:

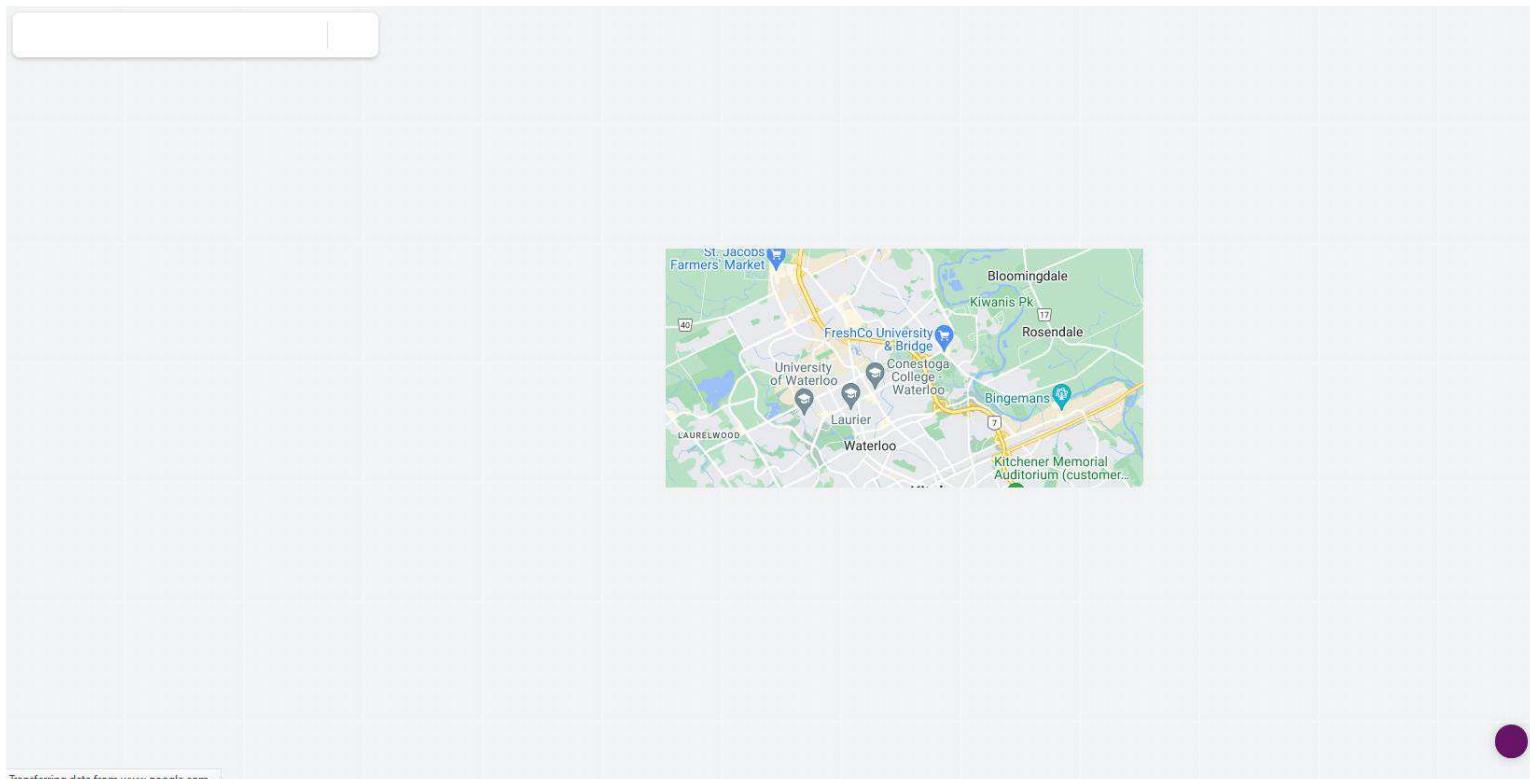
- Skeleton content



# Skeleton Content

Provide user with a minimal version of an interface or data while the rest is being rendered or loaded. This can be generic layout or minimal version of actual content.

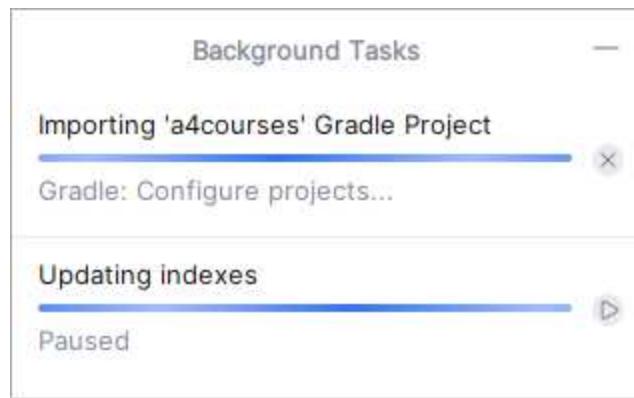
- Users adjust to a layout they will eventually see
- Users can act upon the limited amount of data they are already seeing
- Loading process seems faster because there is an initial result



# Responsive User Interfaces

Perform low-priority system tasks in the background, e.g.:

- Background Tasks



U | CS 349

# Performing CPU-intensive Tasks

# Handling “Long Running Tasks” in a User Interface

How do we handle tasks that take a significant amount of time to execute?

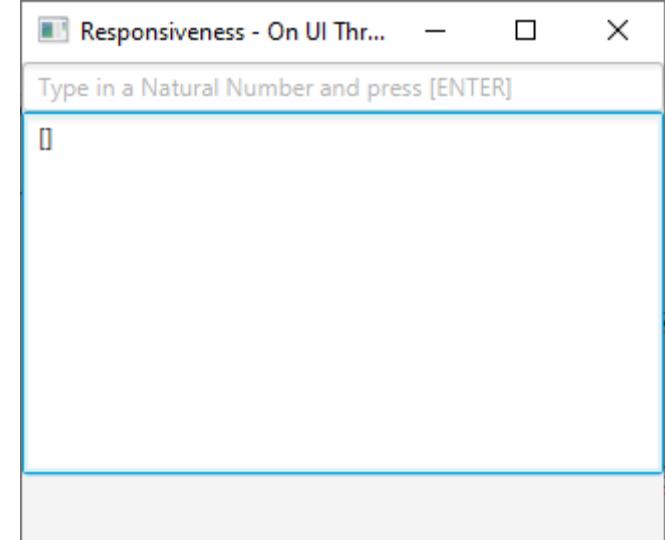
## Goals

- keep UI responsive
- provide progress feedback
- allow long task to be paused or canceled

Do these even if it takes a bit longer to complete the long task.

# Example: Finding primes

```
override fun start(stage: Stage) {  
    val primer = Primer()  
    val result = TextArea("").apply {  
        textProperty().bind(Bindings.concat(primer))  
        isWrapText = true  
    }  
    val progress = ProgressBar(0.0).apply {  
        prefWidth = Double.MAX_VALUE  
    }  
    val input = TextField().apply {  
        promptText = "Type in a Natural Number and press [ENTER]"  
        onKeyPressed = EventHandler {  
            if (it.code == KeyCode.ENTER) {  
                text.toIntOrNull()?.apply {  
                    primer.findPrimes(this, progress)  
                }  
            }  
        }  
    }  
    stage.apply {  
        title = "Responsiveness - Subtasking"  
        scene = Scene(VBox(input, result, progress), 320.0, 240.0)  
    }.show()  
}
```



# Blocking Calls

Performs calculations in a loop on the UI thread.

```
private fun Int.isPrime() : Boolean {
    Thread.sleep(100) // Simulating a task with longer runtime
    return (2 .. sqrt(toDouble()).toInt()).count { this % it == 0 } == 0
}

class Primer : SimpleListProperty<Int>(FXCollections.observableArrayList()) {

    fun findPrimes(end: Int) {
        value.clear()
        (2 .. end).forEach {
            if (it.isPrime()) add(it)
        }
    }
}
```

Not a feasible solution.

# Implementation Approaches for Long Tasks

Alternative solutions are

- Subtasking (on UI thread): periodically execute subtasks between handling UI events
- Multi-threading (on one worker thread): use thread-safe API to communicate between worker and UI thread
- Multi-threading (on multiple worker threads): use thread-safe API to communicate between workers and UI thread

# Subtasking

Splitting the complete task into multiple subtasks. Subtasks run for limited time, thus giving the UI the opportunity to react to user input.

```
class Primer : SimpleListProperty<Int>(FXCollections.observableArrayList()) {  
    private val progressProperty = SimpleDoubleProperty(0.0)  
    private fun findPrimesWorker(start: Int, end: Int, dur: Long) {  
        Platform.runLater {  
            val time = GregorianCalendar().time.time  
            var current = start  
            while ((current <= end) and (GregorianCalendar().time.time - time < dur)) {  
                if (current.isPrime()) add(current)  
                ++current  
            }  
            progressProperty.value = (current - 1.0) / end  
            if (current <= end) findPrimesWorker(current, end, timeInMs)  
        }  
    }  
    fun findPrimes(end: Int, progressBar: ProgressBar? = null, timeInMs: Long = 100L) {  
        progressBar?.progressProperty()?.bind(progressProperty)  
        value.clear()  
        findPrimesWorker(1, end, timeInMs)  
    }  
}
```

# Subtasking

## Advantages:

- Can handle “pausing” (stopping / restarting) task because it maintains information on progress of overall task.
- Useful in single-threaded platforms (e.g., microcontroller)

## Disadvantages:

- Still blocking the UI thread
- Not all tasks can easily break down into subtasks

# Multithreading

Creating one additional worker thread to the UI thread and let a complex task run on that thread. Communication between the worker and the UI threads have to be synchronized.

```
class Primer : SimpleListProperty<Int>(FXCollections.observableArrayList()) {  
  
    fun findPrimes(end: Int, progressBar: ProgressBar? = null) {  
        clear()  
        val worker = object: Task<Unit>() {  
            override fun call() {  
                (1 .. end).forEach {  
                    if (it.isPrime()) Platform.runLater { add(it) }  
                    updateProgress(it.toLong(), end.toLong())  
                }  
            }  
        }  
        progressBar?.progressProperty()?.bind(worker.progressProperty())  
        Executors.newSingleThreadExecutor().apply {  
            execute(worker)  
        }  
    }  
}
```

# Multithreading

Creating multiple additional threads to the UI thread, split up a complex task, and let each subtask run on a separate thread. Communication between the worker and the UI threads have to be synchronized.

```
class Primer : SimpleListProperty<Int>(FXCollections.observableArrayList()) {  
  
    private val progress = SimpleIntegerProperty(0)  
  
    fun findPrimes(end: Int, progressBar: ProgressBar? = null, threadCount) {  
        clear()  
        progressBar?.progressProperty()?.bind(Bindings.createDoubleBinding  
            ({ progress.value.toDouble() / end }, progress))  
        Executors.newFixedThreadPool(threadCount).apply {  
            (0 until threadCount).forEach {  
                submit {  
                    ((1.0 + end * it / threadCount).toInt() ..  
                     (end * (it + 1.0) / threadCount).toInt()).forEach {  
                        if (it.isPrime()) Platform.runLater { add(it) }  
                        Platform.runLater { ++progress.value }  
                    }  
                }  
            }  
        }.shutdown()  
    }  
}
```

# Multithreading

Multi-threading: manage multiple concurrent threads with shared resources but executing different instructions.

Threads are a way to divide computation, reduce blocking.

Concurrency has risks: what if two threads update a variable?

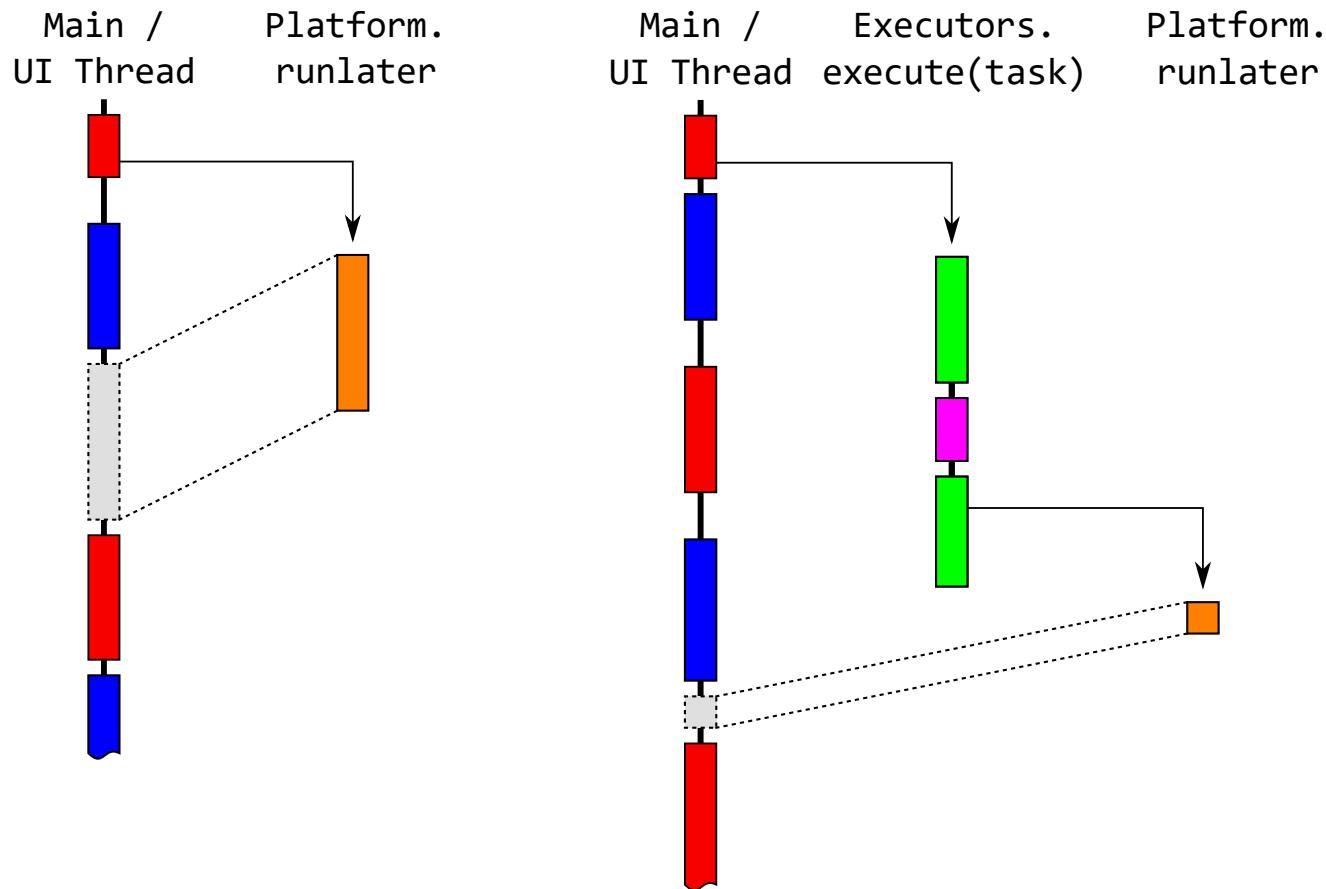
Typically, three types of threads in a UI application:

- One main application thread: access to the scene graph must occur from here
- One render thread: handling of current and calculation of future frame
- 0 or more worker threads (also called “background threads”)

# Thread-Safety

Most UI toolkits (like JavaFX or Swing) are not thread safe.

Transitioning from a worker thread back to the UI thread requires extra care:



# Multithreading

Advantages:

- Conceptually, easiest to implement
- Takes advantage of multi-core architectures

Disadvantages:

- Need to be careful about inter-thread communication
- All the usual threading caveats: race conditions, deadlocks, ...

# End of the Chapter



Any further questions?

-



X

# Undo-Redo

Principles and Concepts

Undo Patterns

Implementation

U

CS 349

U  
CS 349

# Principles and Concepts

# Benefits of Undo / Redo

Undo lets you **recover from errors**

- input errors (human) and interpretation errors (computer)
- you can work quickly (“without fear”)

Undo enables **exploratory learning**

- try things you don’t know the consequences of  
(without fear or commitment)
- try alternative solutions  
(without fear or commitment)

Undo lets users **evaluate modifications**

- fast do-undo-redo cycle to evaluate last change to document

Unless stated otherwise, “undo” means “undo / redo” in these slides.

# Checkpointing

A manual undo method

- you save the current state so you can rollback later (if needed)

Consider a video game ...

- You kill a monster
- You save the game
- You try to kill the next monster
- You die
- You reload the saved game
- You try to kill the next monster
- You kill the monster
- You save the game



# Undo Design Choices

As a designer, you need to consider the following:

- Undoable Actions: what actions should (or can) be undone?
- State restoration: what part of UI is restored after undo?
- Granularity: how much should be undone at a time?
- Scope: is undo global, local, or someplace in between?

# Undoable Actions

Some actions may be omitted from undo:

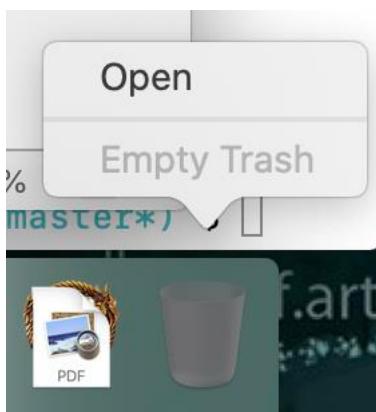
- Change to selection? Window resizing? Scrollbar positioning?

Some actions are destructive and not easily undone

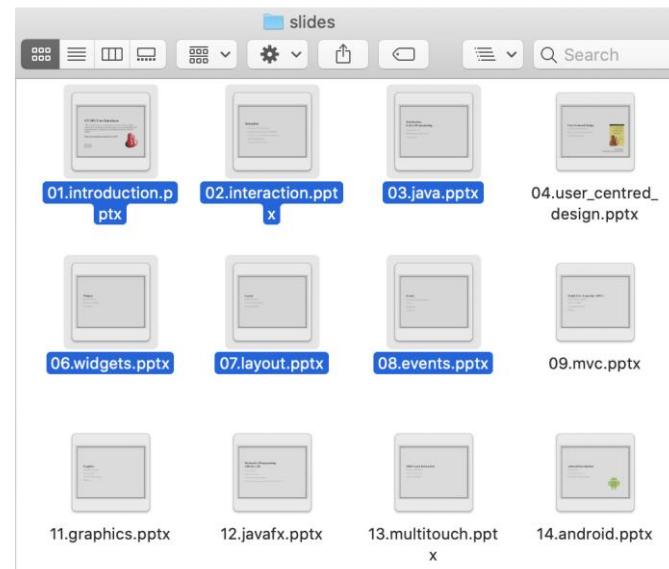
- Quitting program with unsaved data, emptying trash

Some actions cannot be undone

- Printing



*Users typically cannot undo emptying the trash.*



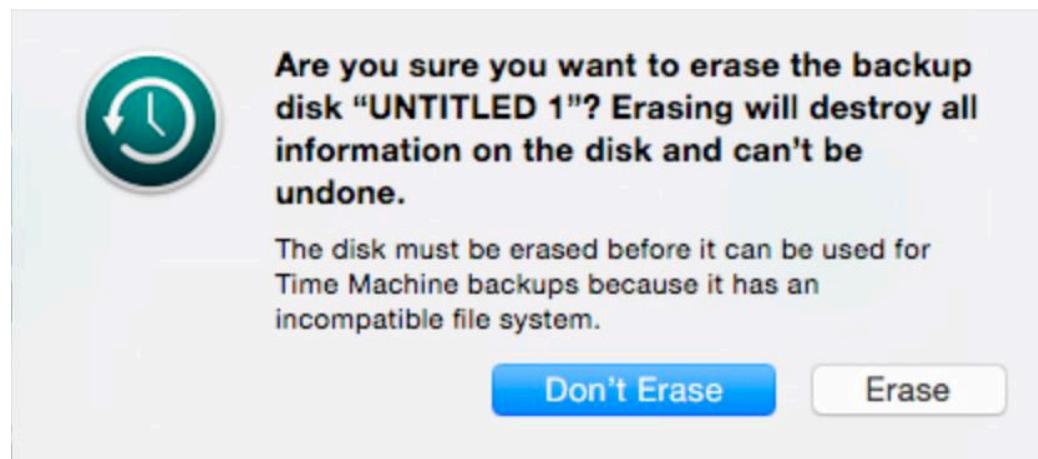
*Can you undo selection of files in a window?*

# Undoable Actions

All changes to document (i.e., the Model) should be undoable.

Changes to the View(i.e., the document's interface) should be undoable only if they are tedious or require significant effort. Typically view changes are not undoable.

Ask for confirmation before doing a destructive action which cannot easily be undone.



# State Restoration

What is the user interface state after an undo or redo?

- e.g., highlight text, delete, undo: is text highlighted?
- e.g., select file icon, delete, undo: is file icon selected?

User interface state should be meaningful after undo/redo action

- Change selection to object(s) changed as a result of undo/redo
- Scroll to show selection, if necessary
- Give focus to the control that is hosting the changed state

These provide additional undo feedback.

# **Granularity**

Undo one chunk: the conceptual change from one document state to another.

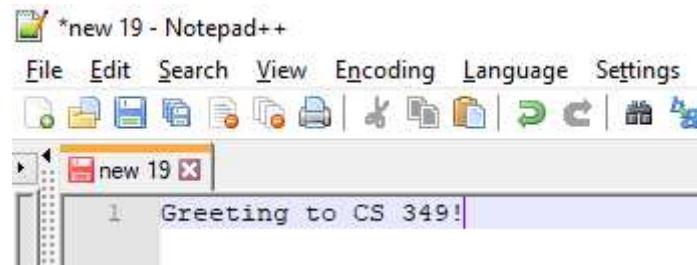
# Granularity – Text

Undo one chunk:

- One typed character?
- One word? (Next problem: what is a word?)
- One time-interval? (Next problem: how long should that be?)
- How to deal with auto-correct?

In Word: typed    auto-correct    undo

This.is¶      This.is¶      This.is¶



# Granularity – Drawing

Undo one chunk:

- Per Pixel?
- Between strokes?
- Between MousePress and MouseRelease?
- How to deal with window switching?

-



X

# Undo Patterns

U

CS 349

# Undo Patterns

Two possible approaches:

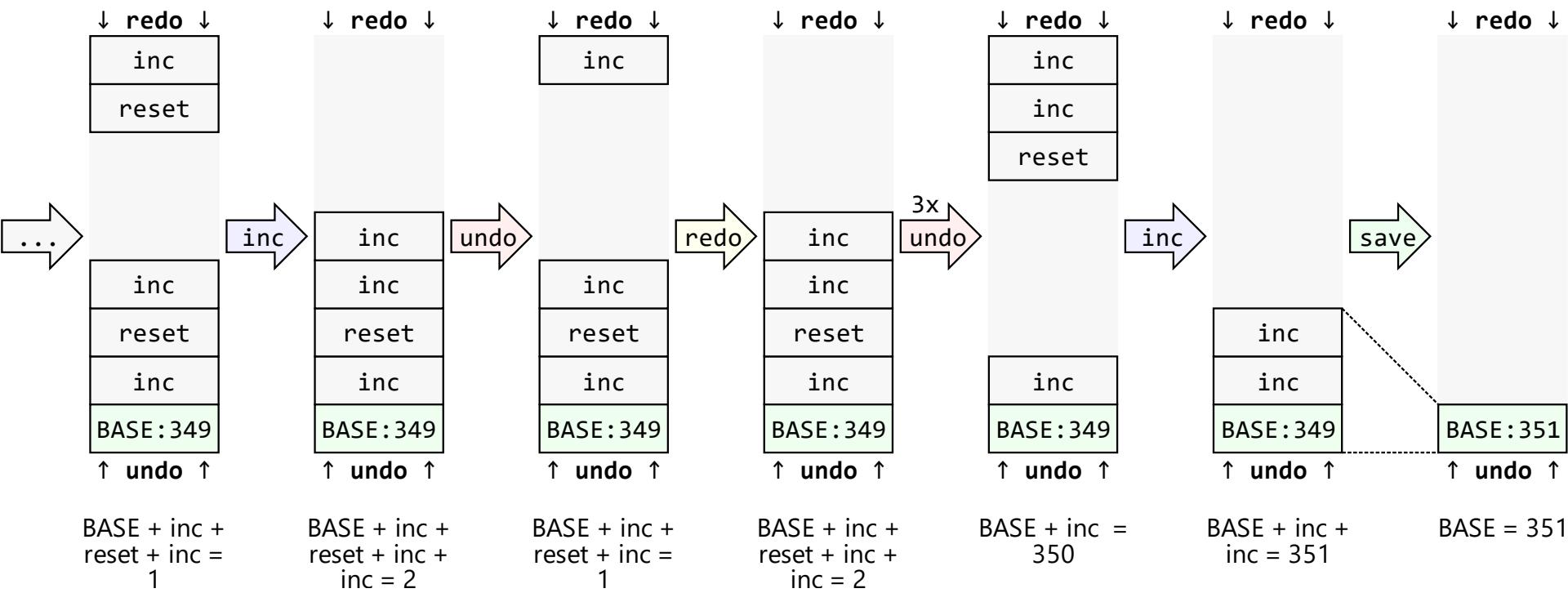
- Forward Undo:
  - save complete baseline document state at some past point:  $S$
  - save change records to transform baseline document into current document state:  $c(b(a(S)))$
  - to undo last action, do not apply last change record:  $S' = \text{undo}(c(b(a(S)))) = b(a(S))$
- Reverse Undo
  - save complete current document state:  $S$
  - save reverse change records to return to previous state:  $\{c^{-1}, b^{-1}, a^{-1}\}$
  - to undo last action, apply last reverse change records:  $S' = \text{undo}(S) = c^{-1}(S)$

Using either of these options requires two stacks

- Undo stack: all change records, saved as you perform actions
- Redo stack: change records that have been undone (so you can reapply)

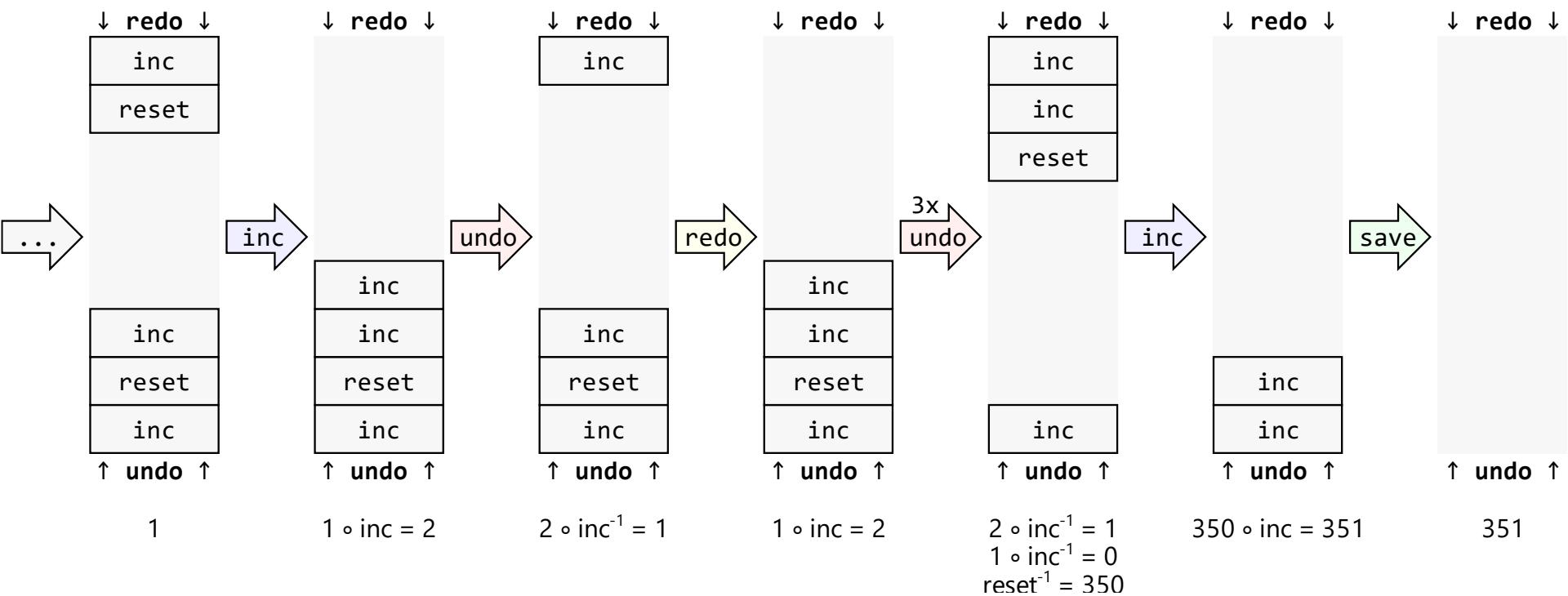
# Undo Stacks

Forward Undo: calculating the current value based on a base value and the “Do” commands on the undo-stack. Undo moves commands from the undo to the redo stack and forces re-calculation; analogous for redo. Save (i.e., creating a memento) consolidates the undo and redo stacks.



# Undo Stacks

Backward Undo: calculating the current value based on the previous value combined with the “Do”- or “Undo”-action of the latest command.



# Implementing Undo

Forward Undo: GoF Command & Memento patterns

- Save one (or more) snapshots
- Apply “Do” commands until desired state has been reached

Backward Undo: GoF Command pattern

- Apply “Undo” commands until desired state has been reached
- Caveat: an action must be “undo-able”.

U  
CS 349

# Implementation

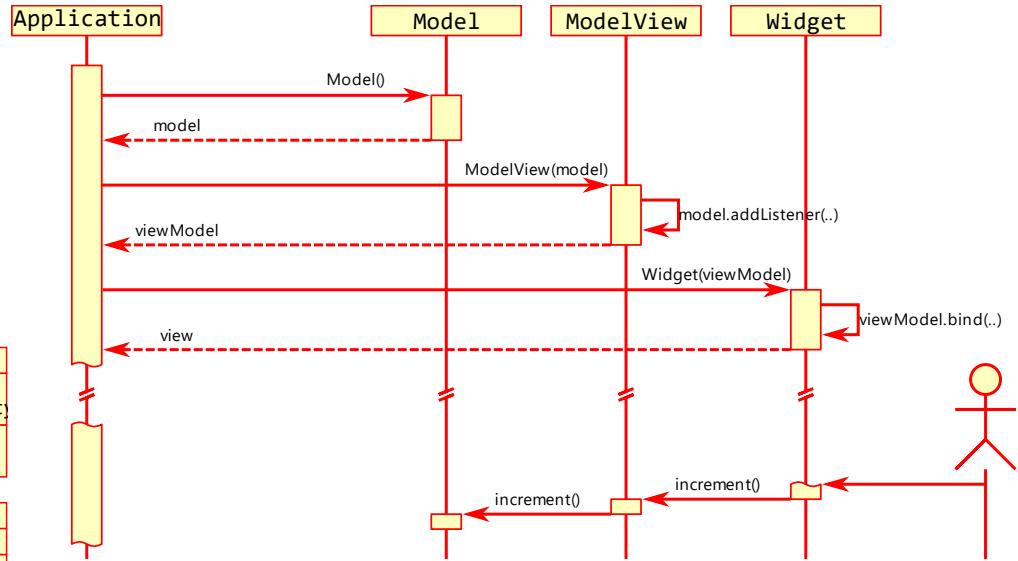
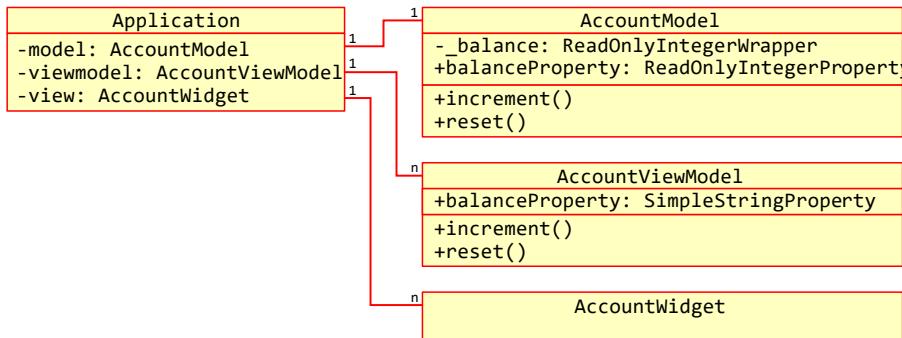
# Implementing Undo – Command Pattern

The command pattern uses an object to encapsulate all information needed to perform an action or trigger an event. This information can include the business logic / function pointer to the business logic, the owner / target / receiver, and additional parameters.

```
class IncrementCommand : UndoableCommand {  
  
    override fun execute(value: Int) : Int {  
        return value + 1  
    }  
  
    override fun undo(value: Int) : Int {  
        return value - 1  
    }  
}
```

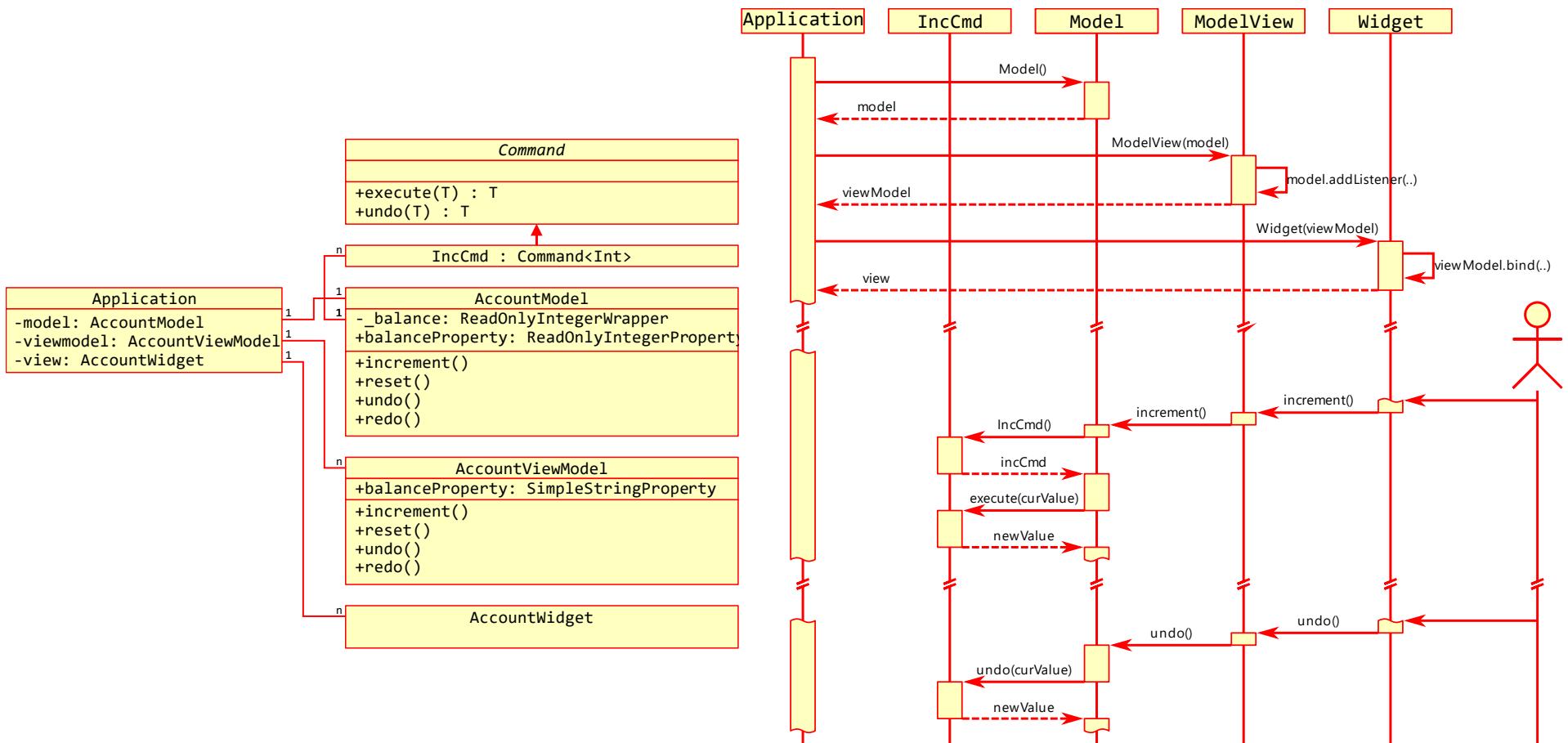
# Undo Implementation

Application with MVVM,  
without undo / redo:



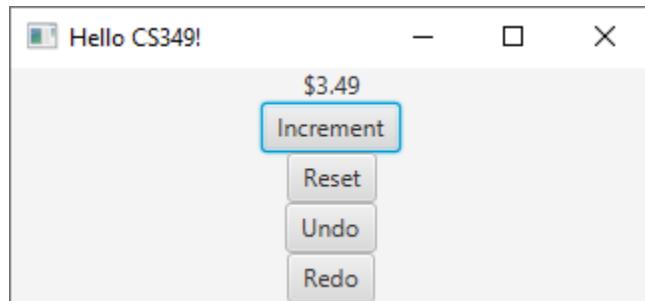
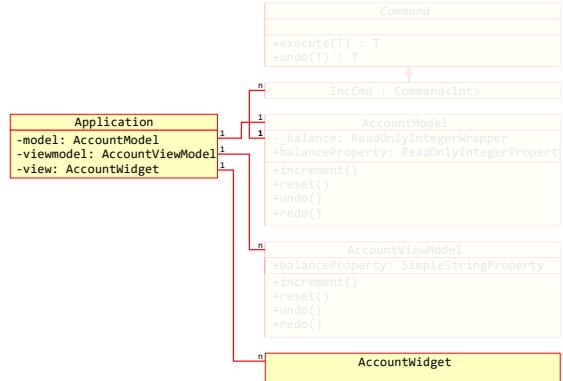
# Undo Implementation

Application with MVVM  
and undo / redo:



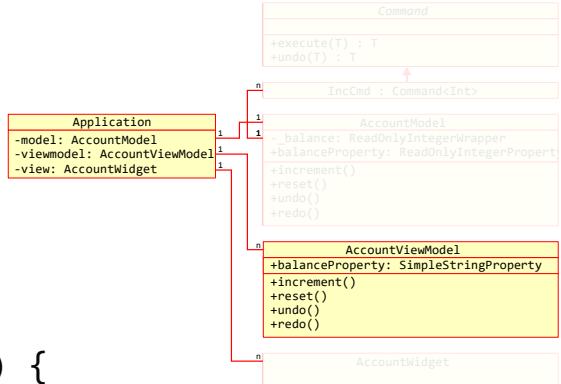
# Undo Implementation – View

```
class BalanceWidget(viewModel : NumberViewModel): VBox() {  
    init {  
        children.addAll(  
            Label(viewModel.numberProperty.value).apply {  
                textProperty().bind(viewModel.numberProperty)  
            },  
            Button("Increment").apply {  
                onAction = EventHandler { viewModel.increment() }  
            },  
            Button("Reset").apply {  
                onAction = EventHandler { viewModel.reset() }  
            },  
            Button("Undo").apply {  
                onAction = EventHandler { viewModel.undo() }  
            },  
            Button("Redo").apply {  
                onAction = EventHandler { viewModel.redo() }  
            })  
        alignment = Pos.CENTER  
    }  
}
```



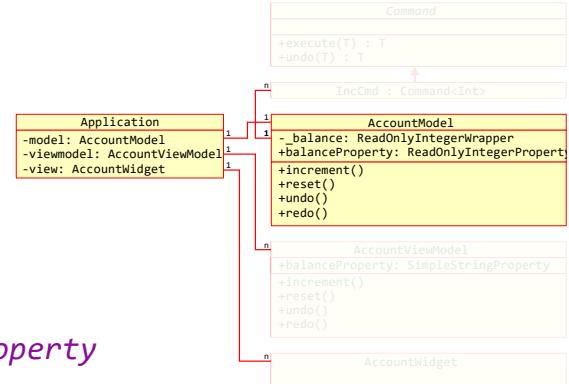
# Undo Implementation – ViewModel

```
class BalanceViewModel(private val numberModel: BalanceModel) {  
  
    val balanceProperty = SimpleStringProperty()  
  
    init {  
        balanceProperty.bind(Bindings.createStringBinding(  
            { "${model.balanceProperty.value / 100}." +  
              "${model.balanceProperty.value % 100 / 10}" +  
              "${model.balanceProperty.value % 10}" },  
            model.balanceProperty)  
    )}  
  
    fun increment() { numberModel.increment() }  
    fun reset() { numberModel.reset() }  
    fun undo() { numberModel.undo() }  
    fun redo() { numberModel.redo() }  
}
```



# Backward Undo Implementation – Model

```
class AccountModel {  
    private val _balance = ReadOnlyIntegerWrapper(0)  
    val balanceProperty: ReadOnlyIntegerProperty = _balance.readOnlyProperty  
    private val undoCommands = mutableListOf<Any>()  
    private val redoCommands = mutableListOf<Any>()  
    fun incrementBalance() {  
        IncrementCommand().apply {  
            undoCommands.add(this)  
            _balance.value = execute(_balance.value)  
        }  
        redoCommands.clear()  
    }  
    fun resetBalance() {  
        ResetCommand().apply {  
            undoCommands.add(this)  
            _balance.value = execute(_balance.value)  
        }  
        redoCommands.clear()  
    }  
    fun undo() {  
        undoCommands.removeLastOrNull()?.apply {  
            redoCommands.add(this)  
            _balance.value = (this as UndoableCommand<Int>).undo(_balance.value)  
        }  
    }  
    fun redo() {  
        undoCommands.removeLastOrNull()?.apply {  
            undoCommands.add(this)  
            _balance.value = (this as UndoableCommand<Int>).execute(_balance.value)  
        }  
    }  
}
```

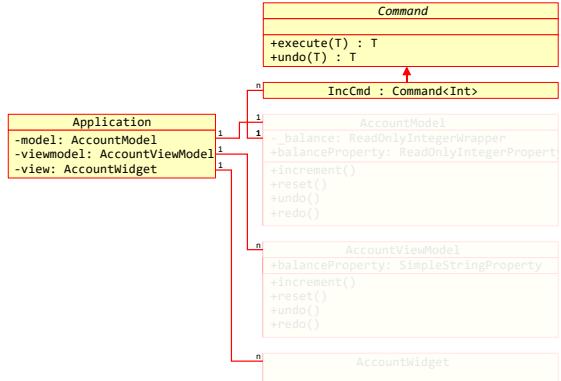


# Backward Undo Implementation – Model

```
interface UndoableCommand<T> {
    fun execute(value: T) : T
    fun undo(value: T) : T
}

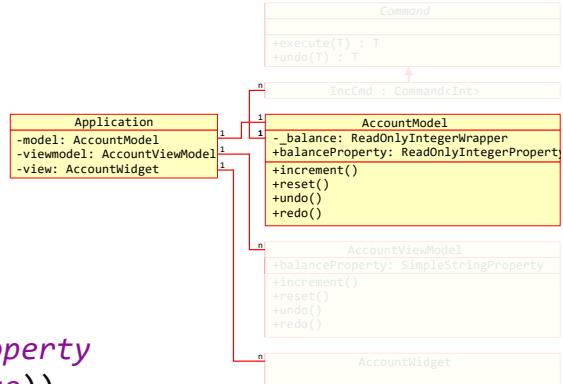
class IncrementCommand : UndoableCommand<Int> {
    override fun execute(value: Int) : Int {
        return value + 1
    }
    override fun undo(value: Int) : Int {
        return value - 1
    }
}

class ResetCommand() : UndoableCommand<Int> {
    private var oldVal = 0
    override fun execute(value: Int) : Int {
        oldVal = value
        return 0
    }
    override fun undo(value: Int) : Int {
        return oldVal
    }
}
```



# Forward Undo Implementation – Model

```
class AccountModel {  
    private val _balance = ReadOnlyIntegerWrapper(0)  
    val balanceProperty: ReadOnlyIntegerProperty = _balance.readOnlyProperty  
    private val undoCommands = mutableListOf<Any>(Memento(_balance.value))  
    private val redoCommands = mutableListOf<Any>()  
    fun incrementBalance() { ... }  
    fun resetBalance() { ... }  
    fun undo() {  
        if (undoCommands.size > 1) {  
            undoCommands.removeLast().apply {  
                redoCommands.add(this)  
                _balance.value = undoCommands.fold((undoCommands[0] as Memento).execute(0)) {  
                    acc, cur -> (cur as Command<Int>).execute(acc)  
                }  
            }  
        }  
    }  
    fun redo() { ... }  
}
```



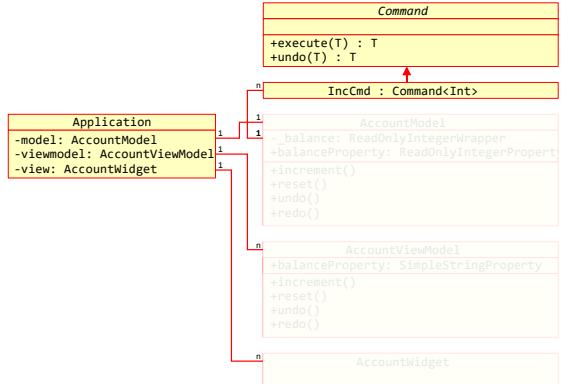
# Forward Undo Implementation – Model

```
interface Command<T> {  
    fun execute(value: T) : T  
}
```

```
class IncrementCommand : Command<Int> {  
    override fun execute(value: Int) : Int {  
        return value + 1  
    }  
}
```

```
class ResetCommand() : Command<Int> {  
    override fun execute(value: Int) : Int {  
        oldVal = value  
        return 0  
    }  
}
```

```
class Memento(private val savedState: Int) : Command<Int> {  
    override fun execute(value: Int): Int {  
        return savedState  
    }  
}
```



# Example: Text Editor Undo/Redo Commands

Available Commands:

- `insert(string, start)`
- `delete(start, end)`
- `style(start, end, FontWeight.NORMAL | FontWeight.BOLD)`

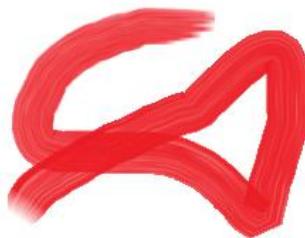
Change	Command	Result
<start>	<code>insert("Quick brown", 0)</code>	Quick brown
<command>	<code>style(6, 10, FontWeight.BOLD)</code>	Quick <b>brown</b>
<command>	<code>insert (" fox", 11)</code>	Quick <b>brown</b> fox
<undo>	<code>delete(11, 14)</code>	Quick <b>brown</b>
<undo>	<code>style(6, 10, FontWeight.NORMAL)</code>	Quick brown
<redo>	<code>style(6, 10, FontWeight.BOLD)</code>	Quick <b>brown</b>
<command>	<code>insert (" dog", 11)</code>	Quick <b>brown</b> dog

# Reverse Command Undo Problems

Consider a bitmap paint application

- `stroke(points, thickness, colour)`
- `erase(points, thickness)`

<start>

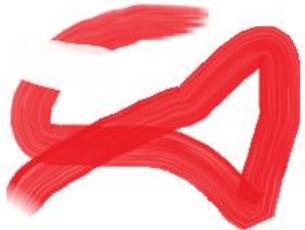


<command>



`stroke(points, 10, black)`

<undo>



`erase(points, 10, black)`

# **Solutions for “Destructive” Commands**

Use forward command undo.

Use backward command undo for “non-destructive” commands, and mementos for destructive commands.

# End of the Chapter



Any further questions?

-



X

# Input Performance

KLM

Fitts' Law

U

CS 349

April 3

# Input Performance Models

When designing a user interface, designers might have to choose between different designs. Implementing all of them, thought, might require too many resources. There must be a way to estimate the performance of a user interface without testing it.

YYYY/MM/DD

Submit

YYYY MM DD

Submit

2021 Apr 4

Submit

2022  
2021  
2020  
2019  
2018  
2017

2021-04-04

< April > < 2021 >

Sun.	Mon.	Tue.	Wed.	Thu.	Fri.	Sat.
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	1
2	3	4	5	6	7	8

U

CS 349

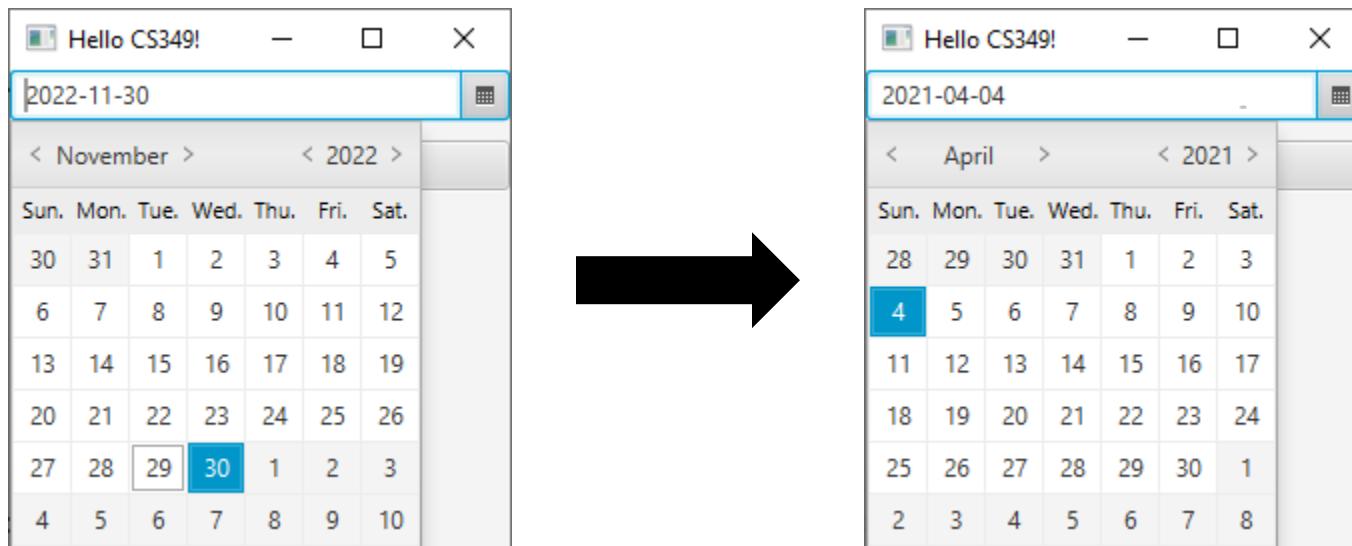
KLM



# Input Performance Models

There are models that abstract how people would use input devices and a user interface, which enables designers to predict time, error, fatigue, learning, etc.

Models most often focus on time and error, as they are easiest to measure.



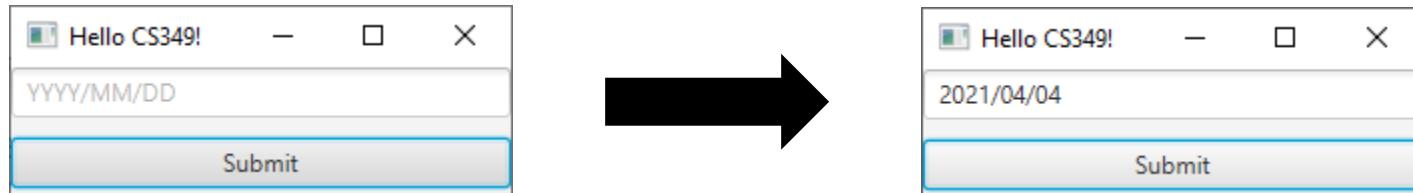
# Keystroke Level Model (KLM)

When using KLM, describe each task with a sequence of operators:

- **K**: Keystroke: typing a single keyboard key
- **P**: Pointing: moving the mouse cursor from one location to another
- **B**: Button: pressing or releasing a mouse button
- **H**: Home: move hand between mouse and keyboard
- **M**: Mental Preparation: planning the next routine action, e.g., finding an icon on the screen.

Sum up times for each operator to estimate how long the task takes.

# KLM Example

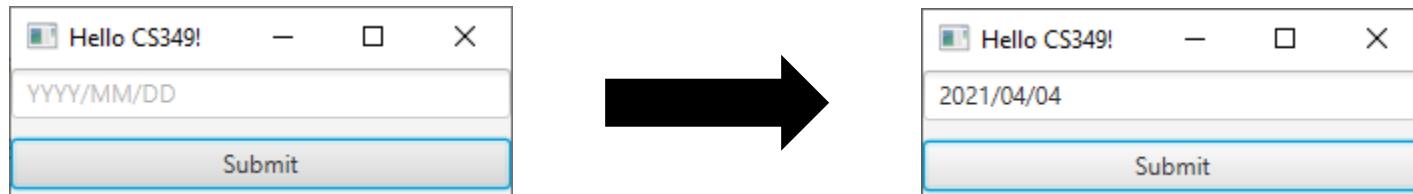


- Assumption: one hand on keyboard, one on the mouse
  - 1. Moving mouse to the TextField **P**
  - 2. Clicking mouse button to set focus on TextField **BB**
  - 3. Switching to Keyboard **H**
  - 4. Pressing 2 0 2 1 / 0 4 / 0 4 **KKKKKKKKKKKK**
  - 5. Switching to Mouse **H**
  - 6. Moving mouse to the Button **P**
  - 7. Clicking mouse button to activate Button **BB**
- 5. – 7. could have been replaced with
  - 5. Press <TAB> to move focus to Button **K**
  - 6. Press <SPACE> to activate Button **K**

# KLM Operators

Code	Operation	Time [s]
K	Key typed	Novice typist
		Average typist
		Expert typist
		Random key
		Key combination
P	Point cursor at target	1.10
B	Button pressed / released	0.10
H	Move hand	0.40
M	Mental preparation	1.20+

# KLM Example



Assumption: one hand on keyboard, one on the mouse

- |    |                   |               |
|----|-------------------|---------------|
| 1. | <b>P</b>          | 1.1 s         |
| 2. | <b>BB</b>         | 0.2 s         |
| 3. | <b>H</b>          | 0.4 s         |
| 4. | <b>KKKKKKKKKK</b> | 2.8 s         |
| 5. | <b>H</b>          | 0.4 s         |
| 6. | <b>P</b>          | 1.1 s         |
| 7. | <b>BB</b>         | 0.2 s = 6.2 s |

5. – 7. could have been replaced with

- |    |          |                 |
|----|----------|-----------------|
| 5. | <b>K</b> | 0.28 s          |
| 6. | <b>K</b> | 0.28 s = 5.06 s |

# KLM Examples

Use KLM to compare performance time of date entry widgets.

Assumption: one hand on keyboard, one on the mouse

A screenshot of a Windows-style application window titled "Hello CS349!". It contains a single input field labeled "YYYY/MM/DD" and a "Submit" button at the bottom. A blue dot is positioned to the left of the window.

P BB H KKKKKKKKKKK K K = 5.06 s

2021/04/04 <TAB> <SPACE>

Op	Time [s]
K	0.28
P	1.10
B	0.10
H	0.40
M	1.20+

A screenshot of a Windows-style application window titled "Hello CS349!". It contains three separate input fields labeled "YYYY", "MM", and "DD", followed by a "Submit" button at the bottom. A blue dot is positioned to the left of the window.

Assumption: cursor jumps to next field

P BB H KKKK KK KK K = 4.22 s

2021 04 04 <SPACE>

A screenshot of a Windows-style application window titled "Hello CS349!". It features a dropdown menu for the year (showing 2021), a dropdown for the month (showing Apr), a dropdown for the day (showing 4), and a "Submit" button at the bottom. A blue dot is positioned to the left of the window.

Assumption: auto-select

P BB H KKKK K K K K K = 4.5 s

2021 <TAB> A <TAB> 4 <TAB> <SPACE>

# Including Mental Operators (M)

Most actions when interacting with a UI do not require conscientious cognitive processing. Sometimes, however, users need to contemplate or strategize their actions beforehand. This can be modelled by one (or multiple) **M** operations.

Examples include:

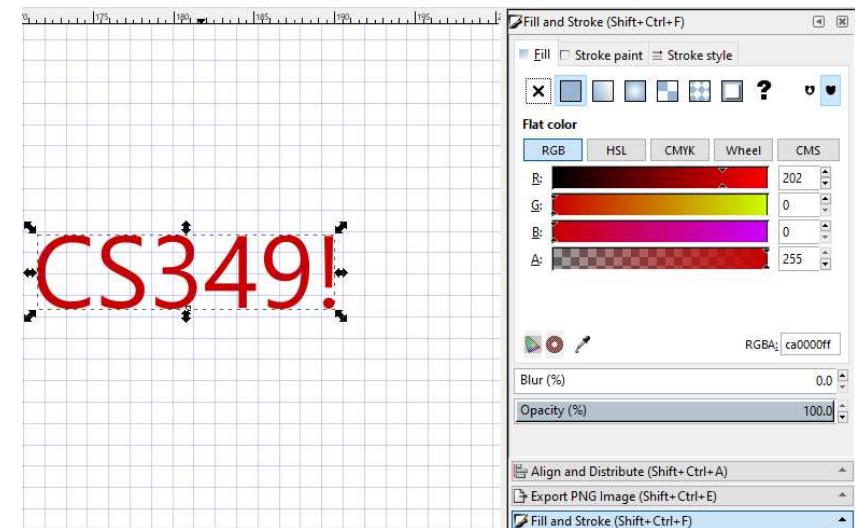
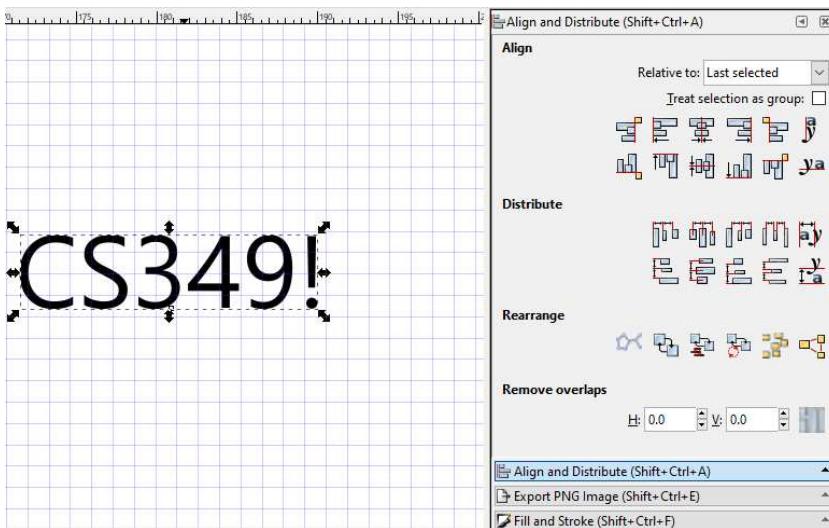
- initiating a new task: e.g., typing a text, then changing the text font.
- retrieving information from semantic (long-term) memory
- find something on the display: e.g., finding a currently visible icon
- think of a task parameter: e.g., setting font-size
- verify that a specification / action is correct (i.e., evaluate feedback)

# Including Mental Operators (M)

Task: Make the text red.

Assumption: text highlighted, mouse over text

- Find “Fill and Stroke”-menu: M 1.2 s
- Move mouse cursor to menu and click: P BB 1.3 s
- Move mouse cursor to “R”-bar: P 1.1 s
- Contemplate value of red (being undecided): MM 2.4 s
- Move mouse to the right value and click: P BB 1.3 s = 7.3 s



# KLM Critique

## Advantages:

- Easy to model
- Does not require mockup, prototype, or implementation

## Disadvantages:

- Some time estimates are generalizing too much
- Some time estimates are inherently variable

# KLM Critique

KLM does not model pointing well and instead uses constant 1.1 s for pointing:

- some pointing devices are faster than others
- intuitively, it should take longer to move the mouse a long distance, or point at a small target



U

CS 349

# Fitts' Law



# Fitts' Law

Fitts' Law is a predictive model for pointing time considering pointing device, travel distance, and target size.

- based on rapid, aimed movements
- works for many kinds of pointing “devices”: finger, pen, mouse, joystick, foot, ..

## Paul Fitts

- Psychologist at Ohio State University
- Early advocate of user-centred design  
(in terms of matching system to human capabilities)



## Distance vs. Size

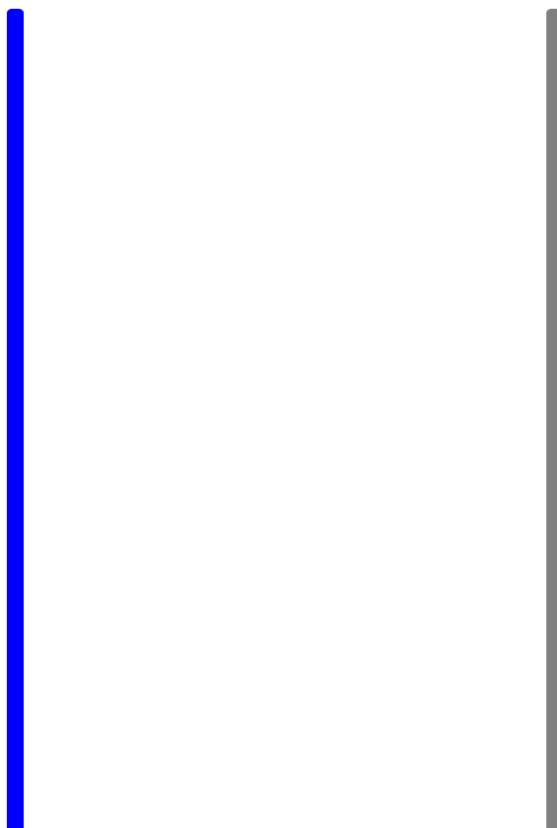
The travel distance  $D$  proportional to the movement time  $MT$ , and the target size  $S$  is negatively proportional the **time**  $MT$ :

$$MT \propto \frac{D}{S}$$

# Fitts' Law Tests

When blue rectangle appears, click on it as fast as possible

<http://ergo.human.cornell.edu/FittsLaw/FittsLaw.html>



# Fitts' Law Tests

When blue rectangle appears, click on it as fast as possible

<http://www.simonwallner.at/ext/fitts/>

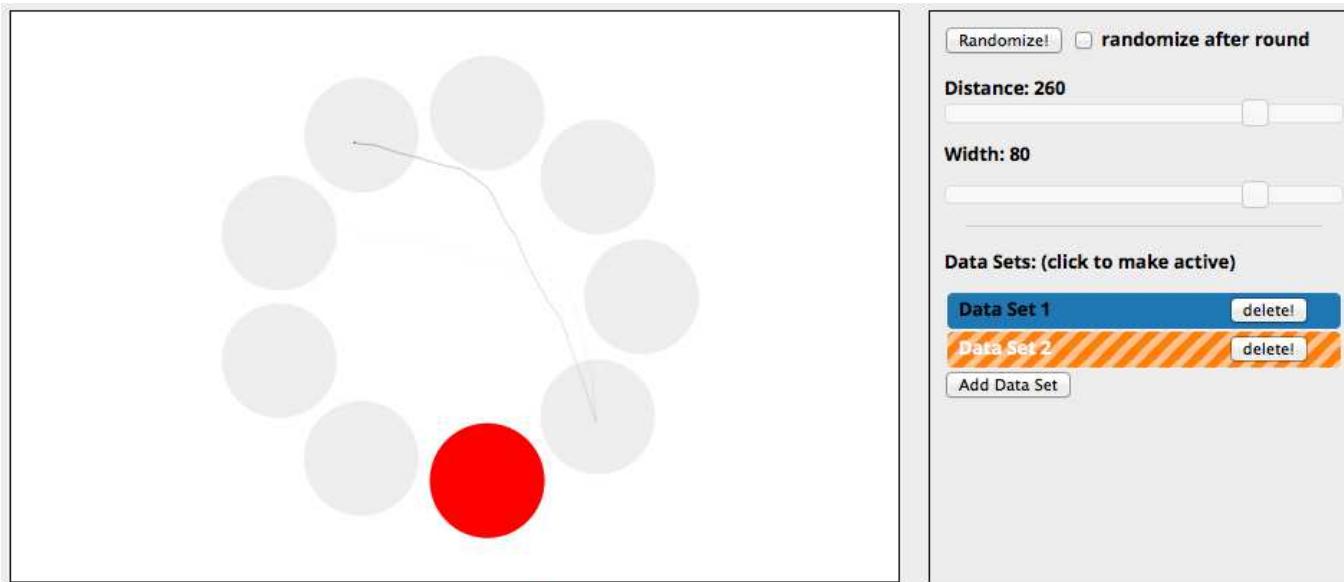


fig. 1a: Test Area: Try to click the red circle as fast as possible but at the same time try to avoid errors.

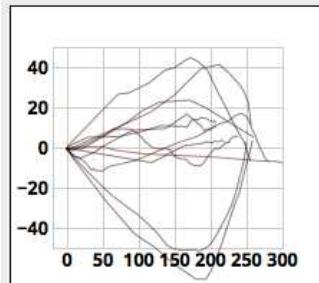


fig. 1b: Deviation form straight path over path distance in px.

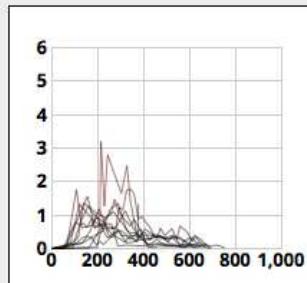


fig. 1c: Movement speed in px/ms over time in ms.

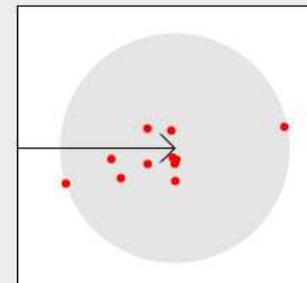


fig. 1d: Click position relative to approach direction.

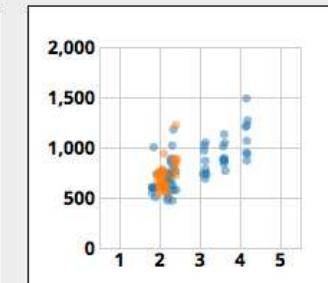


fig. 1e: Time in ms over ID.

## 2D Targets: $W$ as Cross Section of the Pointer Trajectory

$W$  can be interpreted as the minimum error a user can make along the moving direction (“overshooting”) and perpendicular to it (“off-target”).

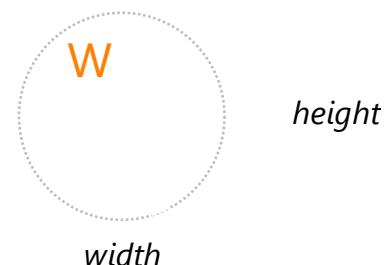
$W$  can be represented as largest circle that can be inscribed in the target.



## 2D Targets: $W$ as Minimum of Target width and height

$$MT = a + b \log_2 \left( \frac{D}{W} + 1 \right)$$

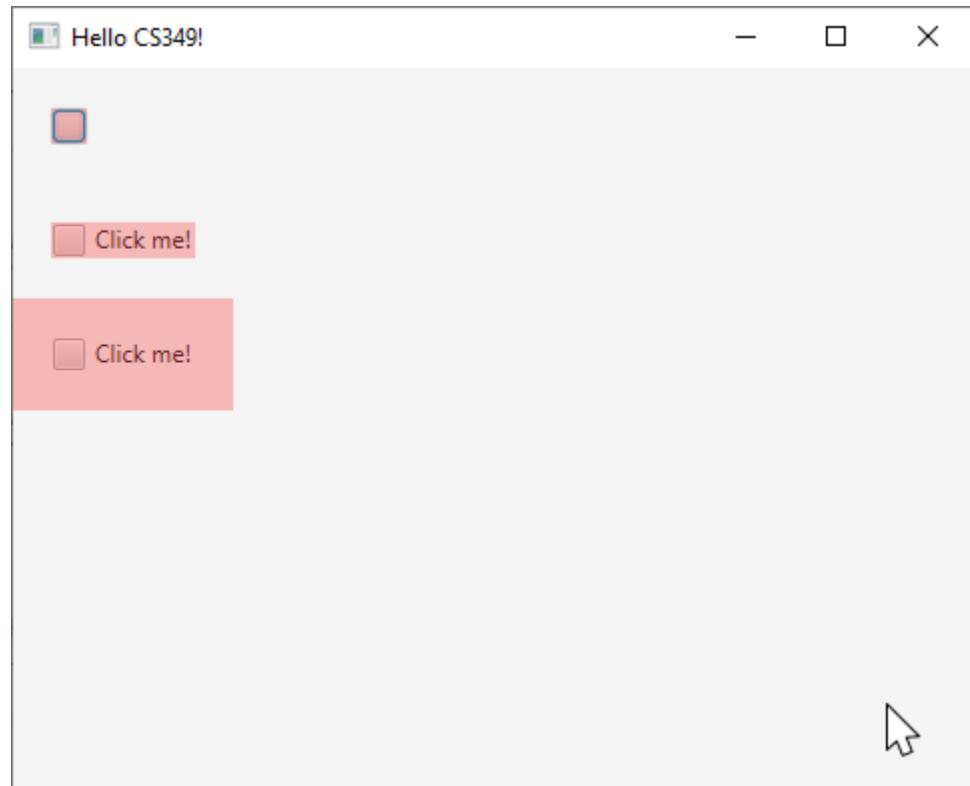
For simplification, we usually interpret  $W$  as the minimum of the targets' *width* and *height*:  $W = \min(\text{width}, \text{height})$ .



D

# Example

1. Checkbox vs
2. Labelled checkbox vs
3. Labeled checkbox with margins

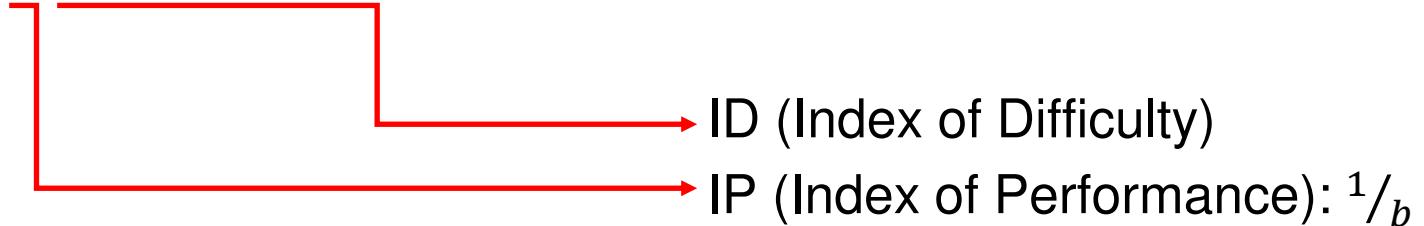


Assuming:  $a = 150$ ,  $b = 450$ :

1.  $MT = a + b \log_2 \left( \frac{500}{18} + 1 \right) = a + 4.85b \Rightarrow MT = 2331 \text{ ms} = 2.33 \text{ s}$
2.  $MT = a + b \log_2 \left( \frac{445}{18} + 1 \right) = a + 4.49b \Rightarrow MT = 2258 \text{ ms} = 2.26 \text{ s}$
3.  $MT = a + b \log_2 \left( \frac{425}{18} + 1 \right) = a + 3.31b \Rightarrow MT = 1640 \text{ ms} = 1.64 \text{ s}$

# Index of Difficulty

$$MT = a + b \log_2 \left( \frac{D}{W} + 1 \right)$$



# Fitts' Law in the Wild

- Large scale study logging cursor movements with real applications

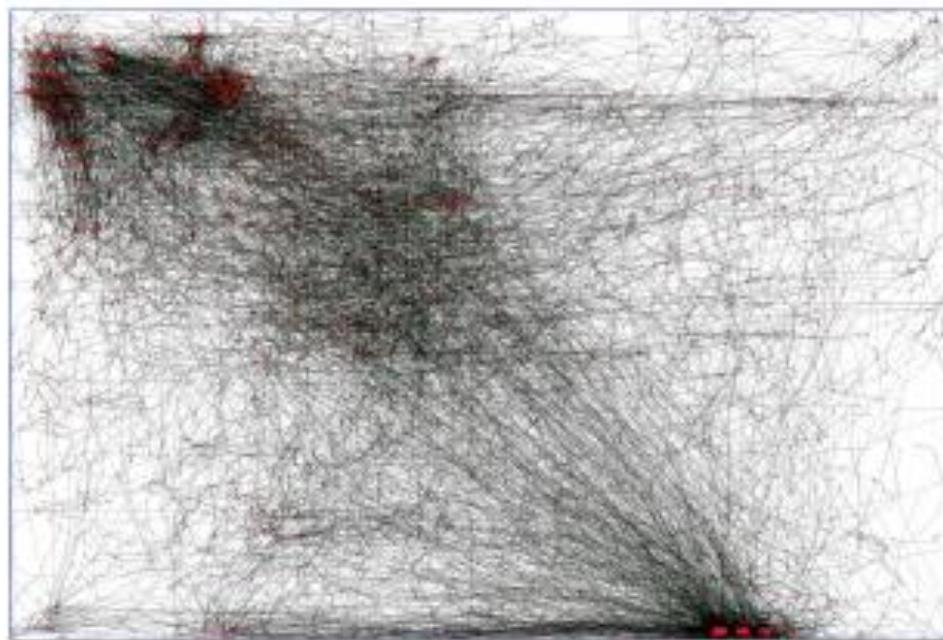


Figure 1. Mouse trajectories (black) and clicks (red).

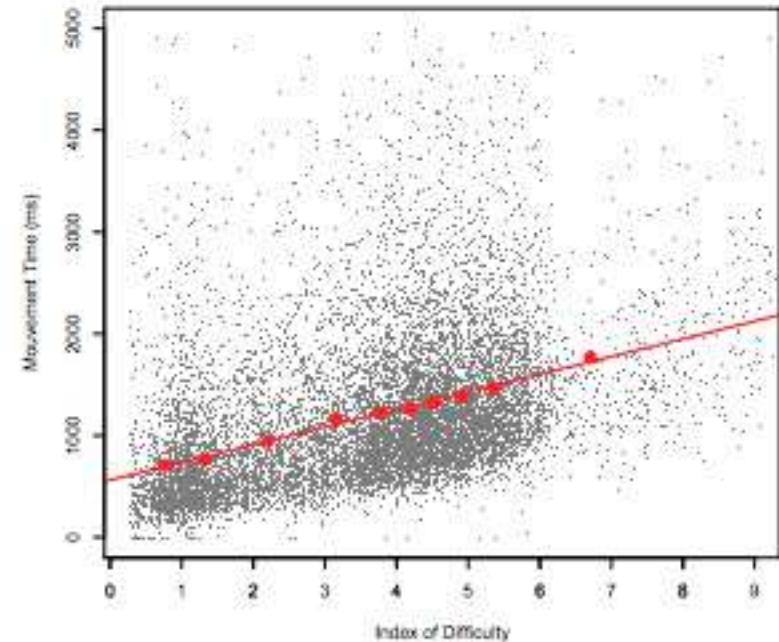


Figure 7. Fitts' linear fit of  $MT$  by  $ID$  using the averaging process for one user (# 4). Small (black) points represent all the points. Large (red) points are the means of each quantile.

# Menu Target Size in MacOS and Windows

Standard menus are great at displaying large number of entries, but they do not allow for high performance.

Context menus lower  $D$ , but  $W$  is still a problem, i.e., too small.

24

Fits' Law in the Wild

- Large scale study logging cursor movements with real applications

Figure 1: Mouse trajectories (black) and clicks (red).

Figure 1: Fitts' Law of RT vs D using the average process for each target size. The red dots show the mean of the process. Error bars show the mean of the error percentiles.

25

Menu Target Size in Mac OS and Windows

Standard menus are great at displaying large number of entries but they do not allow for high performance.

Context menus lower  $D$  but  $W$  is still a problem, i.e., too small.

26

Context Menus, Pie Menus, Marking Menus

With Pie Menus, the active area has a better balance of  $D$  and  $W$ .

Windows

M

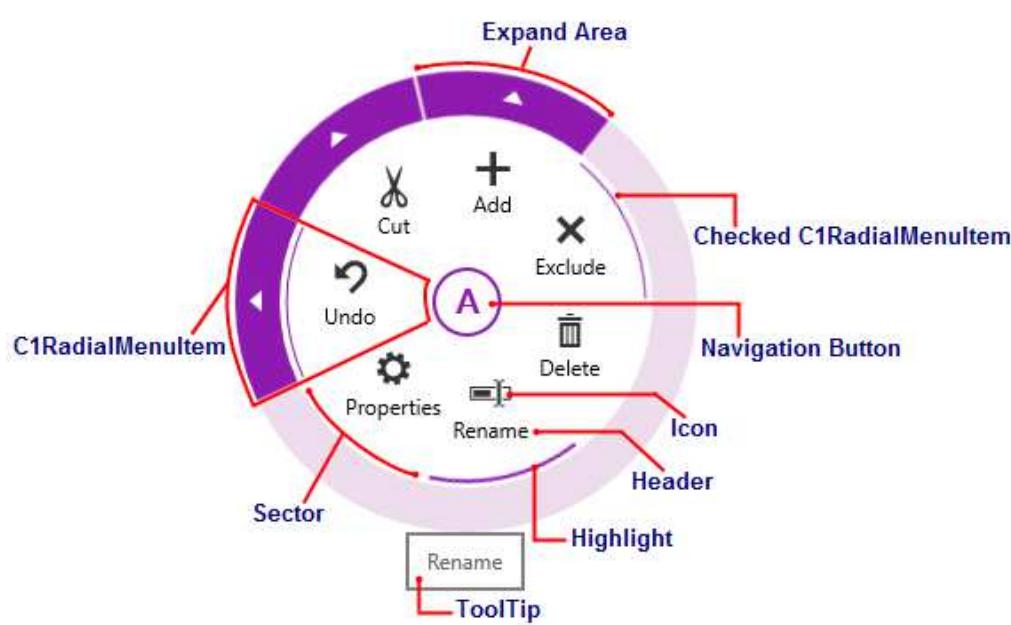
Search the menus

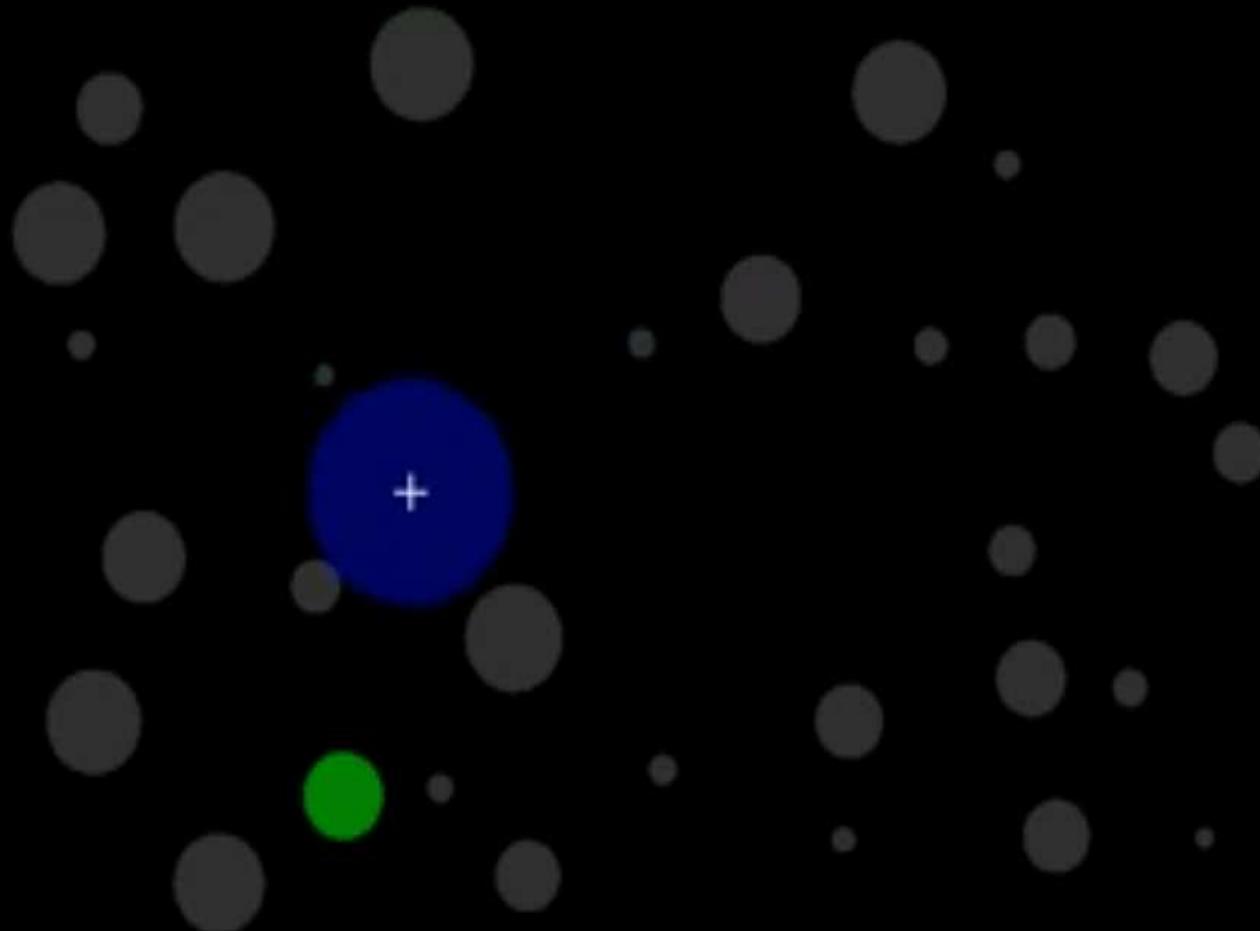
- X Cut
- C Copy
- P Paste Options...
- E Exit Edit Test
- A Font...
- P Paragraph...
- B Bullets
- N Numbering
- Convert to SmartArt
- Lock
- Link
- Search
- Reuse Slides
- Synonyms
- Translate
- Format Text Effects...
- Format Shape...
- New Comment

24

# Context Menus, Pie Menus, Marking Menus

With Pie menus or radial menus, the active area has a better balance of *D* and *W*.





Bubble Cursor (Grossman and Balakrishnan, 2005) [http://youtu.be/JUBXkD\\_8ZeQ](http://youtu.be/JUBXkD_8ZeQ)

# Motor Space vs. Visual Space

Dynamically change CD Gain based on position of cursor:

- Make the cursor move more slowly when over the save button makes it larger in “motor space” even though it looks the same size in “screen space”.

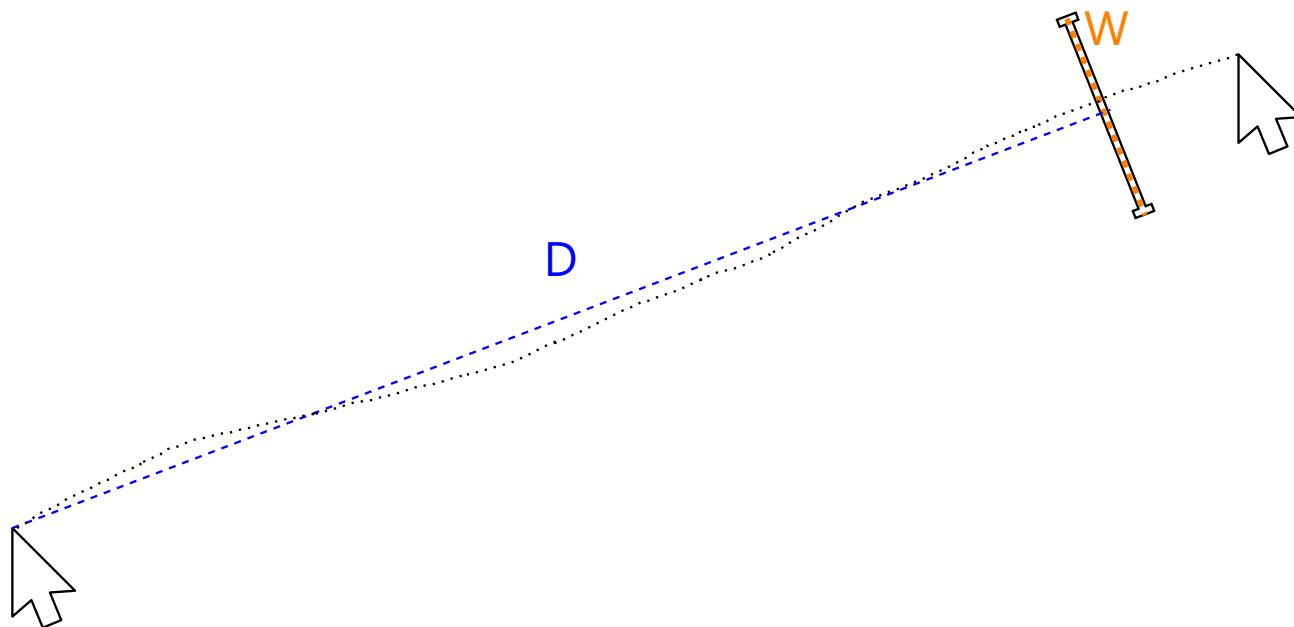


```
var prevCoords = Point2D(0.0, 0.0)
val btnSticky = Button("Sticky").apply {
    addEventFilter(MouseEvent.MOUSE_ENTERED) {
        prevCoords = Point2D(it.screenX, it.screenY)
    }
    addEventHandler(MouseEvent.MOUSE_MOVED) {
        prevCoords = Point2D(prevCoords.x + (it.screenX - prevCoords.x) / 4.0,
                             prevCoords.y + (it.screenY - prevCoords.y) / 4.0)
        Robot().mouseMove(prevCoords.x, prevCoords.y)
    }
}
```

# Crossing Selection

Steering the cursor through a target of width  $W$ .

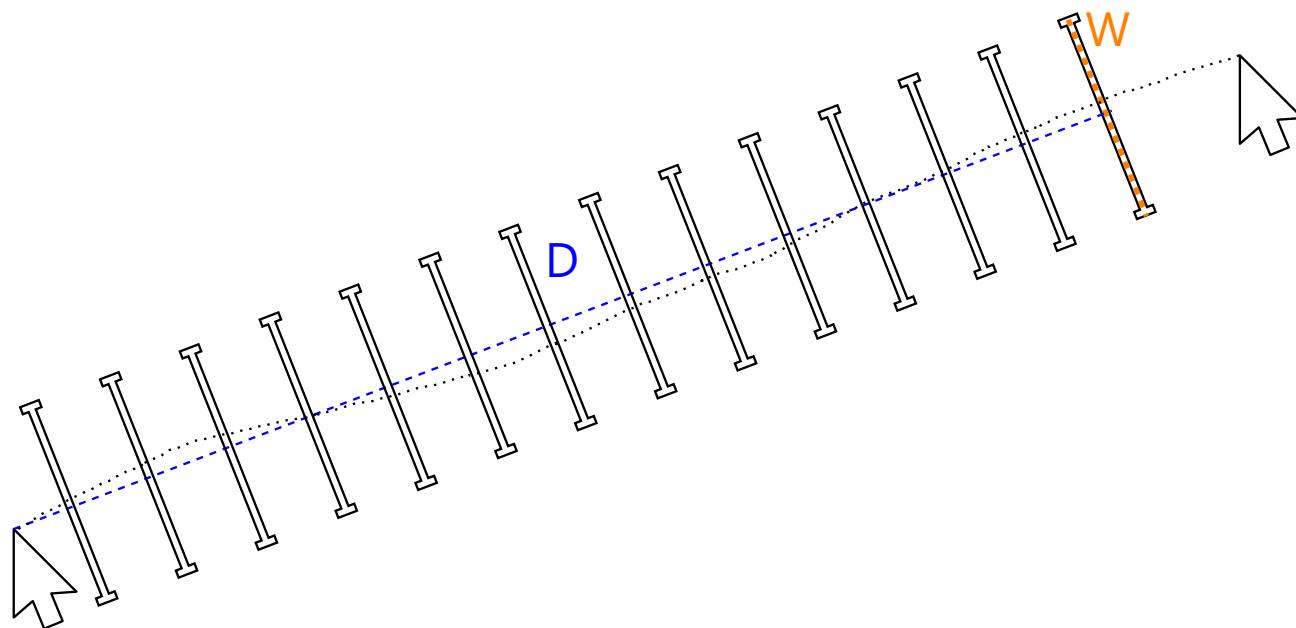
- Fitts' Law is valid for this task as well.



# Steering Law

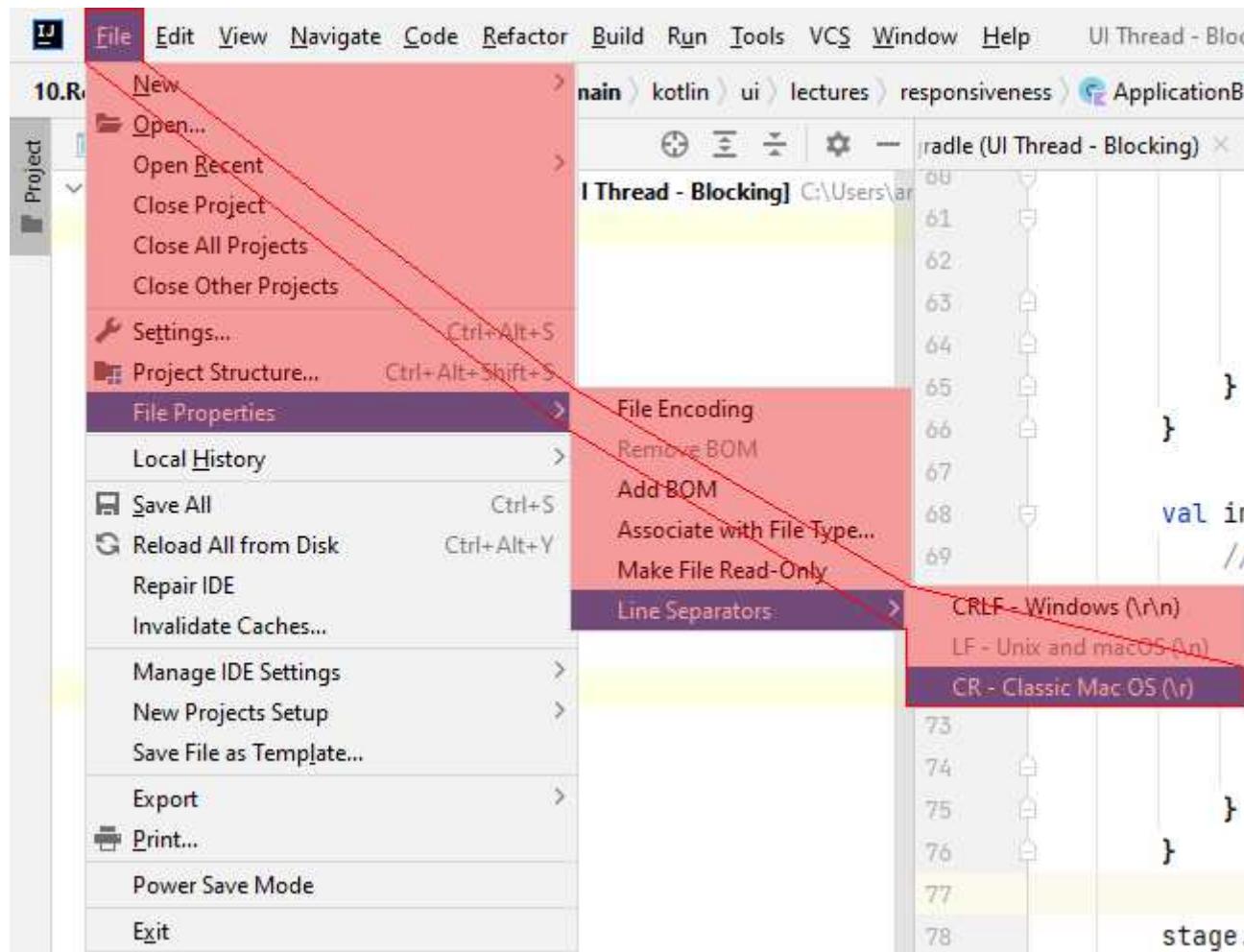
Steering a cursor along a path without exiting the imaginary "tunnel".

- An adaptation of Fitts' Law:  $MT = a + \frac{b}{\ln 2} \times \frac{D}{W} = a + \tilde{b} \frac{D}{W}$



# Steering Law

Moving through a constrained path takes longer



# End of the Chapter



Any further questions?