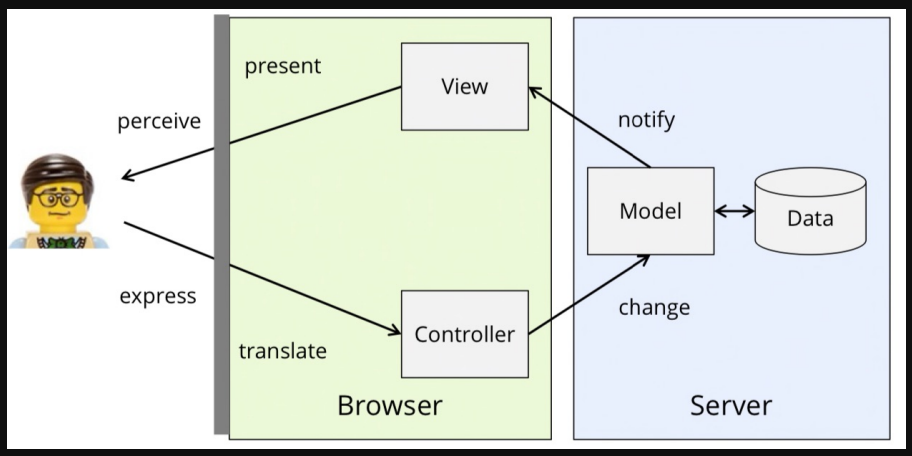
WEB APPS

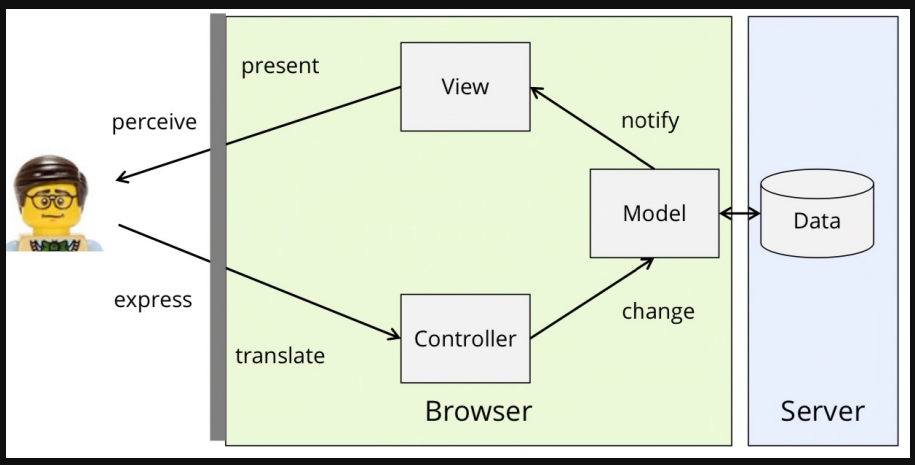
- web apps are software interfaces accessed thru browser
  - ↳ browser acts like OS
    - handles I/O
    - provides UI toolkit (HTML, CSS)
    - provides machine code layer (i.e. JS compiler)
  - ↳ conceptual split btwn UI (client-side) & business logic (server-side)

MVC

- for Model View Controller (MVC) of early web apps
  - ↳ Model on server sends webpage to browser to render View
  - ↳ Controller in browser sends user events to Model
  - ↳ Model processes changes, then sends new webpage to browser

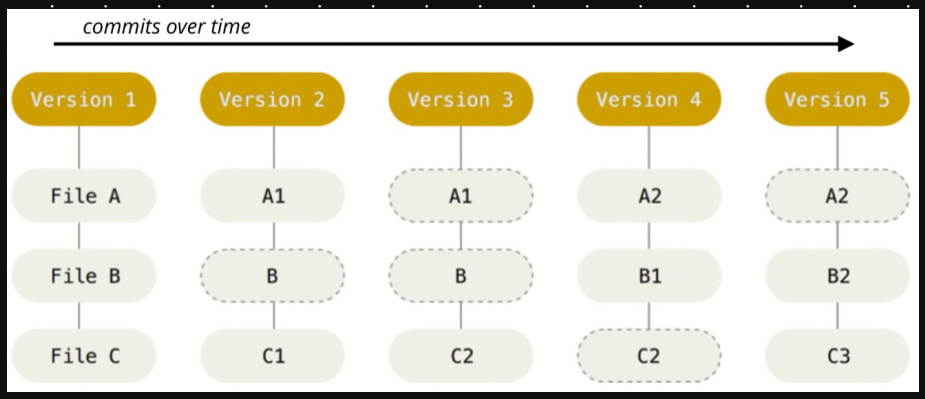


- focus on single page apps (SPAs) w/o server data & processing in this course
  - ↳ browser handles full MVC cycle



GIT

- core conceptual unit is commit
  - ↳ snapshots of tracked files over time



- `.gitignore` is a file that specifies untracked files Git should ignore
  - ↳ for CS349: node, react, macos, linux, windows

VS CODE

- `Prettier` ext is required for CS349
- document where `CoPilot` is used in asgnts

TYPESCRIPT

JAVASCRIPT

- JS is interpreted + just-in-time compiled
- Java syntax w/ fctional roots
- prototype-based OO lang
- has dynamic weakly/loosely types
- has strict + non-strict mode
  - ↳ Vite + TS uses strict
- semicolon line endings are optional
- ternary operator

```
A ? B : C   // equivalent to "if A then B else C"
```

- 3 ways to declare vars
  - ↳ var is not recommended b/c it has non-block scope hoisting
  - ↳ let
  - ↳ const
- primitive types are boolean, number, string, null, + undefined
  - ↳ everything else is obj
    - some other primitives like Symbol
- auto type conversion
- typeof keyword returns type as str
- truthy + falsy have surprising defns
  - ↳ e.g. using ==

```
0 == ""  // true!
1 == "1" // true!
```

  - ↳ generally use === for strict equality comparisons
    - e.g.

```
0 === "" // false
1 === "1" // false
```

- logical OR operator (||) is often used to assign default val if var is undefined
  - ↳ e.g.

```
let v;  // v is undefined
v = v || 456; // v is 456 because a was undefined
```

  - ↳ truthy + falsy behaviour may create bugs
    - e.g.

```
let v = 0;
v = v || 456; // v is 456 since 0 is falsy
```

  - ↳ use nullish coalescing op b/c it's only false when null / undefined

- e.g.

```
let v = 0;
v = v ?? 456; // v is 0 because not undefined or null
```

- 3 ways to create fcns:

Function *declaration*

```
function add1(a, b) { return a + b; }
console.log(add1(1, 2)); // 3
```

Function *expression*

```
const add2 = function (a, b) { return a + b; }
console.log(add2(1, 2)); // 3
```

Function *expression* using "arrow notation" / "lambda notation"

```
const add3 = (a, b) => a + b;
console.log(add3(1, 2)); // 3
```

- fcn can be assigned to var
  - ↳ e.g.

```
function sayHello() { return "Hello, "; }
const saySomething = sayHello;
console.log(saySomething()); // "Hello, "
```

- fcns can be passed to other fcns
  - ↳ common for callback fcns
  - ↳ e.g.

```
function greeting(msg, name) { return msg() + name; }
console.log(greeting(sayHello, "Sam")); // Hello, Sam
```

- fcns can be returned from other fcns
  - ↳ aka factory fcns
  - ↳ e.g.

```
function makeGreeting() {
  return function (name) { return “Hi ” + name; }
}

const greet = makeGreeting();
console.log(greet("Sam")) // Hi Sam
```

an anonymous function

- closures are when an inner fcn references state of outer fcn
  - ↳ e.g.

```
function makeRandomGreeting() {
  const sal = Math.random() > 0.5 ? "Hi " : "Hello ";
  return function (name) { return sal + name; }
}
const greeting = makeRandomGreeting();
console.log(greeting("Sam")) // ?? Sam
```

↳ e.g. outer state includes fcn params

```
function makeGreeting(msg) {
  return function (name) { return msg + name; }
}

const greeting1 = makeGreeting("Hello");
console.log(greeting1("Sam")); // Hello, Sam
```

e.g. factory fcn that captures fcn (i.e. passing fcn to factory fcn)

```
function makeGreeting(msg  ) {
  return function (name) { return msg() + name; }
}
function sayHello() { return "Hello, "; }
const greeting2 = makeGreeting(sayHello);
console.log(greeting2("Sam")); // Hello, Sam
```

↳ in this context, common to use lambda fcns

- e.g.

```
const greeting3 = makeGreeting(() => "Howdy! ");
```

an anonymous lambda function

```
console.log(greeting3("Sam")); // Howdy, Sam
```

string literals delimited by backtick (`) enables string interpolation, multi-line strs, + tagged templates

↳ e.g.

```
const v = 15.7;
const units = "cm";
```

Without string interpolation:

```
let msg = "Length is " + v + " " + units + ".";
```

With string interpolation:

```
let msg =`Length is ${v} ${units}.`
```

formatting method of primitive type

Can use *expressions* in template literal:

```
let msg =`Length is ${(v / 100).toFixed(2)} cm.`
```

JS objects can be defined using JSON-like notation

↳ e.g.

```
const square = {
  colour: "red",
  size: 10,
  draw: function () {
    return `A ${this.size} pixel ${this.colour} square.`;
  }
}
```

Get property

```
console.log(square.colour); // red
```

Set property

```
square.colour = "blue";
```

Call "method" (technically a "function property")

```
console.log(square.draw()); // A 10 pixel blue square.
```

JS has no formal separation of classes + objs

↳ objs are linked to special obj called **prototype**

- all objs have [[Prototype]] property
- prototype contains properties + methods for linked objs
- multiple prototypes form a **chain**

objs created using a ctor fcn + **new** keyword

e.g. prototype chain using ctor fcn

```
// a constructor function
function Shape(colour) {
  this.colour = colour;
  this.draw = function () {
    return `A ${this.colour} shape.`;
  }
}

function Square(colour, size) {
  Shape.call(this, colour);
  this.size = size;
  this.draw = function () {
    return `A ${this.colour} square with size ${this.size}`;
  }
}

const square = new Square("red", 10);
```

"this" refers to object context

call prototype constructor and link to this object

a "shadow" property

class keyword is abstraction for prototypical inheritance mechanism

```
class Shape {
  constructor(colour) { this.colour = colour; }
  draw() { return `A ${this.colour} shape.`; }
}

class Square extends Shape {
  constructor(colour, size) {
    super(colour);
    this.size = size;
  }
  draw() {
    return `A ${this.colour} square size ${this.size}`;
  }
}

const square = new Square("red", 10);
```

call prototype constructor and link to this object

a "shadow" property

arrays are an example of an iterable obj

↳ ways to declare:

```
let arr1 = [] // empty array with length 0
let arr2 = Array(5); // empty array with length 5
let arr3 = [1, 2, 3, 4, 5]; // populated array
let arr4 = Array(5).fill(99); // 5 elements, all 99
```

"empty" is not a typo!

↳ ways to iterate:

```
for (let i = 0; i < arr3.length; i++) {
  console.log(arr3[i])
}

for (const x of arr3) { console.log(x) }

arr3.forEach((x) => console.log(x));
```

e.g. fontional arr methods

```
let arr3 = [1, 2, 3, 4, 5];
```

map returns array with transformed elements:

```
const arr4 = arr3.map((x) => x * 10);
// [10, 20, 30, 40, 50]
```

find returns first element that satisfies condition:

```
const a = arr3.find((x) => x % 2 == 0);
// 2
```

filter returns all elements that satisfy condition

```
const arr5 = arr3.filter((x) => x % 2 == 0);
// [2, 4]
```

reduce executes a function that accumulates a single return value

```
const arr6 = arr3.reduce((acc, x) => acc + x, 0);
// 15
```

destructuring means to unpack arr elmts / obj properties into distinct vars

↳ e.g.

From Arrays

```
let arr3 = [1, 2, 3, 4, 5];
let [a, b] = arr3; // a = 1, b = 2
```

From Objects

```
let obj = { "a": 1, "b": 2, "c": 3 };
let { a, b } = obj; // a = 1, b = 2
```

Can rename destructured variables from objects

```
let obj = { "a": 1, "b": 2, "c": 3 };
let { a: x, b: y } = obj; // x = 1, y = 2
```

means "unpack value for b and store in y"

spread expands iterable obj (e.g. arr, str)
↳ e.g.

```
let arr3 = [1, 2, 3, 4, 5];
let arr4 = [-1, 0, ...arr3, 6, 7];
console.log(arr4); // [-1, 0, 1, 2, 3, 4, 5, 6, 7]
```

rest condenses multiple elmts into single elmt
↳ e.g.

```
let arr3 = [1, 2, 3, 4, 5];
let [a, b, ...c] = arr3; // c = [3, 4, 5]

console.log(c)
const obj = { a: 1, b: 2, c: 3 };
let { a, ...x } = obj; // x = {b: 2, c: 3}
```

e.g. creating prepopulated arr using spread + map

```
let arr5 = [...Array(5)].map((_, i) => i + 1);
```

Spread length 5 empty array

Ignore element value

array index

```
console.log(arr5); // [0, 1, 2, 3, 4]
```

risks of dynamic weak type checking:
↳ surprising behaviour
- re-asgnt to diff w/no error
- ops assume types work fine or silently fail

↳ typo in fcn name won't be caught until runtime
↳ no checks on invalid (i.e. # or type) fcn arguments
↳ typo in property name creates new property
↳ fcn doesn't always return val + returns undefined in some cases

TYPESCRIPT

TS is superset of JS
↳ has type checking
adds static types
↳ enables type checking + code completion
TS is transpiled to JS
↳ browser only executes JS (i.e. JS during runtime)
Vite dev env alr transpiles to TS so no need to install tsc compiler
TS types are annotations so they're checked at compile time, not runtime
↳ e.g.

```
let a = 123;
a = a + "hello";
```

```
Type 'string' is not assignable to
type 'number'.ts(2322)
```

```ts
const obj = { a: 1, b: 2 };
obj.aa = 123;
```

```
Property 'aa' does not exist on type
'{ a: number; b: number; }'.ts(2339)
```

e.g. explicit type annotation

```ts
let n: number = 123;
let b: boolean = true;
let s: string = "Hello";
```

e.g. implicit type inference

```ts
let n = 123;
let b = true;
let s = "Hello";
```

↳ can result in any type, which we want to avoid

- e.g.

```ts
let x;
x = 123;
x = x + "ABC";
```

e.g. arr, obj, & fcn types

Array

– two equivalent type declarations

```ts
let arr1: number[];
let arr2: Array<number>; // generics!
```

Object

```ts
let obj: { a: number; b: string } = {
  a: 1,
  b: "hello",
};
```

Function

```ts
function add(a: number, b: number): number {
  return a + b;
}
```

TypeScript has a **void** type for functions that don't return anything

type alias is when we define type annotation to reuse

↳ e.g.

Object type alias example:

```ts
type Point = { x: number; y: number };
```

Function type alias example:

```ts
type SimpleCallback = (data: string) => void;
```

the convention is CamelCase for type aliases

interfaces are an alt way to define obj type

↳ e.g.

```ts
interface Point { x: number; y: number; }
```

↳ unlike type alias, interface can be extended

- e.g.

```ts
interface Shape { colour: string; }

interface Square extends Shape { size: number; }

let square = { colour: "red", size: 10 } as Square;
```

a "type assertion"

structural type checking (aka duck typing): only structure of type matters

↳ e.g.

```ts
function renderSquare(square: Square) {
  console.log(
    `A ${square.colour} square of size  ${square.size}`
  );
}

let square = { colour: "red", size: 10 };
renderSquare(square); // still works!

let square2 = { colour: "blue", size: 20, alpha: 0.5 };
renderSquare(square2); // also works!
```

no type assertion!

an extra property that isn't in Square!

union types combine diff types

↳ e.g.

```ts
function printId(id: number | string) {
  console.log("Your ID is: " + id);
}

printId(101); // ok

printId("202"); // ok

printId({ myID: 22342 }); // error!
```

type narrowing: reducing type of var from broader to more specific

↳ e.g. guarding

```ts
const el = document.getElementById("apple");
if (el)
  el.setAttribute("class", "red");
```

truthiness guard

↳ usually needed w/ union types

- e.g.

```ts
function formatId(id: number | string): string {
  if (typeof id === "number") {
    return `${id}`;
  } else {
    return id.toUpperCase();
  }
}
```

can specify fcnal/obj params as optional

↳ usually type narrowing required

↳ e.g

with default value

```
function add1(a: number, b: number, c: number = 0) {
  return a + b + c;
}
```

with ?

```
function add2(a: number, b: number, c?: number) {
  if (c) {
    return a + b + c;
  } else {
    return a + b;
  }
}
```

e.g. class w/ types

```
class Shape {
  colour: string;
  constructor(colour: string) { this.colour = colour; }
  draw() { return `A ${this.colour} shape.`; }
}

class Square extends Shape {
  size: number;
  constructor(colour: string, size: number) {
    super(colour);
    this.size = size;
  }
  draw() {
    return `A ${this.colour} square size ${this.size}`;
  }
}
```

methods + classes can be abstract

properties can be private, public, or protected

↳ compile-time annotation

↳ e.g.

```
class Shape {                                   // means colour is a property
  constructor(public colour: string) {}
  draw() {
    return `A ${this.colour} shape.`;
  }
}
const shape = new Shape("red");
console.log(shape.draw()); // A red shape.
shape.colour = "blue";                          // mutate colour property
console.log(shape.draw()); // A blue shape.
```

hidden property accessed w/ getter + setter

↳ e.g.

```
class Shape {
  private _colour: string;                      // convention is to use an underscore prefix for "hidden" properties accessed by get/set
  get colour() { ← mutator
    return this._colour;
  }
  set colour(c: string) { ← read-only
    this._colour = c;
  }

constructor(colour: string) {
  this._colour = colour;
}
...
```

- TS has `enum` type
  - ↳ e.g.:

```
enum State {
  Idle,
  Down,
  Up,
}

let state1: State = State.Idle;
```

    - e.g. same result using `literal str types`

```
let state2: "idle" | "down" | "up" = "idle";
```

- avoid circumventing TS type safety
  - ↳ use any type implicitly / explicitly
    - e.g.

```
let a: any = 123;
a = a + "hello";
```

  - ↳ tell TS compiler to ignore errors
    - e.g.

```
let a = 123;
// @ts-ignore
a = a + "hello";
```

  - ↳ use non-null assertion operator (!)
    - e.g.

```
function f(s: string | null) {
  s!.toUpperCase(); // DANGER!
}
```

- `ES modules` are used to organize code by splitting program into multiple files
  - ↳ all vars, fcns, objs, & types are local unless exported
  - ↳ exports from multiple files can be consolidated in index file
  - ↳ to force file to be module if no imports / exports:

```
export {}; // force file to be a module
```

  - ↳ to load module into HTML doc:

```
<script type="module" src=" ... "></script>
```

  - ↳ e.g.

mymodule.ts

```
export function getSecret() {
  return s;
}

let s = "secret";
```

(function is exported; not exported, will be private to module)

main.ts

```
import { getSecret } from "./mymodule";
console.log(getSecret());
```

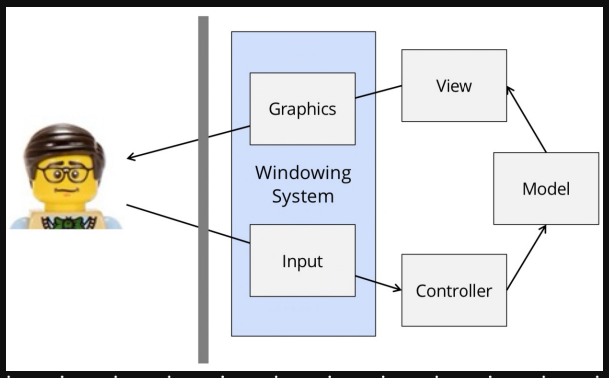
(relative path)

    - console: secret

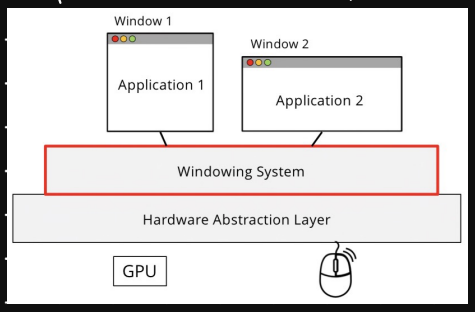
DRAWING

DRAWING MODELS

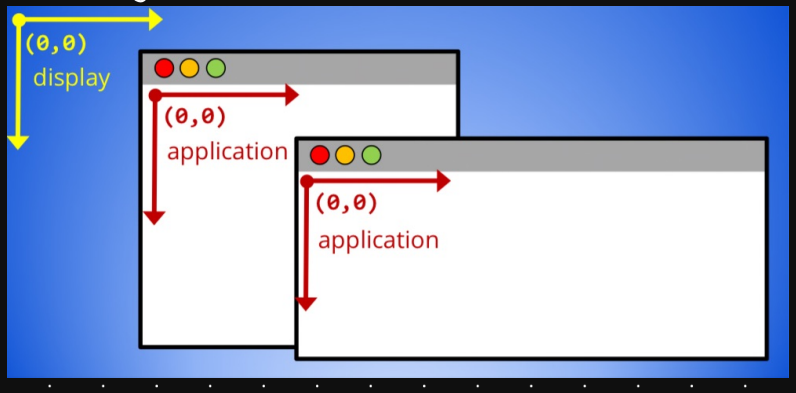
- graphical presentation architecture:



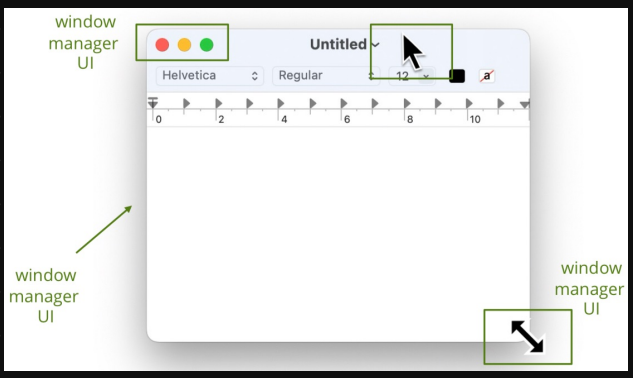
- apps running in windows can be isolated
  - ↳ each has its own mem, resources, + drawing canvas
- windowing system is OS layer for sharing screen space + user input among apps
  - ↳ manages list of windows
  - ↳ provides each app w/ indep drawing area
  - ↳ dispatches low-lvl input events to focused window
  - ↳



- windowing system uses drawing canvas abstraction
  - ↳ each app has defined drawing area within window w/ local coord system ([0,0] is top left)
    - implemented as graphics buffer
  - ↳ use very fast method called bitblt (bit block transfer) to render buffer
  - ↳



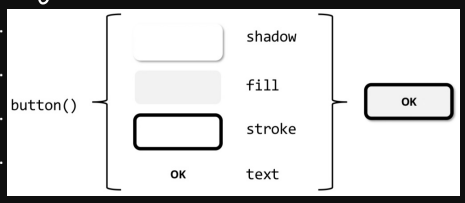
window manager provides a window UI



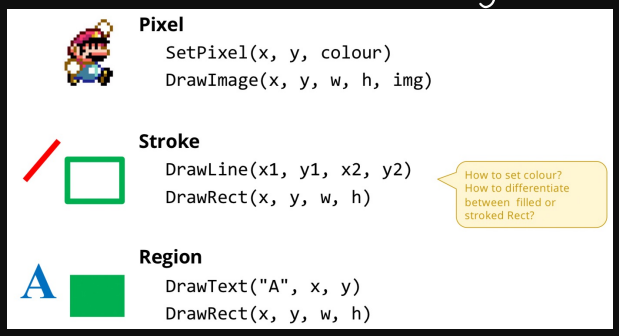
modern web browser is like windowing system

graphical user interface (GUI) is drawing of shapes
↳ user interaction is how shapes change.

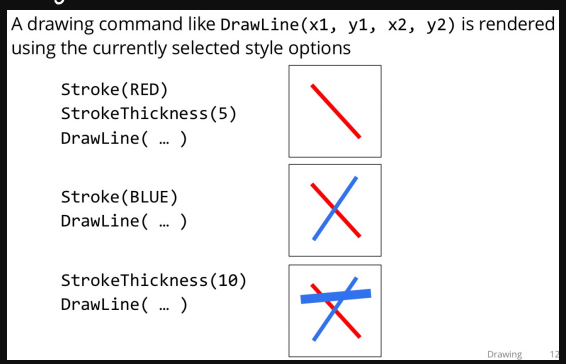
UI toolkit provides lvl of abstraction for frontend programmers
↳ e.g. translates our concept of "button" into rendering of button



3 conceptual models for drawing:



common approach to manage state of drawing style options is to use graphics context, which is set of params + state info that drawing system needs to perform cmds
↳ e.g.:



HTML canvas (HTMLCanvasElement) is literal form of canvas abstraction

↳ e.g. HTML tag

```
<canvas width="256" height="256" style="..." />
```

- default width is 300, height is 150, + bg colour is transparent

↳ e.g.

```
function draw() {
  // create canvas element and add it to the DOM
  const canvas = document.createElement("canvas");
  document.body.appendChild(canvas);

  // set a background style
  canvas.style.setProperty("background", "lightgrey");

  // get graphics context
  const gc = canvas.getContext("2d");
  if (!gc) return;

  // draw something
  gc.fillRect(10, 10, 50, 50);
}

draw();
```

encapsulating in function

need this for all drawing

type narrowing guard

called on page load

SIMPLEKIT

we'll use UI toolkit called SimpleKit for 1st part of course

↳ most basic usage for canvas mode:

1. Import what you need

```
import { startSimpleKit, setSKDrawCallback }
  from "simplekit/canvas";
```

2. Start it up (creates full page canvas, etc.)

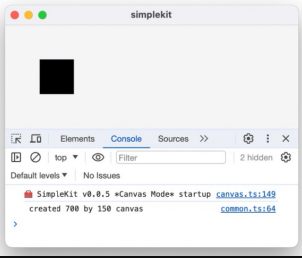
```
startSimpleKit();
```

3. Set the drawing callback

```
setSKDrawCallback((gc) => {
  gc.fillRect(10, 10, 50, 50);
});
```

makes sure it doesn't draw 60 times/s

```
gc.clearRect(0, 0,
             gc.canvas.width,
             gc.canvas.height);
```

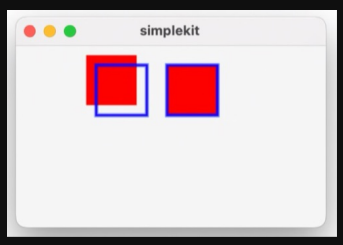


- setSKDrawCallback runs 60 times per s

↳ e.g. rectangleDemo()

```
gc.fillStyle = "red";
gc.fillRect(70, 10, 50, 50);

gc.strokeStyle = "blue";
gc.strokeRect(80, 20, 50, 50);

// stacking to make more complex shapes
gc.lineWidth = 3;
gc.fillRect(150, 20, 50, 50);
gc.strokeRect(150, 20, 50, 50);

// has no effect
gc.strokeStyle = "green";
```

changes state, but nothing is drawn after

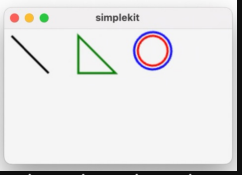


↳ e.g. pathDemo()

```
// line
gc.strokeStyle = "black";
gc.beginPath();
gc.moveTo(10, 10);
gc.lineTo(60, 60);
gc.stroke();

// polyline or polygon
gc.strokeStyle = "green";
gc.beginPath();
gc.moveTo(100, 10);
gc.lineTo(150, 60);
gc.lineTo(100, 60);
gc.closePath();
gc.stroke();

// circle using ellipse
...

// circle using arc
gc.strokeStyle = "red";
gc.beginPath();
gc.arc(200, 30, 20, 0, 2 * Math.PI);
gc.stroke();
```

can also use fill()

line back to first point



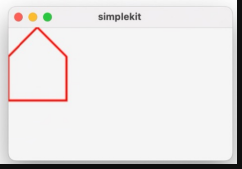
↳ e.g. pathHouseDemo()

```
const housePoints = [
  [40, 0],
  [80, 40],
  [80, 100],
  [0, 100],
  [0, 40],
];

gc.lineWidth = 3;
gc.strokeStyle = "red";

gc.beginPath();
housePoints.forEach((p) => {
  const [x, y] = p;
  gc.lineTo(x, y);
  console.log(`${x}, ${y}`);
});
gc.closePath();
gc.stroke();
```

an array of length 2 arrays

iteration with foreach

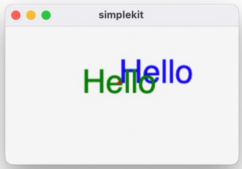
destructuring



↳ e.g. textDemo()

```
const x = 150;
const y = 75;

gc.font = "32pt sans-serif";

// standard alignment
gc.fillStyle = "blue";
gc.fillText("Hello", x, y);

// fully centred alignment
gc.textAlign = "center";
gc.textBaseline = "middle";
gc.fillStyle = "green";
gc.fillText("Hello", x, y);

// dot to show x,y location
gc.fillStyle = "red";
gc.fillRect(x - 2, y - 2, 4, 4);
```

uses CSS font shorthand property syntax



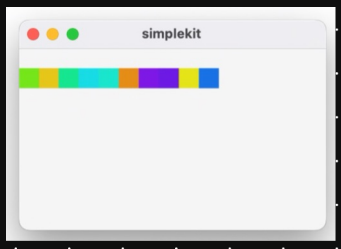
to specify colour, use fillStyle + strokeStyle w/ CSS colour syntax

- **Named** colour (more than 100)
  "red", "blue", "cornflowerblue", "deeppink"
- **Hexadecimal** colour as #RGB, #RRGGBB
  "#f00", "#0000ff", "#6495ed", "#ff1493"
- **RGB**: Red, Green, Blue
  "rgb(255 0 0)", "rgb(100, 149, 237)"

- **HSL**: Hue, Saturation, Luminance
  `"hsl(0deg 100% 100%)"`, `"hsl(219deg 58% 93%")`
- (many other formats and variations)

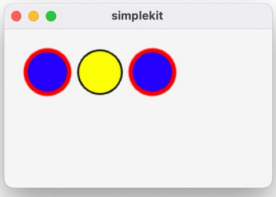
↳ e.g. colourDemo()

```
for (let i = 0; i < 10; i++) {
   const h = Math.random() * 360;
   gc.fillStyle = `hsl(${h}deg 80% 50%)`;   // string template literal
   gc.fillRect(i * 20, 20, 20, 20);
```



to save + restore state of drawing styles
↳ save() pushes curr drawing state to stack
↳ restore() pops last saved drawing state + restores it
↳ e.g saveState()

```
const circle = (x, y) => {...};   // local function to draw circles

gc.fillStyle = "blue";
gc.strokeStyle = "red";
gc.lineWidth = 5;
circle(50, 50);

gc.save();   // save state: fill = blue, stroke = red, lineWidth = 5

gc.fillStyle = "yellow";
gc.strokeStyle = "black";
gc.lineWidth = 2;
circle(110, 50);

gc.restore();   // restore state back to: fill = blue, stroke = red, lineWidth = 5

circle(170, 50);
```



DRAWABLE OBJECTS

drawing using gc API can be tedious, so instead
1) define interface for obj that can be drawn

```
export interface Drawable {
  draw: (gc: CanvasRenderingContext2D) => void;
}
```

2) define drawable objs

```
export class MyShape implements Drawable {
  ...
  draw(gc: CanvasRenderingContext2D) {
    // drawing commands go here
  }
}
```

3) create obj + draw using curr gc

```
const myShape = new MyShape( ... )
myShape.draw(gc);
```

e.g. squareDemo()

```
export class Square2 implements Drawable {
  constructor(
    public x: number,
    public y: number,
    public size: number,
    public fill?: string, // optional parameters
    public stroke?: string,
    public lineWidth?: number
  ) {}

  draw(gc: CanvasRenderingContext2D) {
    gc.beginPath();
    if (this.fill) gc.fillStyle = this.fill;
    if (this.stroke) gc.strokeStyle = this.stroke;
    if (this.lineWidth) gc.lineWidth = this.lineWidth;
    gc.rect(
      this.x - this.size / 2,
      this.y - this.size / 2,
      this.size,
      this.size
    );
    if (this.fill) gc.fill();
    if (this.lineWidth) gc.stroke();
  }
}
```

gc.save()

gc.restore()

painter's algo: draw more complex shapes by combining primitives + drawing back to front

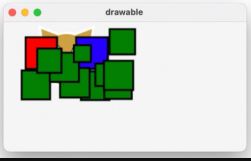
↳ layering img

↳ draw w/ coords in convenient frame (e.g. around (0,0)) + transform shapes to desired loc

↳ keep ordered display list of Drawable objs

- add objs to arr from back to front (or add z-depth field + sort)
- to draw all objs, iterate thru list + draw each
- e.g. displayListDemo()

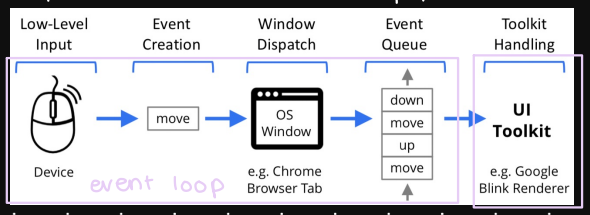
```
export class DisplayList {
  list: Drawable[] = [];

  add(drawable: Drawable) {
    this.list = [...this.list, drawable];
  }

  remove(drawable: Drawable) {
    this.list = this.list.filter((d) => d !== drawable);
  }

  draw(gc: CanvasRenderingContext2D) {
    this.list.forEach((d) => {
      d.draw(gc);
    });
  }
}
```



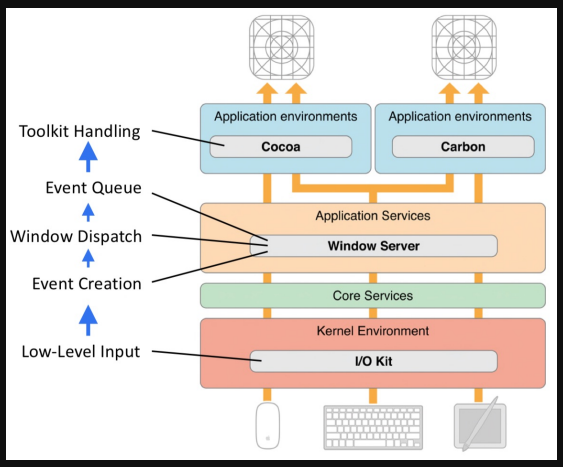
INPUT EVENTS

EVENT-DRIVEN PROGRAMMING

- event-driven programming bases program execution flow on events
- **event**: msg to notify an app that smth has happened
- event types:
  - ↳ device input
  - ↳ window input
  - ↳ system (e.g. timer)
  - ↳ app (e.g. remote data loaded over network)
- OS polls input device state & communicates changes in form of events to active app window
  - ↳ **input event architecture pipeline**:



  - ↳ e.g. OSX event architecture:

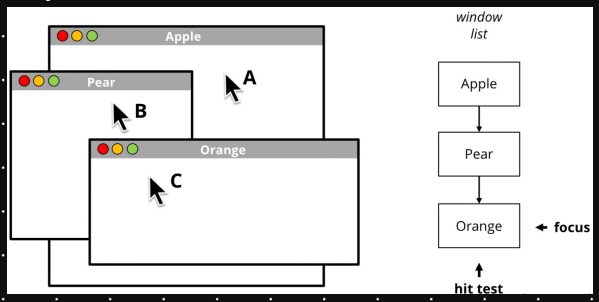


- **low-lvl input**:
  - ↳ most mice & keyboards conform to **Human Interface Devices (HID) std**, where each device reports to OS what data will be sent (aka boot report format)
  - ↳ OS polls device to get curr state, typically every 8ms (125 Hz)
  - ↳ OS filters & transforms input data
- **event creation**:
  - ↳ each transformed low-lvl input is **state**, not event
  - ↳ window manager gens events when state changes
  - ↳ **fundamental low-lvl input** events (aka raw events):
    - keydown

- keyup
- mousedown
- mouseup
- mousemove

↳ each event has timestamp

- **window dispatch**: windowing system maintains list of all windows ordered from back to front, where frontmost window has **focus**
  - ↳ hit test using mousedown pos changes focus
  - ↳ events sent to focused window
    - exceptions include global hooks + overlays
  - ↳ e.g.



- **event queue**: buffer btwn user + each window
  - ↳ lets UI toolkit running in window handle events efficiently
    - toolkit should refer to **event.timestamp**, not when event was pulled off queue or handled by app code.
- OS windowing system continually runs **event loop**, which comprises of 1st 4 events in input event pipeline:

```
loop:
  poll input devices
  create fundamental events
  dispatch event to focused window
  add event to window event queue
```

TOOLKIT RUN LOOP

- UI toolkit handles OS events in its own **run loop**
  - ↳ checks for fundamental events in event queue
  - ↳ calls animation timers, re-renders UI, etc.
- SimpleKit simulates windowing system by using HTML DOM events to create fundamental events
  - ↳ **createWindowingSystem**
    - creates a *shared* fundamental event queue

```
const eventQueue: FundamentalEvent[] = [];
```

- listens to 6 DOM events to use as simulated fundamental events

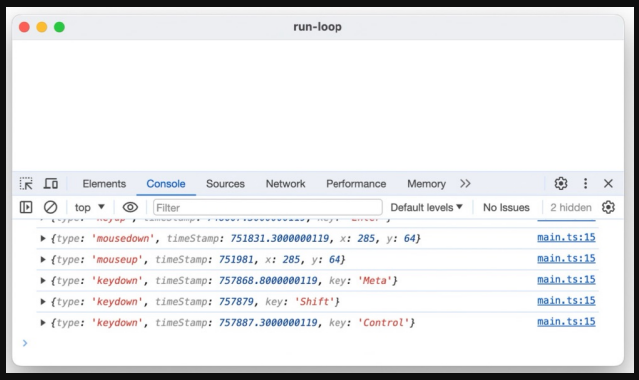
```
window.addEventListener("mousedown", saveEvent);
...
```

- calls a toolkit **run loop** function at approximately 60 Hz

```
export type RunLoopHandler = (
  eventQueue: FundamentalEvent[],
  time: DOMHighResTimeStamp
) => void;
```

- e.g. run-loop

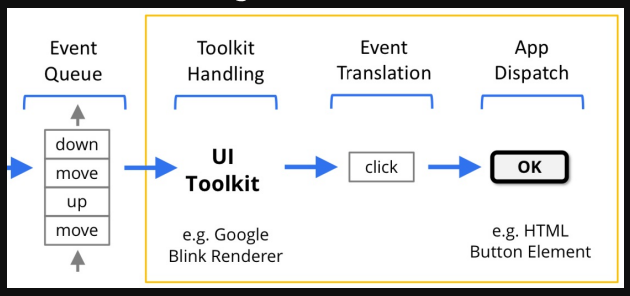
```ts
// creates very basic UI Toolkit run loop

import {
  FundamentalEvent,
  createWindowingSystem,
} from "simplekit/windowing-system";

// this is a very simple UI toolkit run loop
function runLoop(eventQueue: FundamentalEvent[], time: number) {
  // log all fundamental events in the queue
  while (eventQueue.length > 0) {
    const e = eventQueue.shift();
    if (!e) continue;

    console.log(e);
  }

  // many other UI toolkit things in here ...
}

// create the simulated windowing system with
// this UI Toolkit run loop
createWindowingSystem(runLoop);
```



EVENT TRANSLATION

fundamental OS events are translated by UI toolkit into higher-lvl events before dispatching them to app

↳



- higher-lvl events include click, dblclick, + drag
  - ↳ modelled as state machines
- e.g. SimpleKit event classes (simplekit/events/events.ts)

showing simplified forms of real classes

```ts
export class SKEvent {
  constructor(
    public type: string,
    public timeStamp: number,
    ...
  ) {} }

export class SKMouseEvent extends SKEvent {
  constructor(
    ...
    public x: number,
    public y: number,
    ...
  ) {} }

export class SKKeyboardEvent extends SKEvent {
  constructor(
    ...
    public key: string | null = null,
    ...
  ) {} }
```

- e.g. translation
  - ↳ run-loop.ts

```
// list of toolkit events to dispatch
let events: SKEvent[] = [];

// translate fundamental events to toolkit events
while (eventQueue.length > 0) {
  const fundamentalEvent = eventQueue.shift();
  if (!fundamentalEvent) continue;

  translators.forEach((t) => {
    const translatedEvent = t.update(fundamentalEvent);
    if (translatedEvent) {
      events.push(translatedEvent);
    }
  });
}
```

run-loop processes a list of translators

↳ translators.ts

```
export type EventTranslator = {
  update: (fe: FundamentalEvent) => SKEvent | undefined;
};


export const myTranslator = {
  someStateProperty,
  ...
  update(fe: FundamentalEvent): SKEvent | undefined {
    ...
  },
};
```

most translators need to track state over time

can also return specific events inherited from SKEvent, like SKMouseEvent

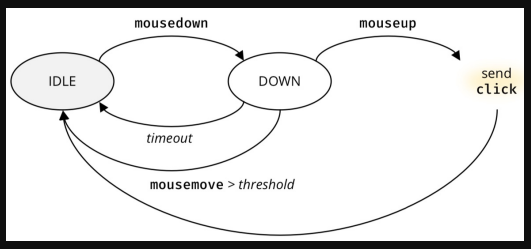
Some translators frequently return undefined

- e.g. fundamentalTranslator implements EventTranslator type

```
export const fundamentalTranslator = {
  update(fe: FundamentalEvent): SKEvent {
    switch (fe.type) {
      case "mousedown":
      case "mouseup":
      case "mousemove":
        return new SKMouseEvent(fe.type, fe.timeStamp,
          fe.x || 0, fe.y || 0);
        break;
      case "keydown":
      case "keyup":
        return new SKKeyboardEvent(fe.type, fe.timeStamp,
          fe.key);
        break;
      ...
```

mouse click state machine (SM): mousedown followed by mouseup within short time + little movement

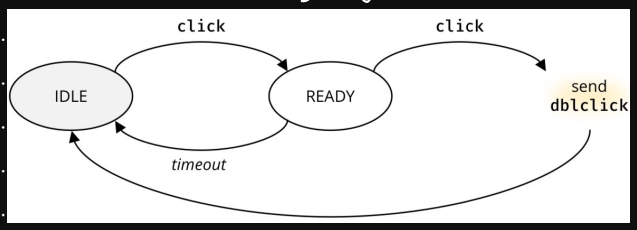
↳



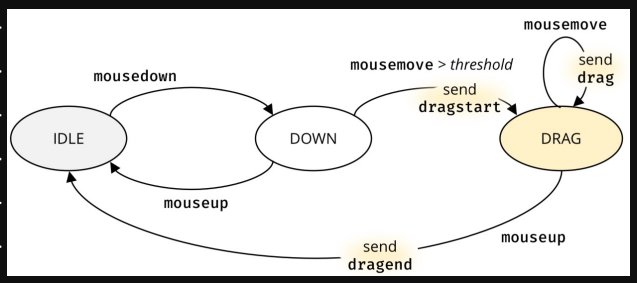
double click SM: click followed by another click in short time interval

↳ init click will also be sent so interactive design must accomodate click followed immediately by dblclick

↳



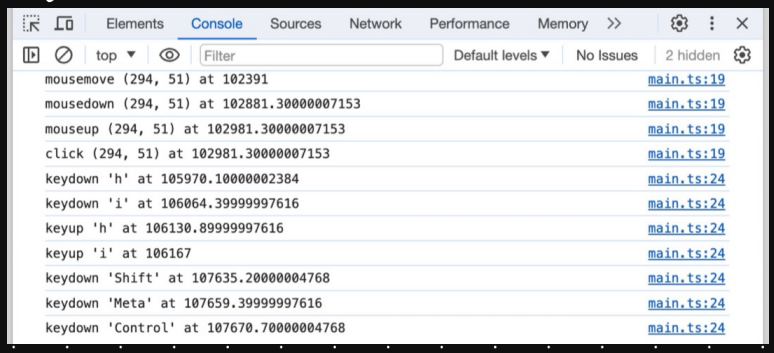
mousedrag SM: dragstart event sends when mouse held + moves more than small amt, then mouseup from dragging state sends dragend event
↳ in dragging state, each mousemove triggers another drag event
↳



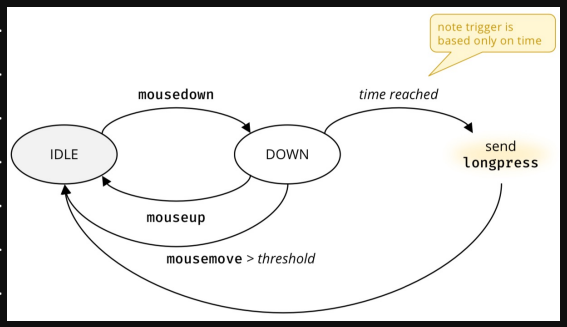
translated events will have same timestamp as triggering events
- e.g. click has same timestamp as mouseup

all translated events dispatched in deterministic order (i.e. app can assume seq. if listening to multiple events)
- e.g. click comes after mouseup

e.g. translation/main.ts



longpress SM: mousedown followed by a little movement + no mouseup for long time
↳



some translators need time even when there's no fundamental events
↳ e.g. longpress w/o moving mouse after mousedown
↳ soln is to send null event when there's no other events

```
if (eventQueue.length == 0) {
  eventQueue.push({
    type: "null",
    timeStamp: time,
  } as FundamentalEvent);
}

// translate fundamental events to toolkit events
while (eventQueue.length > 0) {
  const fundamentalEvent = eventQueue.shift();
  ...
```

- discrete state changes aren't high freq (< 5Hz) so each state transition is important
  - ↳ e.g. mousedown + mouseup
- continuing state are generated at high freq (> 60Hz) so each state transition isn't as important
  - ↳ e.g. mousemove
  - ↳ toolkit might be unable to consume them as quickly as they're generated so multiple events can be coalesced (i.e. remove/combine intermediate events since last update)
    - avoid coalescing if we want precise movement trajectory w/o interactive feedback (e.g. saving signature)
- to send UI toolkit events to app, we typically send to specific part of UI based on hit testing + hierarchal data struct (e.g. DOM, widget tree)
- use simple dispatch method for now
  - ↳ all events handled in single app fcn
  - ↳ e.g.

```
function handleEvent(e: SKEvent) { ... }
```

- use simple binding method for now
  - ↳ app code to handle each event is in event handling fcn
  - ↳ e.g.

```
switch (e.type) {
   case "mousemove":
   // app code here for mousemove
   break;
   case "click":
   // app code here for click
   break;
   ...
```

- UI state + UI drawing code are separated

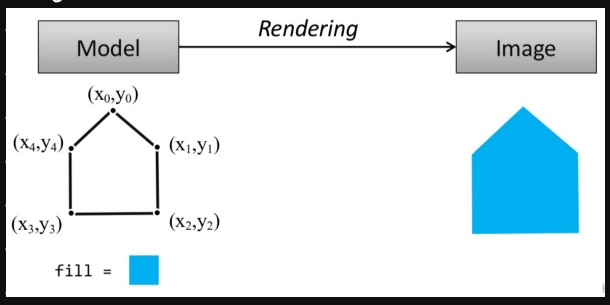
HIT-TESTING

SHAPE MODELS

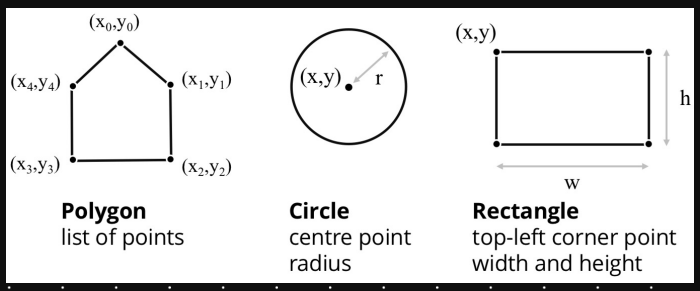
model: mathematical rep of rendered shape
- ↳ geometry
- ↳ visual style properties
- ↳ transformations
- ↳ ops (e.g. event handling, hit-testing)

rendering: process to translate model into img

img: rendered shape based on underlying model



diff shapes have diff geometric reps
- ↳ e.g.

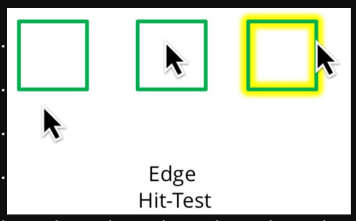


- ↳ many alt reps + other kinds of shapes
- ↳ shape models can be combos of shapes

HIT-TESTING

inside hit-test: is mouse cursor inside shape?
- ↳ closed shapes
- ↳ usually when rendered w/ fill
- ↳ e.g.

edge hit-test: is mouse cursor on shape stroke?
- ↳ open + unfilled shapes
- ↳ e.g.



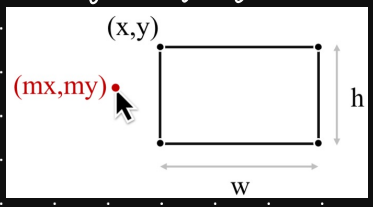
hit-test is specific to shape type + properties
↳ if edge, need to factor in stroke thickness
↳ e.g.

```
function hitTest(
  mx: number,        // mouse position
  my: number,

  // shape properties go here

  strokeWidth: number   // rendering properties needed for some hit-tests
  ): Boolean {
  ...
}
```

rectangle inside hit-test:
↳ given:
- mouse pos (mx, my)
- rectangle top-left corner (x, y)
- rectangle width w + height h

↳ inside hit T when:
- mx ∈ [x, x+w]
- my ∈ [y, y+h]

↳



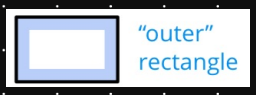
↳ e.g.

```
function insideHitTestRectangle(
  mx: number,
  my: number,
  x: number, y: number,     // rectangle shape properties
  w: number, h: number
) {
  return mx >= x &&
         mx <= x + w &&
         my >= y &&
         my <= y + h
}
```

rectangle edge hit-test:
↳ given:
- mouse pos (mx, my)
- rectangle top-left corner (x, y)
- rectangle width w + height h
- stroke width s

↳ edge hit T when:

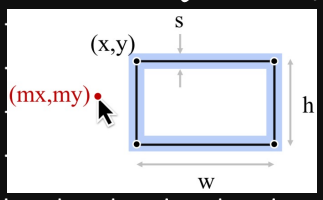
- $mx \in [x - \frac{s}{2}, x + w + \frac{s}{2}]$
- $my \in [y - \frac{s}{2}, y + h + \frac{s}{2}]$



↳ edge hit F when:

- $mx \in (x + \frac{s}{2}, x + w - \frac{s}{2})$
- $my \in (y + \frac{s}{2}, y + h - \frac{s}{2})$

↳



↳ e.g.

```
function edgeHitTestRectangle(
  mx: number,
  my: number,
  x: number, y: number,
  w: number, h: number,
  strokeWidth: number
) {
  // width of stroke on either side of edges
  const s = strokeWidth / 2;

  // inside rect after adding stroke
  const outer = mx >= x - s && mx <= x + w + s &&
                my >= y - s && my <= y + h + s;

  // but NOT inside inner rect after subtracting stroke
  const inner = mx > x + s && mx < x + w - s &&
                my > y + s && my < y + h - s;

  return outer && !inner;

}
```

circle inside hit-test:

↳ given:

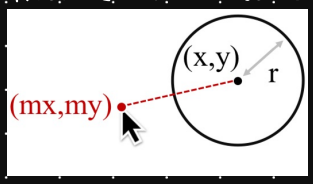
- mouse pos (mx, my)
- circle centre (x, y)
- circle radius r

↳ calc dist from (mx, my) to (x, y)

- i.e. Euclidean dist btwn pts

↳ inside hit T when dist ≤ r

↳



circle edge hit-test:

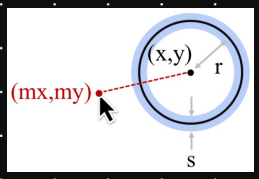
↳ given:

- mouse pos (mx, my)
- circle centre (x, y)
- circle radius r
- stroke weight s

↳ calc dist from (mx, my) to (x, y)

↳ edge hit T when dist $\in [r - \frac{s}{r}, r + \frac{s}{r}]$

↳



line edge hit-test:

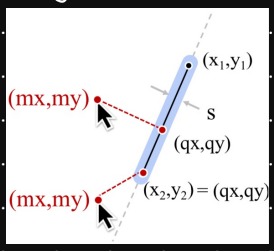
↳ given:

- mouse pos (mx, my)
- line start (x1, y1)
- line end (x2, y2)
- stroke weight s

↳ calc closest pt on line segment (qx, qy) + dist from (mx, my) to (qx, qy)

↳ edge hit T when dist $< \frac{s}{2}$

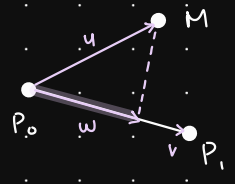
↳



↳ e.g.

```
function hitTestLine(
  mx: number,
  my: number,
  p0x: number, p0y: number,
  p1x: number, p1y: number,
  strokeWidth: number
) {
  // edge hit-test
  const m = point(mx, my);
  const q = closestPoint(m,
                point(p0x, p0y), point(p1x, p1y));
  const d = distance(m.x, m.y, q.x, q.y);
  if (d < strokeWidth / 2) return true;

  // no hit
  return false;
}
```

find closest pt Q on line w/vector projection:



M : mouse pos

P_0P_1 : line

$\vec{u} = M - P_0$

$\vec{v} = P_1 - P_0$

Proj w of u onto v:

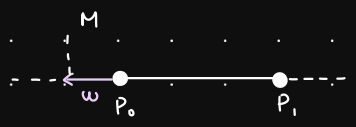
$\vec{w} = \frac{u \cdot v}{v \cdot v} \vec{v}$

$s = \frac{u \cdot v}{v \cdot v}$ ← s is scalar of w

$\vec{w} = s\vec{v}$

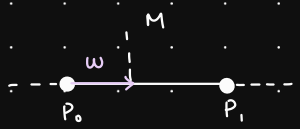
Use s to find closest pt on line P_0P_1:

↳ if $s<0$, $Q = P_0$



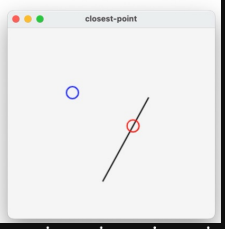
↳ if $s>1$, $Q = P_1$

↳ if $s \geq 0$ & $s \leq 1$, $Q = P_0 + w$



↳ e.g. closestPoint / closestpoint ts

```ts
function closestPoint(m: Point2, p0: Point2, p1: Point2): Point2 {
  const v = p1.subtract(p0); // v = P1 - P0

  // early out if line is less than 1 pixel long
  if (v.magnitude() < 1) return p0.clone();

  const u = m.subtract(p0); // u = M - P0

  // scalar of vector projection ...
  const s = u.dot(v) / v.dot(v);

  // find point for constrained line segment
  if (s < 0) {
    return p0.clone();
  } else if (s > 1) {
    return p1.clone();
  } else {
    const w = v.multiply(s); // w = s * v
    return p0.add(w); // Q = P0 + w
  }
}
```



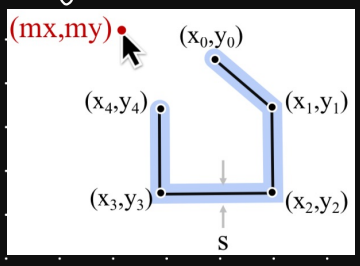
polyline edge hit-test:

↳ given:

- mouse pos (mx, my)
- list of pts
- stroke weight s

↳ edge hit T when T for any line segment

↳



polygon edge hit-test:

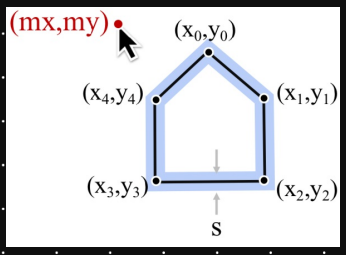
↳ given:

- mouse pos (mx, my)

- list of pts
- stroke weight s

↳ edge hit T when T for any line segment

↳



↳ e.g.

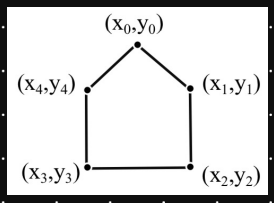
```
function edgeHitTestPolygon(
  mx: number,
  my: number,
  points: Point2[],
  strokeWidth: number
) {
  const m = new Point2(mx, my);
  // assume shape is closed, so start with segment
  // from last point to first point
  let p0 = points.slice(-1)[0];
  for (let p1 of points) {
    if (hitTestLine(m.x, m.y,
                    p0.x, p0.y, p1.x, p1.y,
                    strokeWidth))
      return true;
    p0 = p1;
   }
   return false;
  }
```

polygon inside hit-test:

↳ given:

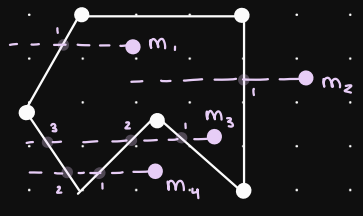
- mouse pos (mx, my)
- list of pts

↳

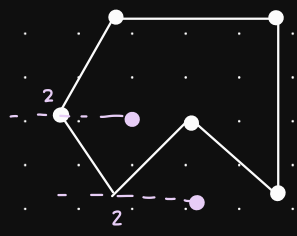


↳ inside hit-test T when we cast horizontal ray from mouse pos + #times it intersects line segments of polygon is **odd** (for most cases)

	Num	Result
m_1	1	in
m_2	2	out
m_3	3	in
m_4	2	out



- problem occurs when ray intersects w/ pt, it's counted as intersection w/ 2 segments
  - → don't know if mouse is in / out



  - → use unit vectors rep dir of 2 segments to decide if it's counted as 1 / 2 intersections

y test vector (y-dir of mouse pos)

- if sign $v_1 \cdot t = v_2 \cdot t$, count as 2 intersections
- if sign $v_1 \cdot t \neq v_2 \cdot t$, count as 1 intersection

- Shape class has geometry that defines shape
  - ↳ geometry properties (e.g. isFilled, isStroked)
  - ↳ visual style properties (e.g. fill, stroke, strokeWeight)
  - ↳ method to draw into provided graphics context
  - ↳ method to do hit-testing w/ x, y cursor pos
  - ↳ e.g. Shape base class

```
abstract class Shape {
  fill: string = "grey";
  stroke: string = "black";
  strokeWidth = 1;

  get isFilled() {
    return this.fill != "";
  }

  get isStroked() {
    return this.stroke != "" && this.strokeWidth > 0;
  }

  abstract draw(gc: CanvasRenderingContext2D): void;

  abstract hitTest(mx: number, my: number): boolean;
}
```

- e.g. Rectangle shape model

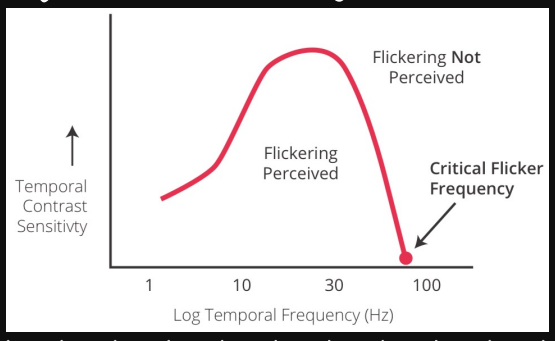
```
class Rectangle extends Shape {
  constructor(
    public x: number, public y: number,
    public w: number, public h: number
  ) { super(); }

  draw(gc: CanvasRenderingContext2D) {
    ...
  }
```

```
  hitTest(mx: number, my: number) {
    let hit = false;
    if (this.isFilled) {
      hit ||= insideHitTestRectangle(mx, my, ...);
    }
    if (this.isStroked) {
      hit ||= edgeHitTestRectangle(mx, my, ... );
    }
    return hit;
  }
}
```

- **shape models** useful b/c they encapsulate related info + fcns
  - ↳ manipulate them dynamically
    - transform pos, size, orientation
    - change shape properties
  - ↳ use multiple instances of same shape
- **optimizations** to hit-testing:
  - ↳ avoid square root in dist calcs
  - ↳ use simpler + less precise hit-test first for early reject
    - e.g. bounding box
  - ↳ split scene into cells + track which ones each shape is in

ANIMATION

- **animation** is simulation of movement using series of imgs
- **frame**: each img / state of animation seq
- **frame rate**: # frames to display per sec
- **easing**: fcn that controls how tweening is calc
- **key frame**: defines beginning & end of tween
- typically animate numerical params that change how graphics are drawn over time
- frame rate is measured in **frames-per-sec (fps)**
  - ↳ can be expressed in Hertz (Hz)
    - e.g. 60 fps = 60 Hz
- **critical flicker frequency (CFF)**: when perception of intermittent light source changes from flickering to continuous light
  - ↳



  - ↳ CFF typically 60-90Hz
- SimpleKit lets us define single **callback for animation**
  - ↳ callback passed as curr time in ms
  - ↳ e.g. in simplekit

```
type AnimationCallback = (time: number) => void;

function setSKAnimationCallback(animate: AnimationCallback) ...
```

  - ↳ e.g. in program

```
setSKAnimationCallback((time) => { /* animate in here */ });
```

- animation can be created thru real time **simulation** using fcns, conditionals, etc
  - ↳ typically just loops / continues
- e.g. simulation

```
// if it hits the edge of the box, change direction
if (dot.x < dot.r || dot.x > box.width - dot.r) {
    dx *= -1.0;
}
...

// update the dot position
dot.x += dx;
dot.y += dy;
```

- **timer** is simple animation that needs:
  - ↳ duration
  - ↳ start time
  - ↳ update fcn to check if timer's finished
  - ↳ method to check if timer's running
  - ↳ usually callback fcn to call when timer's finished
- e.g. simpleTimerDemo

```ts
/**
 * A timer that you have to check if it's running
 */
export class BasicTimer {
  constructor(public duration: number) {}

  private startTime: number | undefined;

  start(time: number) {
    this.startTime = time;
    this._isRunning = true;
  }

  get isRunning() {
    return this._isRunning;
  }
  private _isRunning = false;

  update(time: number) {
    if (!this._isRunning || this.startTime === undefined) return;

    const elapsed = time - this.startTime;
    if (elapsed > this.duration) {
      this.startTime = undefined;
      this._isRunning = false;
    }
  }
}
```

- e.g. callbackTimerDemo

```ts
/**
 * A timer that calls a callback when it finishes
 */
export class CallbackTimer extends BasicTimer {
  constructor(
    public duration: number,
    public callback: (t: number) => void
  ) {
    super(duration);
  }

  update(time: number) {
    if (this.isRunning) {
      super.update(time);
      // if state switches from running to not running, call the callback
      if (!this.isRunning) {
        this.callback(time);
      }
    }
  }
}
```

- **tweening**: interpolation btwn keyframes to create individual frames
  - ↳ keyframes are #s

↳ params:
- startVal (keyframe 1)
- endVal (keyframe 2)
- duration for tween
- startTime for when tween begins

↳ proportion of time completed so far: `t = (time - startTime) / duration`

↳ interpolate start + end val to get current tweened val:
`value = startValue + (endValue - startValue) * t`

linear interpolation (lerp) fcn smoothly interpolates changes from 1 val to another

↳ e.g.

```
// linear interpolation from start to end
// at normalized time t (in [0, 1])
const lerp = (start: number, end: number, t: number) =>
  start + (end - start) * t;
```

easing fcns control how tweening is calc

↳ lerp gens linear change in val, but easing fcn can change how val changes over time

↳ e.g.

Type
```
type EasingFunction = (t: number) => number;
```

Common functions
```
const flip = (t) => 1 - t;

const easeOut = (t) => Math.pow(t, 2);

const easeIn = (t) => flip(easeOut(flip(t)));

const easeInOut = (t) => lerp(easeOut(t), easeIn(t), t);
```
exponent changes the "amount" of easeOut

UI TOOLKIT ANIMATION ARCHITECTURE

some toolkits provide animation manager that allows programmer to set animation + not explicitly manage

↳ keeps list of active animations

↳ updates each animation every frame

↳ removes animations when finished

e.g. manager

```
class AnimationManager {
  protected animations: Animater[] = [];

  add(animation: Animater) {
    this.animations.push(animation);
    ...
  }
```

```ts
update(time: number) {
  // update every animation currently running
  this.animations.forEach(
      (a) => a.update(time));

  // remove animations that finished
  this.animations =
    this.animations.filter(
      (a) => a.isRunning);
}
```

Animation 21

- tween is 2 keyframes:
  - ↳ keyframe 1: start time, start val
  - ↳ keyframe 2: end time, end val
- seq of keyframes enables animations over time
  - ↳ find keyframe i + keyframe (i+1) for curr time
    - (time > keyframe[i].time) && (time < keyframe[i+1].time)
  - ↳ tween val i + val (i+1)
- e.g.

keyframes

	time	targetValue
0:	0	100
1:	1000	300
2:	2500	400
3:	5000	50
4:	6000	50
5:	6500	100

Example 1
when time is 800,
tween keyframe 0 and keyframe 1:

```
t = (800 - 0) / (1000 - 0)
value = 100 + t * (300 - 100)
```

Example 2
when time is 3000,
tween keyframe 2 and keyframe 3:

```
t = (3000 - 2500) / (5000 - 2500)
value = 400 + t * (50 - 400)
```

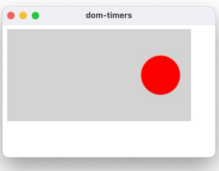
Practice:
when time is 5250
value = 50 b/c animation is paused
when time is 7000
same as end time (t = 6500)

- when doing animation using built-in timers, timer triggers event after some time period
  1. set time interval for desired frame rate
     - e.g. 30 fps has interval of $\frac{1}{30}$ s
  2. in timer finished event handler:
     - update params you want to animate
     - optionally redraw img for next frame
  3. restart timer for next interval
- DOM/HTML engine has general-purpose interval timer
  - ↳ e.g. demoIntervalTimer

```ts
const duration = 2000;
let timer = setInterval(() => {
  const timePassed = performance.now() - start;
  dot.x = lerp(50, 250, timePassed / duration);
  gc.clearRect(0, 0, canvas.width, canvas.height);
  dot.draw(gc);
  // stop after certain time
  if (timePassed > duration) {
    clearInterval(timer);
  }
}, 1000 / 60);
```

1000 ms / 60 frames = 16.666 ms/frame = 60 FPS

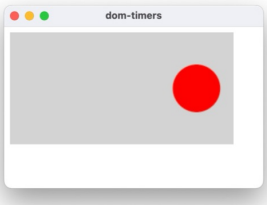
Check assignment specification, using this timer may not be allowed.



Animation 26

DOM/HTML engine also provides special animation callback
↳ SK uses to create run-loop
↳ e.g. demoRequestAnimationFrame

```
const duration = 2000;
requestAnimationFrame(function animate(timePassed) {
  dot.x = lerp(50, 250, timePassed / duration);
  gc.clearRect(0, 0, canvas.width, canvas.height);
  dot.draw(gc);
  // continue unless done animation
  if (timePassed < duration) {
    requestAnimationFrame(animate);
  }
});
```

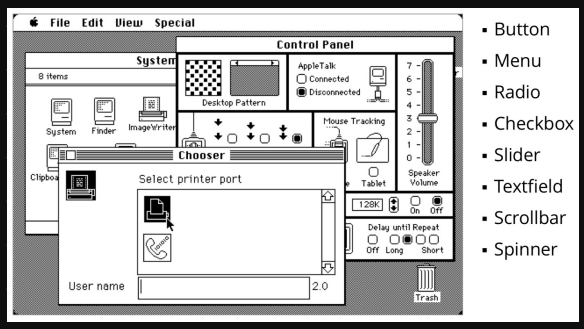


Check assignment specification,
using this timer may not be allowed.

most UI frameworks are single-threaded to avoid deadlocks & race conds due to unpredictable user-generated events

WIDGETS

- **widget** : generic name for parts of interface that have their own behaviour
  - ↳ aka components, controls, elmts, views, etc.
  - ↳ e.g.

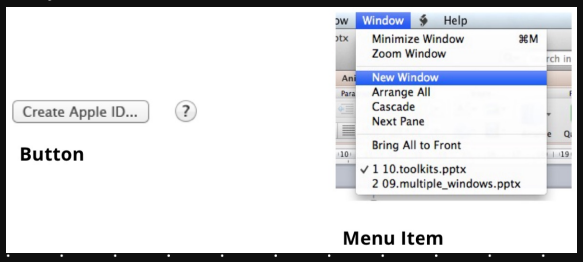


- Button
- Menu
- Radio
- Checkbox
- Slider
- Textfield
- Scrollbar
- Spinner

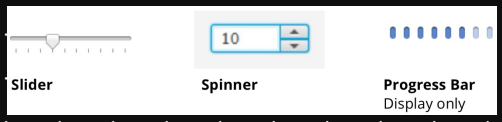
- **widget fcns** :
  - ↳ display user feedback (like a View)
    - convey curr state & when action is triggered
  - ↳ handle user input (like a Controller)
    - capture user input & gen events

LOGICAL INPUT DEVICES

- **logical input device** describes widget based on fcn
  - ↳ widget is specific instance of logical input device
  - ↳ category of widgets is based on common fcnality
    - e.g. logical btn devices gen action events
- logical **button** device : input single action + send "action happened" event
  - ↳ e.g.

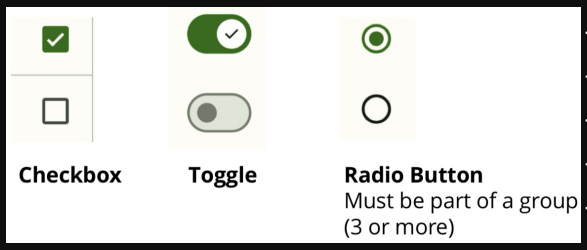


- logical **number** device : input & display numeric val
  - ↳ usually in certain range
  - ↳ may support validation
  - ↳ sends "val changed" event
  - ↳ e.g.

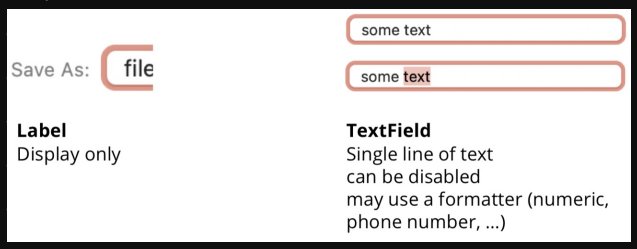


- logical **boolean** device : input & display bool val

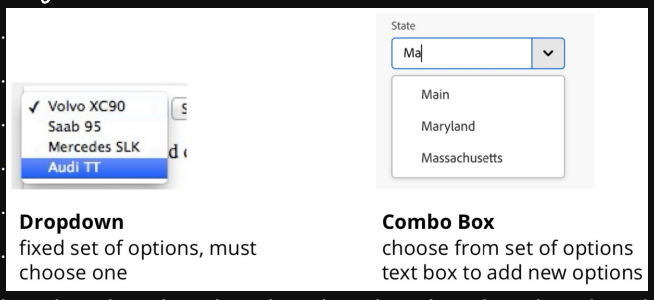
↳ sends "val changed" event
↳ e.g.



- logical `text` device: display + edit text val
  ↳ sends "val changed" event
  ↳ e.g.



- logical `choice` device: input + display | choice among list
  ↳ sends "selection changed" event
  ↳ e.g.



WIDGET IMPLEMENTATION

- `base widget class` has location, size, inside rectangle hitTest(), + abstract draw()
  ↳ use "props" method for ctor arguments
  ↳ e.g. element.ts

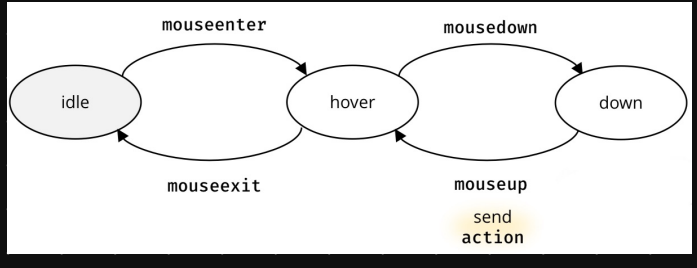
```ts
import { insideHitTestRectangle } from "simplekit/utility";

const minElementSize = 32;

// element properties
export type SKElementProps = {
  x?: number;
  y?: number;
  width?: number;
  height?: number;
};
```

```ts
// element is a base widget class
export abstract class SKElement {
  constructor({
    x = 0,
    y = 0,
    width = minElementSize,
    height = minElementSize,
  }: SKElementProps = {}) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
  }
```

- `props` method
  - ↳ can create obj w/ that type
  - ↳ destructure local vars from props obj
  - ↳ use props type as arg to fcn
  - ↳ e.g.

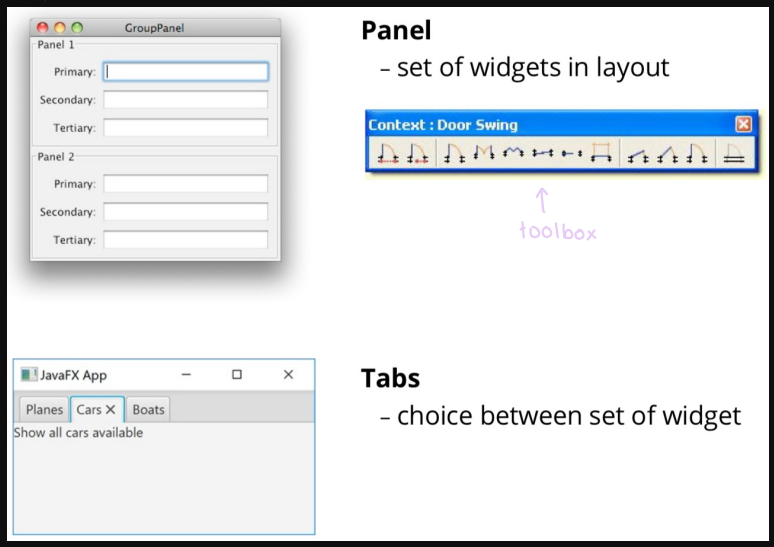
```ts
{
  // the props object type
  type MyProps = {
    name: string;
    city?: string;
    age: number;
    num?: number;
  };

  // a props object
  const props: MyProps = { name: "Harry", age: 21 };

  // destructure props
  const { name, city = "Waterloo", age, num = 0 } = props;

  console.log(name, city, age, num);

  // use props as function argument
  function func({ name, city = "Waterloo", age, num = 0 }: MyProps) {
    console.log(name, city, age, num);
  }

  func({ name: "Harry", age: 21 });

  func({ name: "Luna", city: "Sackville", age: 22, num: 2 });

  func({ name: "Draco", city: "Toronto", age: 20 });
}
```

- e.g. button
  - ↳ draws based on bubble state
  - ↳



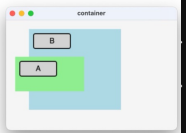
- e.g. label
  - ↳ alignment applies when width > text
  - ↳ uses `MeasureText` util fcn in SK
    - uses gc.measureText()
    - measures rendered text in canvas buffer
    - caches canvas buffer
    - height is ascent + descent
    - 

- e.g. textfield
  - ↳ focus state
  - ↳ text prop get / set
  - ↳ textWidth + method to draw cursor when focused
  - ↳ applyEdit method
- e.g. `container` widgets



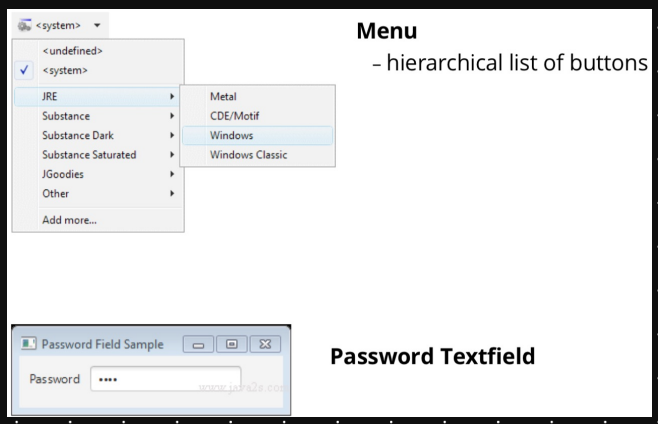
- `SKContainer` holds child widgets
  - ↳ children are drawn relative to container pos
  - ↳ containers create `widget tree` (similar to DOM)
  - ↳ e.g.

```
children: SKElement[] = [];

addChild(element: SKElement) {
  this.children.push(element);
}

removeChild(element: SKElement) {
  this.children =
   this.children.filter((el) =>
     el != element);
}
```



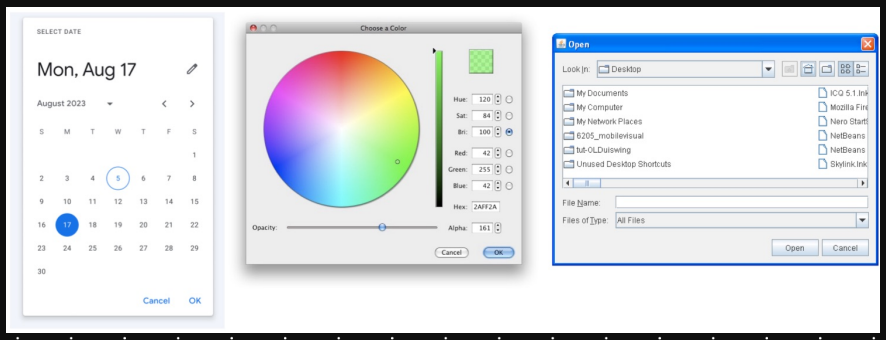
- buttonB is child of blueContainer
- buttonA is child of greenContainer

e.g. specialized widgets



↳ some special val widgets are date, colour, & file pickers

- e.g.



widget features that are desirable in widget toolkit:

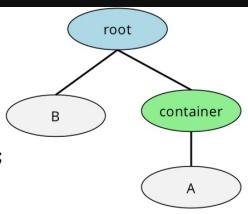
↳ completeness
↳ consistency
↳ customizability

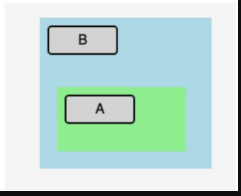
DISPATCH

EVENT DISPATCH

- UI toolkits typically organize widgets into **tree**
  - ↳ 1 root elmt
  - ↳ container widgets for non-leaf nodes
  - ↳ child order dictates draw order (e.g. left to right)
- e.g. container

```
const blueContainer = new SKContainer( ... );
blueContainer.fill = "lightblue";

const buttonB = new SKButton( ... );

const greenContainer = new SKContainer( ... );
greenContainer.fill = "lightgreen";

const buttonA = new SKButton( ... );

// build the UI tree
blueContainer.addChild(buttonB);
blueContainer.addChild(greenContainer);
greenContainer.addChild(buttonA);
```



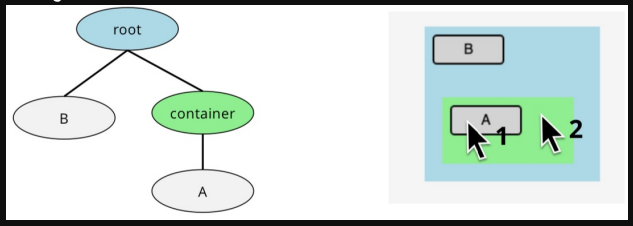


- **event dispatch** is to route event to appropriate widget / code
  1) target selection
     - frontmost widget under mouse
  2) route construction
     - path from root to target node
  3) propagation
     - capture down from root to target
     - bubble up from target to root

  } only apply to positional dispatch
- **target selection** is determined by type of event
  - ↳ mouse event: node at loc of cursor
    - **positional dispatch**
  - ↳ key event: node that has focus
    - **focus dispatch**
  - ↳ touch event: more complex target selection
    - e.g. continuous gesture like pinch-to-zoom, swipe

target selection in positional dispatch is last widget drawn under mouse

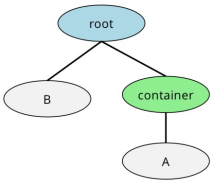
- ↳ e.g. 1 is buttonA + 2 is greenContainer

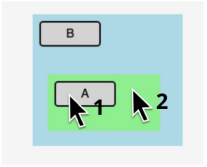


- e.g. route construction in positional dispatch

▪ mouse event at position 1:
- blueContainer , greenContainer, buttonA,

▪ mouse event at position 2:
- blueContainer , greenContainer





↳ target-route

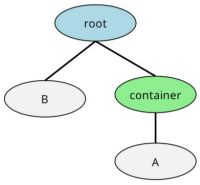
NOTE: simplified TypeScript

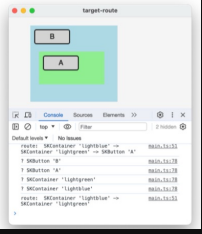
```
// returns list of elements under mouse (from back to front)
function buildTargetRoute(mx, my, SKElement) {
  route = [];
  if (element instanceof SKContainer) {
    element.children.forEach((child) =>
      route.push(
        ...buildTargetRoute(
          mx - element.x,
          my - element.y,
          child
        )
      )
    );
  }

  if (element.hittest(mx, my)) {
    return [element, ...route];
  } else {
    return route;
  }
}
```

visit children from back to front

add this element to route if hit

o.w. just return route





EVENT BINDING

- difficult to demo propagation w/o talking abt event binding, which is how events are associated w/code
- common implementations:
  - ↳ switch stmt binding in run loop: all events consumed in 1 central app event loop
    - loop checks for any events & switch selects code to handle them
    - e.g.

```
while( true ) {
    XNextEvent(display, &event); // wait next event
    switch(event.type) {
    case Expose:
        // ... handle expose event ...
        cout << event.xexpose.count << endl;
        break;
    case ButtonPress:
        // ... handle button press event ...
        cout << event.xbutton.x << endl;
        break;
    ...
```

  - ↳ switch stmt binding in global event callback: used in early Windows & each app window registers WindowProc fcn that's called each time event is dispatched

- e.g.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg,
                            WPARAM wParam, LPARAM lParam) {
   switch (uMsg) {
      case WM_CLOSE:
         DestroyWindow (hWnd);
         break;
      case WM_SIZE:
         …
      case WM_KEYDOWN:
         …
```

this is basically what SimpleKit does with setSKEventHandler

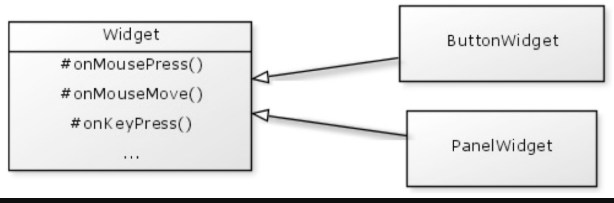
↳ problems w/ switch stmt binding:
- difficult to maintain
- events not delegated to obj so events are handled in loop itself
- better if widgets handled events themselves

↳ inheritance binding: event is dispatched to generic OO widget
- widget extends from base class w/all event handling methods
- base class chooses specificity of method
  - → e.g.

- general event types, e.g. onMouse, onKeyboard
- specific events, e.g. onMouseMove, onMouseClick

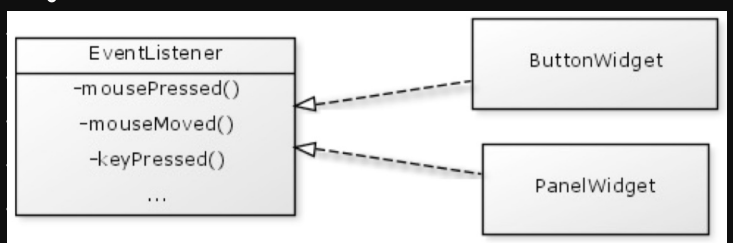
this is what SimpleKit does in SKElement

- Used in Java 1.0



↳ problems w/ inheritance binding:
- multiple event types still processed thru each event method
  - → switch stmt, but in widget
- no filtering of events can affect performance
  - → e.g. all mousemove events delivered
- if using specific event methods, doesn't scale well
  - → e.g. need to add penButtonPress to base class

↳ listener model binding: define interfaces for specific event/device types
- create obj that implements interface to handle certain events
  - → e.g. KeyListener, MouseListener
- when event is dispatched, relevant listener method called
- e.g. JavaFX



- SK event binding uses form of inheritance binding
  - ↳ widgets implement methods to handle toolkit events
    - e.g.

```
handleMouseEvent(me: SKMouseEvent) {

  switch (me.type) {
    case "mousedown":
      this.state = "down";
      return true;
      break;

    case "mouseup":
      this.state = "hover";
      ...
```

  - ↳ SKElement has methods to bind widget events (events gen by widget itself) to event handlers
    - defines generic event handler fcn type + maintains table of binding routes to event handlers

```
type EventHandler = (me: SKEvent) => void;


type BindingRoute = {
  type: string; // event type
  handler: EventHandler;
};

bindingTable: BindingRoute[] = [];
```

showing simplified version of code

    - has methods to add/remove event listeners

```
addEventListener(
  type: string,
  handler: EventHandler
) {
  this.dispatchTable.push({ type, handler });
}
```

to bind code w/ widget event:

- ↳ call addEventListener method of widget w/ name of event type (e.g. action) + fcn that widget should call when that event fires (i.e. event handler)
- ↳ this adds BindingRoute obj to bindingTable
  - obj has type of widget event, ref to event handler, + type of propagation (capture if T + bubble if F)
- ↳ when event is dispatched to widget that triggers that widget event, it'll call sendEvent w/ event type
- ↳ sendEvent loops thru bindingTable to find matching handler
- ↳ when providing handler, return T to stop propagation of event b/c it's been handled

EVENT PROPAGATION

- most UI toolkits support top-down + bottom-up propagation
  - ↳ top-down is capture
  - ↳ bottom-up is bubbling
- any widget in path can use event during pass
- handler can stop all following propagation

↳ e.g. propagation

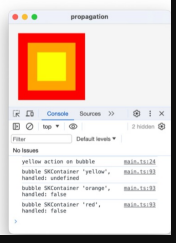
```
function dispatch(me: SKMouseEvent, root: SKElement) {
  let route = buildTargetRoute(me.x, me.y, root);

  // capture
  const stopPropagation = !route.every((element) => {
    return !element.handleMouseEventCapture(me);
  });

  if (stopPropagation) return;

  // bubble
  route.reverse().every((element) => {
    return !element.handleMouseEvent(me);
  });
}
```

capture binding

returns true to stop propagation

bubble binding

- Demo
  - capture flag for listeners
  - return true in handler to stop propagation



capture phase is useful b/c events higher up in widget tree can filter events
↳ e.g. parent can disable children

pure positional dispatch can lead to odd behaviour
↳ e.g. mouse drag starts in scrollbar but moves outside: send events to adjacent widget?

focus dispatch: events dispatched to widget regardless of mouse pos
↳ needed for all keyboard + some mouse events
↳ max 1 keyboard + 1 mouse focus
↳ needs positional dispatch
- mousedown event sets mouse + keyboard focus to widget
  → only text entry widgets should request keyboard focus

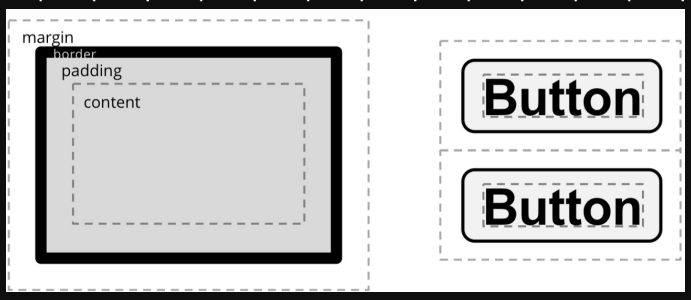
UI toolkits gen events when mouse enters + exits widget
↳ used for hover effects
↳ approach:
1) get frontmost widget in target route
2) if that elmt wasn't last elmt entered, set mouseexit event to last elmt entered + mouseenter event to this frontmost elmt

sometimes, apps can access system events
↳ aka global event queue hooks
↳ monitor events across all apps
↳ inject events to another app

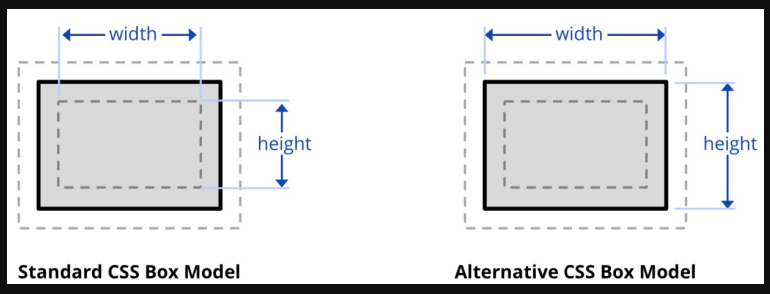
LAYOUT

BOX MODEL

- UI layout: visual arrangement of widgets in UI
- UI elmts use hierarchy of dimensions for size
  - ↳ e.g. CSS box model
    - margin: outside space away from other elmts
      - → not part of rendered elmt
    - border: thickness of stroke outlining elmt
    - padding: inside space btwn border + content
    - content: acc content of elmt



- 2 CSS box models:
  - ↳ std: width + height defined by content size
    - acc rendered size = content size + padding + border
  - ↳ alt: width + height define rendered elmt size
    - content size = acc rendered size - (padding + border)

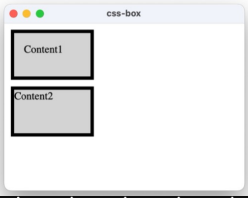


- e.g. css-box

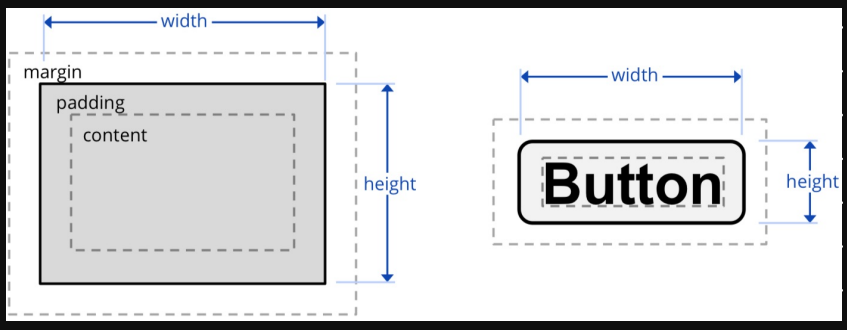
```
.box {
  box-sizing: border-box;
  width: 125px;
  height: 75px;
  margin: 10px;
  padding: 15px;
  border: 5px solid black;
  background: lightgrey;
}


<div class="box">Content1</div>
<div class="box">Content2</div>
```

Use alternate box model

Note CSS can set different margin, border, and padding for each side



- SK box model uses 3 dimensions + CSS alt box model (i.e. width + height define rendered elmt size)
  - ↳ dimensions are margin, padding, + content
  - ↳ top, right, bottom, left can't be set separately



- elmt may be rendered larger/smaller than its width / height
  - ↳ e.g. width / height < (2 * padding)
  - ↳ e.g. layout may expand / shrink elmt
- width + height don't have to be defined
  - ↳ let layout choose instead
- basis size: elmt's normal size w/ padding + margin (i.e. min dimensions)
  - ↳ if width + height are undefined, basis could be 0,0
  - ↳ e.g. width = 15 + padding = 20, then basis width = 40
  - ↳ e.g. if width undefined for Button widget, basis width is min width to fit button text
- layout size: elmt's size after layout w/ padding + margin
  - ↳ i.e. space elmt occupies in layout

LAYOUT STRATEGIES

- strategy design pattern: factor out layout algo into separate fcn + allow elmt to choose algo

```
export type LayoutMethod = (
  boundsWidth: number,
  boundsHeight: number,
  elements: SKElement[]
) => Size;
```
width and height type

- layout method property in SKContainer:

```
protected _layoutMethod: LayoutMethod | undefined;
set layoutMethod(lm: LayoutMethod) {
  this._layoutMethod = lm;
}
```

- factory pattern for making layout fcn
  - ↳ can also enable optional properties for layout algo
    - e.g. gap btwn elmts

```ts
export function makeWrapLayout(
  props: { gap: number } = { gap: 0 }
): LayoutMethod {
  return (
    boundsWidth: number,
    boundsHeight: number,
    elements: SKElement[]
  ) => {
    return wrapLayout(boundsWidth, boundsHeight, elements, props);
  };
}

// actual layout function
export function wrapLayout(
  boundsWidth: number,
  boundsHeight: number,
  elements: SKElement[],
  props: { gap: number }
): Size {
```

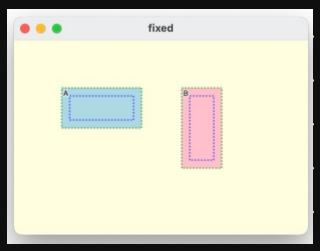
props specific to layout

e.g. **fixed**

↳ allows elmts to set x, y, width, height

↳ simplest layout

↳ warns if elmt is outside container bounds



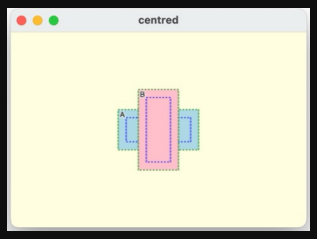
e.g. **centre**

↳ centres all elmts in centre of container

- stacked back to front

↳ sets new x + y pos for each elmt

↳ warns if elmt is outside container bounds



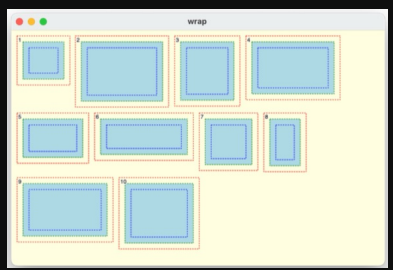
e.g **wrap**

↳ places elmts in rows, wrapping to next row as needed

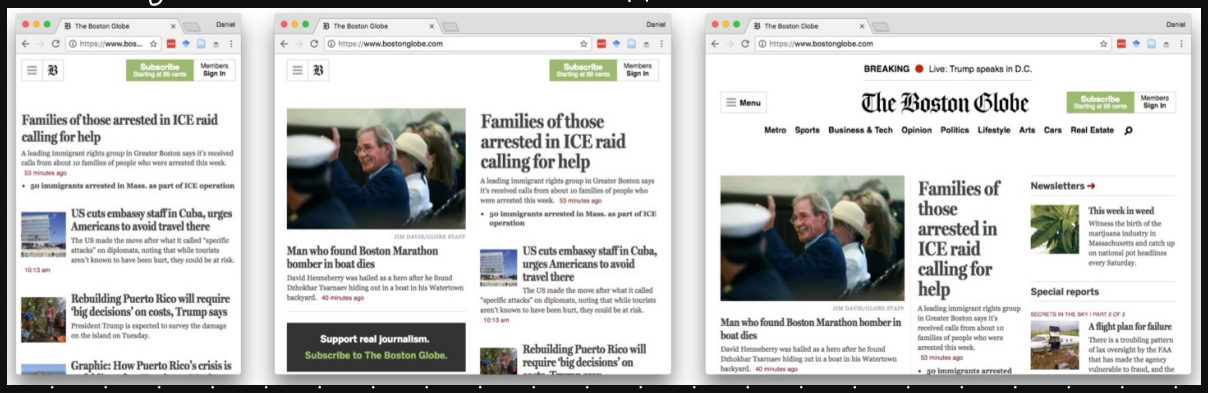
↳ takes gap param

↳ warns if elmt wider than bounds + if vertical overflow

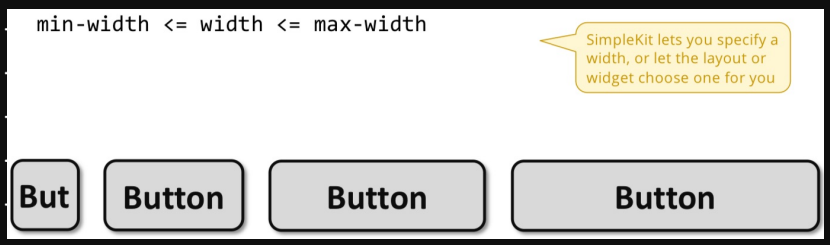
↳ tallest elmt determines row height



e.g. responsive
↳ dynamically reposition, resize, hide content in response to change in screen resolution + app window resize



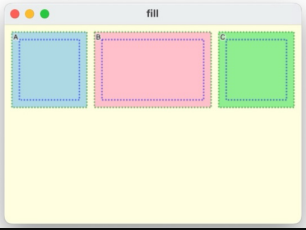
widgets must be flexible in size + pos
↳ store own pos + width/height, but layouts can change them
some UI toolkits support multiple basis size hints
↳ e.g. CSS attrs



var intrinsic layout uses widget preferred basis sizes, but queries all widgets first, then allocates space to them as group
↳ determined in 2 passes:
1). get each child widget's preferred size (basis)
2). decide on layout, then iterate thru each child + set its layout size + pos

e.g. fill
↳ lays out elmts in a row + they can grow/shrink to fit space

fillRow.ts in simplekit/layout
- element "basis" width to find available space
- element fillWidth to find proportion to fill
- fillWidth property on SKElement (sets proportional change)
- fillHeight option
- calculate bounds



every time widget size / layout-related property changes, at least some of UI must be laid out again (i.e layout invalidation)
↳ could be multiple changes each frame of run-loop

↳ best practice is to run layout at most once per frame
↳ e.g. imperative-mode.ts

- SimpleKit uses a global invalidate layout function to set a flag

```
function invalidateLayout() { layoutDirty = true; }
```

- If widget size or layout property changes, call `invalidateLayout`
  then layout is done just before drawing next run-loop frame

```
if (uiTreeRoot && layoutDirty) {
  layoutRoot();
  layoutDirty = false;
}
```

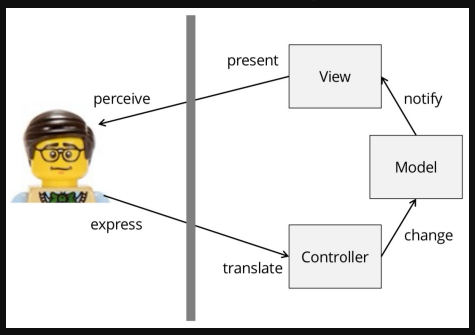
sets root to canvas size, and calls doLayout() which will call doLayout for all widgets in the widget tree

tips + strategies:
↳ break up UI recursively into regions
↳ consider how controls are laid out within each region
↳ nest layouts

MVC

BENEFITS

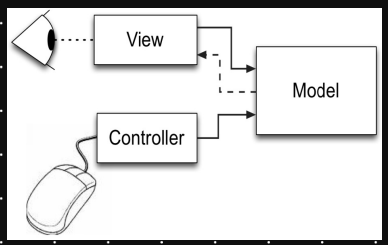
- MVC stands for Model-View-Controller
  - ↳ 1st MV* interactive system architecture
  - ↳



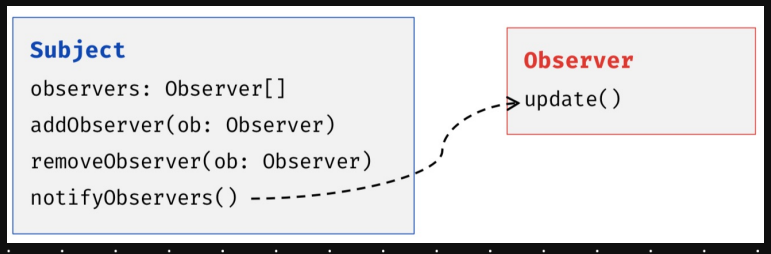
- we use MVC b/c:
  - ↳ separate data, state, & business logic from UI
    - V & C implementations can change w/o changing M implementation
  - ↳ support multiple views of same data
  - ↳ separation of concerns in code
    - code reuse
    - unit testing
- define single model that provides info to multiple views
  - ↳ multiple views should sync (i.e. when 1 changes, so do others)

IMPLEMENTATION

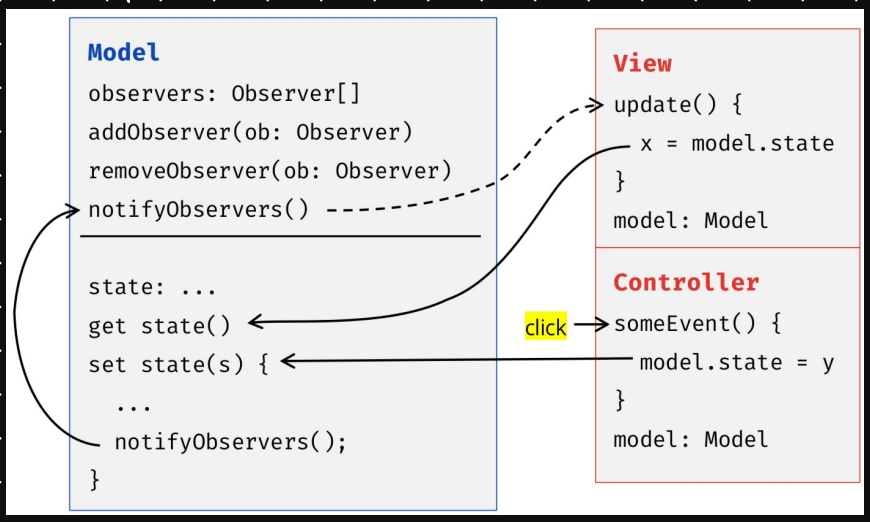
- to implement MVC, decompose interface architecture into 3 parts
  - ↳ model: manages app data & modifications
  - ↳ view: manages interface to present data
  - ↳ controller: manages interaction to modify data
  - ↳



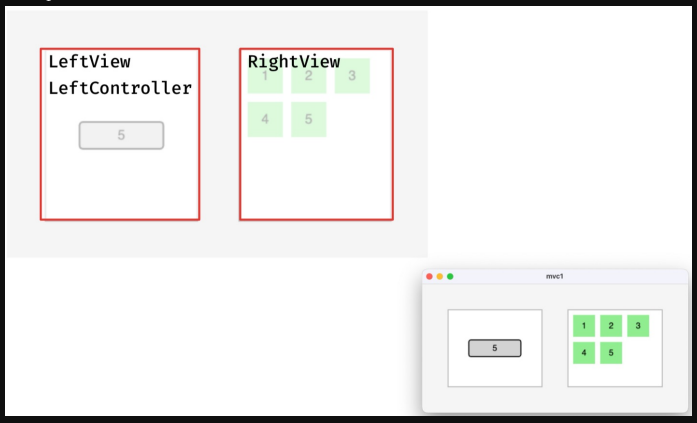
- observer pattern:



↳ MVC implementation:



↳ e.g. mvc1



- Observer interface + Subject base class:

```ts
export interface Observer {
  update(): void;
}

export class Subject {
  private observers: Observer[] = [];

  addObserver(observer: Observer) {
    this.observers.push(observer);
    observer.update();
  }

  ...

  protected notifyObservers() {
    this.observers.forEach((o) => o.update());
  }
}
```

single generic update notification

first view update

call this every time state changes

- View for leftController:

```
export class LeftView extends SKContainer implements Observer {

  update(): void {
    this.button.text = `${this.model.count}`;
  }

  button: SKButton = new SKButton({ text: "?" });

  constructor(private model: Model, controller: LeftController) {
    super();

    this.addChild(this.button);

    // set an event handler for button "action" event
    this.button.addEventListener("action", () => {
      controller.handleButtonPress();
    });

    // register with the model when we're ready
    this.model.addObserver(this);
  }
}
```

when model changes

references to model and controller

connect to controller

- Controller:

```
export class LeftController {

  constructor(private model: Model) {}

  handleButtonPress() {
    this.model.increment();
  }
}
```

- Model

```
export class Model extends Subject {

  // model data (i.e. model state)
  private _count = 0;
  get count() {
    return this._count;
  }

  // model "business logic"
  increment() {
    this._count++;
    // need to notify observers anytime the model changes
    this.notifyObservers();
  }
}
```

called whenever state changes

- in theory, V + C are loosely coupled
- in practice, V + C are tightly coupled
- adaption to pure MVC is to integrate C w/ V
  - ↳ most typical MVC approach in practice
  - ↳ users view data + provide input in same places
  - ↳ some view-lvl data manipulations don't make sense to handle thru M
    - e.g. validating input
  - ↳ e.g. mvc2 where View has integrated Controller

```
export class LeftView extends SKContainer implements Observer {

  update(): void {
    this.button.text = `${this.model.count}`;
  }

  button: SKButton = new SKButton({ text: "?" });

  constructor(private model: Model) {
    super();

    this.addChild(this.button);

    //  Controller
    this.button.addEventListener("action", () => {
       model.increment();
    });

    // register with the model when we're ready
    this.model.addObserver(this);
  }
}
```

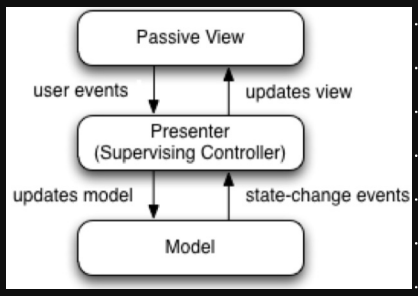
this is the controller

- for each view update, everything in every view is refreshed from model
  - ↳ could add params to update to indicate if it can ignore change.
  - ↳ simpler is often better so don't worry abt efficiency for now
- in Model, usually have CRUD (create, read, update, delete) methods
  - ↳ CUD must notify observers

MVC VARIANTS

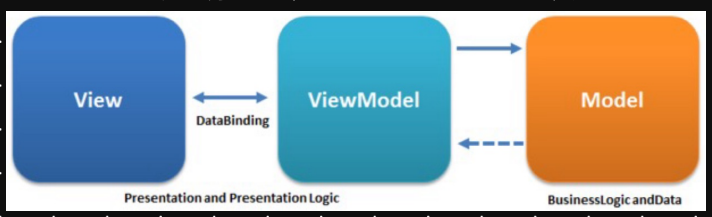
Model - View - Presenter: P is middle-man btwn M + V

- ↳ P manages business logic for Vs
  - i.e. determines how data should be handled
- ↳ often used as broker w/ multiple views to determine which view should respond to data (assuming only 1 is active)
- ↳



Model - View - ViewModel: VM mediates btwn M + V

- ↳ VM manages View's display logic + display-indep logic is delegated to M.
- ↳ useful when we have view-dependent state
  - e.g. localized UI that uses time + date format specific to region
    - → universal data in M.
    - → data format conversion in VM.
- ↳



~~HTML~~ ~~CSS~~

HTML AND CSS

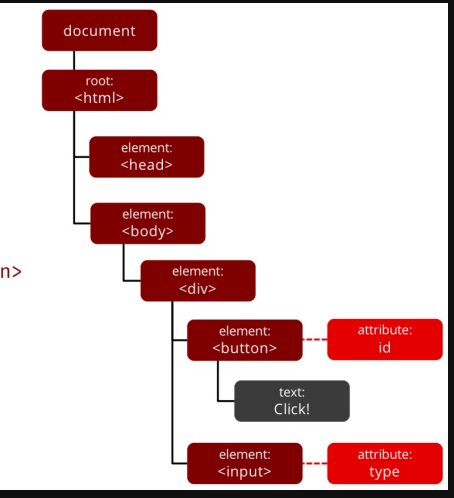
- HTML stands for Hyper Text Markup Lang
  - ↳ defines meaning & struct of interface
  - ↳ declarative syntax, as opposed to imperative in SK
- tag is syntax that defines elmt & its attrs
  - ↳ attr is extra info to define elmts
  - ↳ elmt is what tag & attrs create
    - e.g.

- `<input type="text" />` creates a textfield element
- `<div>inner content</div>` creates a container element
- `<button id="b" >Click!</button>` creates button element

The **id attribute** is a unique name for an element in the current page

- Document Obj Model (DOM) is cross-platform & lang-indep interface that treats HTML doc as tree struct of node objs
  - ↳ every elmt in DOM is a node
    - web page is doc node
    - elmt node
    - attr node
    - text node
    - comment node
  - ↳ events are also part of DOM
  - ↳ e.g.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  </head>
  <body>
    <div>
      <button id="b">Click!</button>
      <input type="text" />
    </div>
  </body>
</html>
```



- & are generic HTML container widgets
  - ↳ div tag is block-lvl elmt used for associating & grouping tgr nested elmts
    - like SKContainer

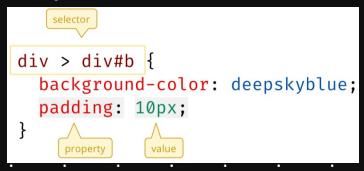
↳ span tag is inline elmt used for associating + grouping tgt nested elmts
- often used for styling

e.g. HTML widgets

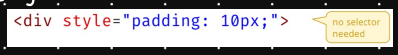
- <button name="...">Hello</button>
- <input type="..." >
  - "range" for a slider (numeric logical device)
  - "checkbox" for a checkbox (boolean logical device)
  - "radio" for a radio button (boolean logical device)
  - "color" for a colour picker!
  - ~~"button" for a button~~
  - ...
- <label> for label associated with a widget
- <textarea> for editable multiline text
- <select> or <datalist> with <options> for a menu

CSS stands for Cascading Style Sheets
↳ used for visual style, layout, + animation
↳ declarative syntax

CSS rule has 3 parts:
↳ selector
↳ declaration block
↳ lr props w/ vals
↳ e.g.



to specify CSS:
↳ can do directly in HTML tag using elmt style attr
- e.g.



↳ inline in doc in <style> elmt
- e.g

```
<style>
  div { padding: 10px; }
</style>
```

↳ link to file
- e.g.

```
<link rel="stylesheet" href="style.css" />
```

in "style.css" file:

```
div {
  padding: 10px;
}
```

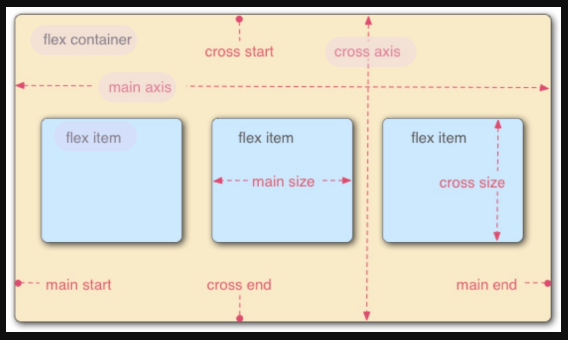
- CSS selector is a pattern to select / find elmts in DOM
  - ↳ basic:
    - div {...} selects by tag type
    - .foo {...} selects by class name
    - #a {...} selects by elmt id attr
    - [type = "text"] {...} selects elmts matching attr val
  - ↳ hierarchal:
    - div > div {...} selects child elmts by parent-child rltnship
    - div div {...} selects child elmts by descendant rltnship
  - ↳ pseudo class:
    - :first-child selects 1st child
    - :hover selects when mouse is over elmt
  - ↳ combining:
    - div#a {...} selects elmts matching all selectors
    - div, #a {...} selects elmts matching at least 1 selector
- CSS cascade defines precendence of CSS rules when multiple declarations can apply to same elmt
  - ↳ who: agent (browser) → author → user
  - ↳ where: style.css or <style> → inline style attr
  - ↳ when: order of rules in inline css or files
  - ↳ use !important for importance
  - ↳ specificity of rule: std method where most specific CSS selector sets style
    - e.g.:

```
div#a { background-color: blue; }
div { background-color: red; }
...
<div id="a">A</div>
```

div will be blue

FLEXBOX LAYOUT

- CSS flexbox is used to design complex layouts more efficiently



- to use flexbox layout:
  - ↳ make parent a flex container
    - i.e. set CSS display prop to "flex"
  - ↳ children become flex items
- CSS props for flex items:

↳ flex-grow : proportion to grow elmt to fill space
↳ flex-shrink : proportion to shrink elmt to fill space
↳ flex-basis
- auto means use width / height
- # means use that as basis
- else, use content size

↳ flex shorthand: grow shrink basis
- e.g.

```
flex: 1 2 auto /* grow 1, shrink 2, basis auto */
flex: 1 /* grow 1, shrink 1, basis auto *
```

default:

```
flex: 0 1 auto  / * same as flex: initial */
```

CSS props for **parent control** over flex items:
↳ align-items : how items align along cross axis

```
align-items: < stretch, flex-start, flex-end, center >
```

↳ align-self: how it aligns along cross axis

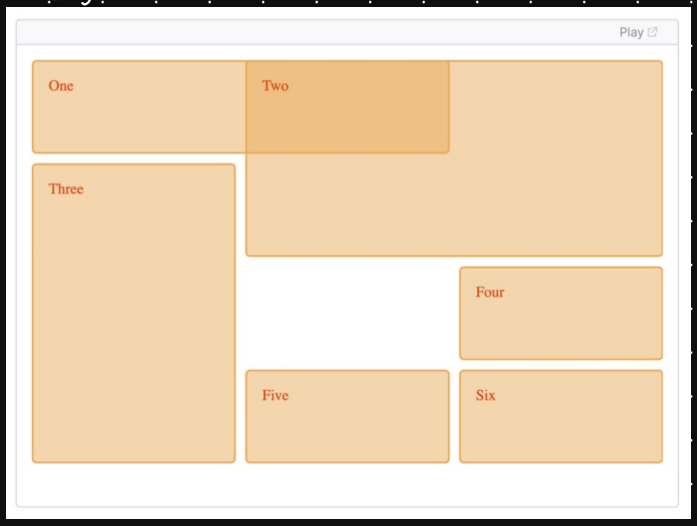
```
align-self: < stretch, flex-start, flex-end, center >
```

↳ justify-content: how items align along main axis

```
justify-content: < flex-start, flex-end, center,
                   space-between, space-around, space-evenly >
```

↳ gap

CSS grid layout module divides container into major regions, defining child rltnships in terms of size, pos, + layer
↳ e.g.



DOM MANIPULATION WITH TS

general dev steps:
1) mockup HTML as static pg
2) add CSS to create visual style, layout, etc.
3) divide HTML up for manipulation approach

4) create views for main parts of interface
- divide CSS up into views
- build view from code/strs
- create controllers using event listeners
- attach everything to root elmt for view
- implement Observer update method

to get refs to DOM elmts
↳ use unique elmt id

```
document.getElementbyID("my-id")
```

↳ use CSS selector syntax (more flexible)

```
querySelector("#my-id")
```

↳ best practice is to specify elmt type expected
↳ must handle case if ref not found
- e.g.

```
const el = document.querySelector("#inc") as HTMLButtonElement;
if (!el) throw new Error("leftView inc button not found");
this.button = el; // save reference in view
```

DOM events dispatch essentially same as SK
↳ capture + bubble phases
↳ event.stopPropagation() method
↳ e.g. setting event handlers

```
button.addEventListener("click", (e) => { ... });
```

2 Vanilla HTML manipulation approaches are to build HTML in imperative steps or from str
↳ other approaches are using HTML templates + web components

e.g. imperative

- Build DOM for views step-by-step in code:

```
document.createElement( ... )
container.appendChild( ... )
container.replaceChildren( ... )
```

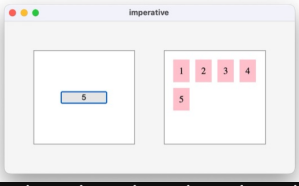
- Demos
  - App has div#app root
  - All views have a "root" element (see View interface) to enable this:

```
panel.appendChild(new LeftView(model).root);
```

  - Separate css files with selectors that will apply only to view
  - Style files can be imported in code, e.g.

```
import "./leftView.css";
```

  - Button controller



using DocumentFragment leads to more efficient performance when needing to

add a lot of elmts

↳ e.g.

```
const fragment = document.createDocumentFragment();

[ ...Array( ... )].forEach((i) => {
  const div = document.createElement("div");
  ...
  fragment.appendChild(div);
});

this.container.appendChild(fragment);
```

tag transforms template literal before creating str

↳ e.g.:

```
const result = myTag`<div class="msg">${message}</div>`;
```

(tag; string[0]; value[0]; string[1])

↳ fcn that's called w/args from template literal

- 1st arg is arr of N+1 strs
- other args are N strs created from template vals
- e.g.

```
function myTag(
  strings: TemplateStringsArray,
  ...values: string[]
) {
  return strings.reduce(
    (acc, str, i) => acc + str + (values[i] || ""),
    ""
  );
}
```

A real tag function would transform the strings and value

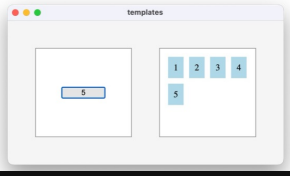
↳ best practice to pass HTML templates thru html tag

e.g. templates

- Build DOM for views step-by-step in code:
  - Set `innerHTML` to the result of a HTML tagged template literal
- **Creating** View root uses template element

```
var temp = document.createElement("template");
temp.innerHTML = html`
    <div id="left"><button id="increment">?</button></div>
  `;
this.container = temp.content.firstElementChild as HTMLDivElement;
```

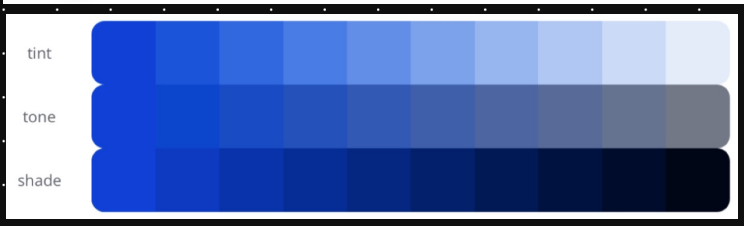
- Demos
  - extra code to get ref to button



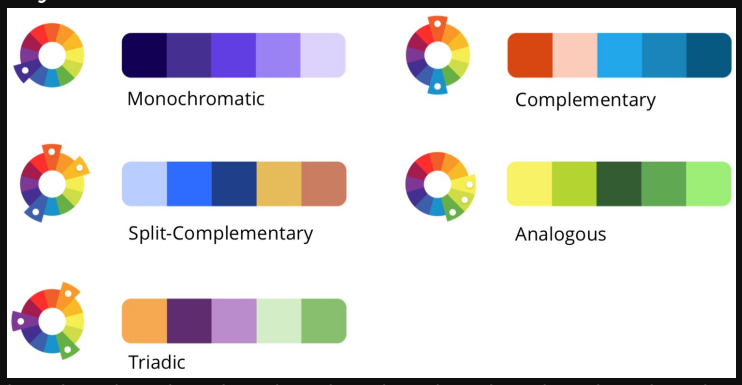
VISUAL DESIGN

COLOUR

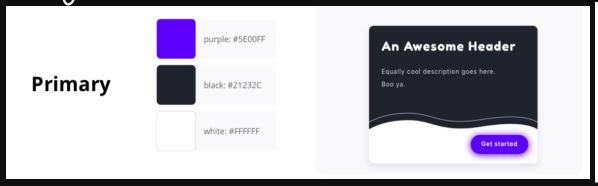
- **hue**: pure, dominant colour in colour wheel
- **saturation**: intensity + vividness of colour
- **tint**: amt of white added to colour
- **tone**: amt of black + white added to colour
  - ↳ mutes colour
- **shade**: amt of black added to colour
- e.g.

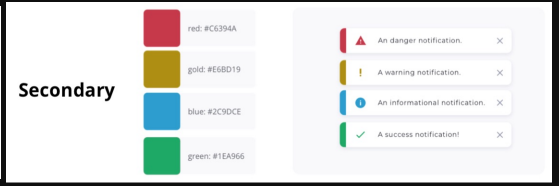


e.g. harmonious palettes



- colour design system defines how colours are used
  - ↳ **primary colours** are used frequently in designs
    - includes tints, shades, + tones of primary brand colours
    - for branding, interaction elmts, layouts, + text
  - ↳ **secondary colours** are used occasionally
    - accents based on primary brand colours
    - for fcns (e.g. task success, error, selected elmt)
  - ↳ e.g.





e.g. colour

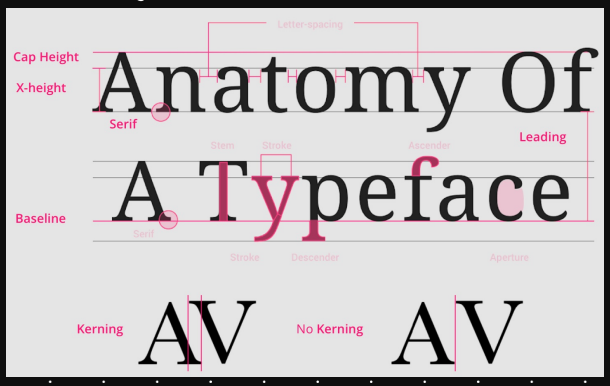
```css
/*
CSS variables for colours
Note different ways of defining colours
*/

:root {
  --primary1: #5e00ff;
  --primary2: rgb(33 35 44);
  --primary3: white;
  --primary4: lightgrey;

  --secondary1: hsl(353 55% 50%);
  --secondary2: #e6bd19;
}

h1 {
  color: var(--primary1);
}

button {
  background-color: var(--primary4);
  color: var(--primary2);
  min-width: 80px;
}

button.primary {
  background-color: var(--primary1);
  color: var(--primary3);
}

/* danger div */

div.danger {
  margin: 10px;
  padding: 10px;
  border-radius: 10px;
  border: 1px solid var(--secondary1);
  border-left: 10px solid var(--secondary1);
  color: var(--secondary1);
}

div.danger > p {
  margin: 0;
}

div.danger > button {
  margin-top: 10px;
  background-color: var(--secondary1);
  color: var(--primary3);
}

```

↳ use :root pseudoclass to define `css vars` for colours

TYPOGRAPHY

`typography` is art of arranging letters & text in a way that makes it legible, clear, & visually pleasing

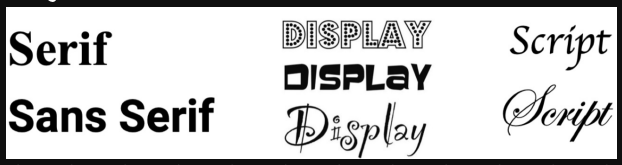
terminology:



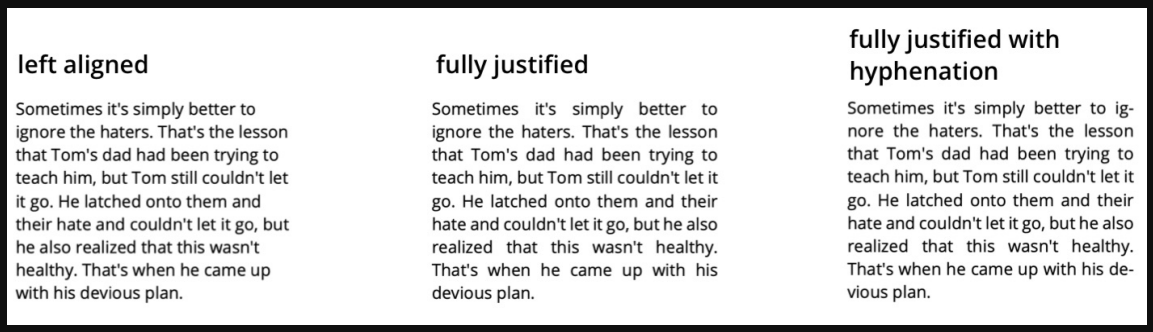
`typeface`: overall design of a set of chars
↳ e.g. Times New Roman, Helvetica, Arial

`font`: specific style & size of typeface
↳ particular weight (e.g. bold), emphasis (e.g. italic), & pt (0.351mm = $\frac{1}{72}$ inches)

- 4 main typeface styles: serif, sans serif, display, & script
  - ↳ e.g.



  - ↳ prefer std & simple
    - sans serif is most readable
    - serif can be effective for headings & feels classic
- define hierarchical names in typographic system
  - ↳ e.g. footer, body, caption, subheading, heading, title, & hero
  - ↳ use size, weight, & emphasis to create visual hierarchy
    - 7 diff scales are usually sufficient
  - ↳ sparingly use all caps for headings
- leading: space btwn lines
  - ↳ inc leading to emphasize lines
  - ↳ add space btwn paragraphs instead of indentation of 1st line
- alignment guidelines:
  - ↳ left for paragraphs
  - ↳ left / centre for headings
  - ↳ centre for buttons + interface controls
  - ↳ nvr use fully justified
  - ↳ e.g.



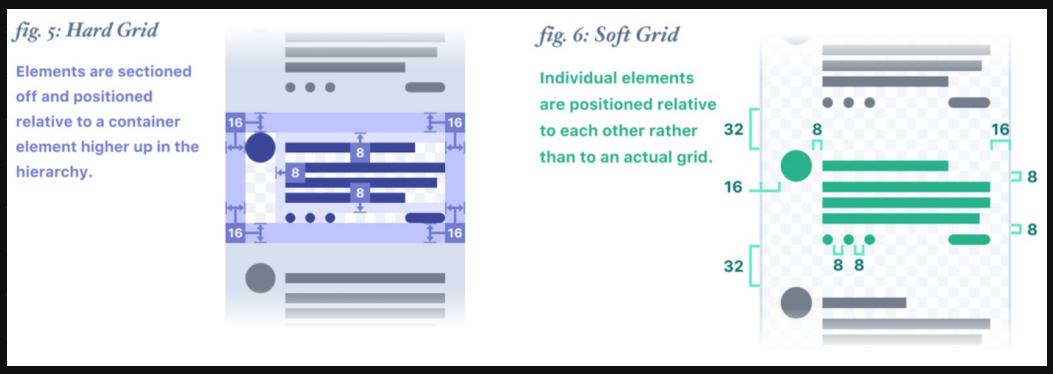
- typography rules of thumb:
  - ↳ avoid display typefaces (esp comic sans)
  - ↳ avoid 2+ typefaces in design
  - ↳ avoid 3+ weights (avoid black + ultralight)
  - ↳ avoid underlining unless hyperlink
  - ↳ avoid fully justified text

LAYOUT

- layout: struct that organizes visual elmts in interface
  - ↳ creates visual paths, connections, & gaps to group, rank, & make sense of info
- grid: underlying regularized struct to organize elmts in layout

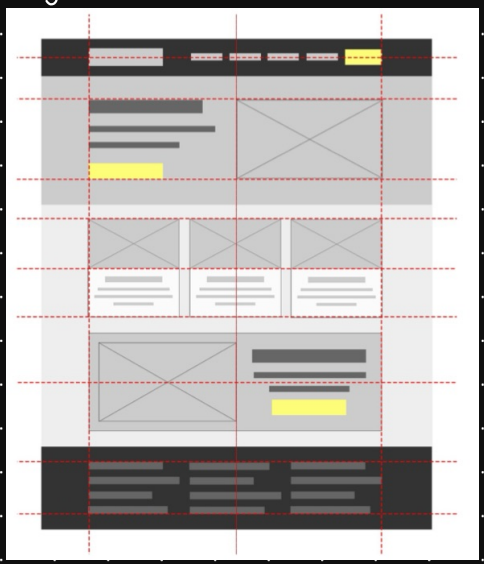
↳ most basic form is columns + gaps (aka gutters)
  - often 12 cols
↳ often has baseline grid to place texr

8-pt pixel grid: use multiples of 8 to define dimensions, paddings, + margins
↳ cols, gaps, elmt sizes, + spacing are all multiples of 8
↳ type can be multiples of 8 (although usually 4)
↳ e.g.



negative space: white space btwn content
↳ groups + divides content
↳ lets user's eyes rest
↳ more -ve space = more emphasized content

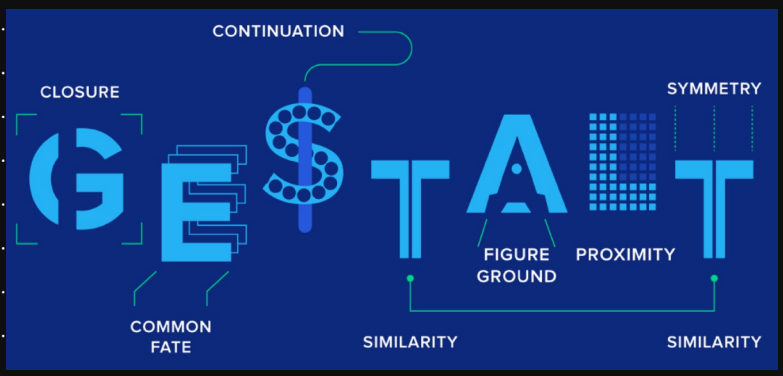
alignment: dominant visual parts of multiple elmts form implied line
↳ use grid to force good alignment
↳ e.g.



aspect ratio: measure of img, illustration, or type's width rltive to height
↳ changing it changes balance + perception
↳ don't change ratio to fit layout

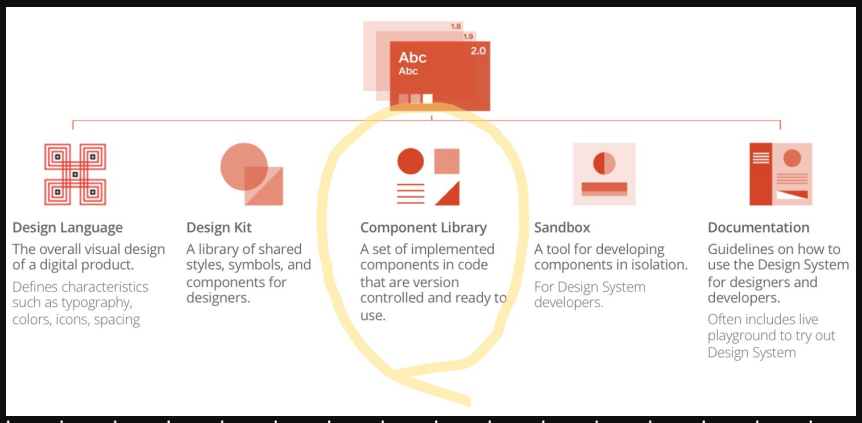
layout rules of thumb:
↳ use underlying grid to guide design

- can use flexbox
  - ↳ don't try to use all of grid
  - ↳ can break grid for added emphasis
  - ↳ use -ve space
  - ↳ good layout uses GESTALT principles:



DESIGN SYSTEMS

- design system: systematic approach to product dev
  - ↳



- Boolstrap is free, open-src CSS framework for responsive, mobile-first, FE web dev
  - ↳ design templates w/ typography + forms
  - ↳ rdy to use UI components
    - e.g. buttons, navigation
- Material Design is Google's open-src design system
  - ↳ UI components
  - ↳ UX guidance
    - e.g. colour, icons, motions, accessibility, adaptive design

TEXT

CHARACTER SETS

- text: series of chars
- sets of chars form writing system in human lang
- need stdardizing encodings for chars in bin
- American Std Code for Info Interchange (ASCII) is char encoding std that uses #s to rep text chars
  - ↳

ASCII control characters

00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

- Unicode is superset of ASCII
  - ↳ capacity up to 1114 112 chars
  - ↳ every char in every lang has unique encoding
  - ↳ general struct:
    - #0 - #127: same as ASCII
    - #128 - #256: common signs + accented chars
    - after #256: more accented chars
    - after #800: Greek alphabet, then Cyrillic, etc.
  - ↳ char codes written as U+ then hexadecimal
    - e.g. H is U+0048
  - ↳ Unicode encoding needs >1B
    - implementation isn't defined by Unicode
- Universal Char Set Transformation Format 8 bit (UTF-8) is char encoding that uses multi-byte var width encoding
  - ↳ internally, web browsers use 4B wide chars
  - ↳ problem w/sending, receiving, + storing text
    - some software uses 1B units
    - 4B for each Latin char bloats storage

↳ e.g.

Code point ↔ UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
0 U+0000	127 U+007F	0xxxxxxx			
128 U+0080	2047 U+07FF	110xxxxx	10xxxxxx		
2048 U+0800	65535 U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
65536 U+10000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

how many bytes to expect (e.g. 4) — identifies this as a "continuation" byte

INTERNATIONALIZATION

internationalization (i18n): designing + developing software so it can be adapted to diff cultures + langs

↳ i18n features include Unicode chars + bidirectional text

↳ support locale formats for #s, currency, date, time, etc.

↳ plan for regional diffs in storing info

↳ separate localization elmts from source code + content

localization (l10n): act of implementing i18n

↳ locale: region + lang

- e.g. en_CA, fr_CA

↳ to implement in browser:

1) use HTML data attr to identify i18n text elmts

```
<label data-i18n="label-name" ...
```

2) create JSON translation data structs

```
en: { label-name: "Name" ...
fr: { label-name: "Nom" ...
```

3) use preferred browser locale

```
navigator.language
```

4) recommended: add lang switcher

VALIDATION

interfaces often need to validate text input from user

↳ i.e. ensure that user input meets specified criteria before it's processed/accepted

↳ e.g. required field, certain format, within certain range, + unique

we need form validation b/c:

↳ system needs right data in right format or logic won't work

↳ to guide users

- e.g. making secure passwords

↳ protect system from attacks mounted thru unprotected text submission

- in practice, input sanitization also done on server
- e.g.

Username bird2002

```
sql = `SELECT * FROM users WHERE name = '${userName}';`
```

If a real username is entered everything is fine

.... but a malicious user could enter:

Username | a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't

```
sql = `SELECT * FROM users WHERE name = '${userName}';`

SELECT * FROM users WHERE name = 'a';DROP TABLE users; SELECT *
FROM userinfo WHERE 't' = 't';
```

- guidelines for form validation:
  - ↳ place error msgs near fields
  - ↳ use colour to differentiate errors from normal field states
  - ↳ accept data formatted in diff ways when possible
  - ↳ filter invalid chars from being entered when possible
  - ↳ validate field before input is complete when possible
- HTML form basics:
  - ↳ <form> to group diff input widgets tgt
  - ↳ <label> to associate text w/ input field using elmt id
    - e.g.

```
<label for="name">Name</label>
<input type="text" id="name" />
```

Name

  - ↳ <input> placeholder text
    - e.g.

```
<input type="text" placeholder="Name" id="name" />
```

Name

- HTML5 widgets have built-in validation
  - ↳ use specific type of input

```
type="number"
type="email"
```

  - ↳ use attrs to configure validation

```
required
minlength, maxlength
min, max
pattern
```

  - ↳ use CSS pseudoclass selectors to provide validation feedback

```
:required
:invalid
```

- regular expr (aka regex/re): seq of chars that specifies a search pattern in text
  - ↳ describes deterministic finite automaton (DFA)
  - ↳ used in form validation to test if str is in correct format
    - e.g.

Postal Code (upper case only with optional space in between)
`[A-Z]\d[A-Z]\s*\d[A-Z]\d`

Number (decimals allowed, positive only, optional leading 0)
`\d*\.?\d*`

Phone Number (10 digit North American with formatting options)
`\(?\d{3}\)?[\s.\-]?\d{3}[\s.\-]?\d{4}`

- can do custom validation w/ **Constraint Validation API**
  - ↳ only avail on some widgets (e.g. button, input, select)
  - ↳ use **novalidate** attr in form elmts to turn off std validation msgs
  - ↳ API props + methods:

```
validity: ValidityState
checkValidity(): boolean
```

- must write custom validator for custom input widgets
  - ↳ create classes for invalid
  - ↳ listen to input event
  - ↳ test against conds (usually regex)
- **input formatting**: updates str in textfield as user types
  - ↳ input event listener rewrites textfield w/ std formatting
  - ↳ e.g.

User types: 519 5555	*Textfield displays:*	+1 (519) 555-5

- **input masking**: provides graphical rep of final format + fills it in as user types
  - ↳ input event listener rewrites textfield w/ std formatting + placeholders
  - ↳ e.g.

User types: 5195555	*Textfield displays:*	+1 (519) 555-5___

UNDO

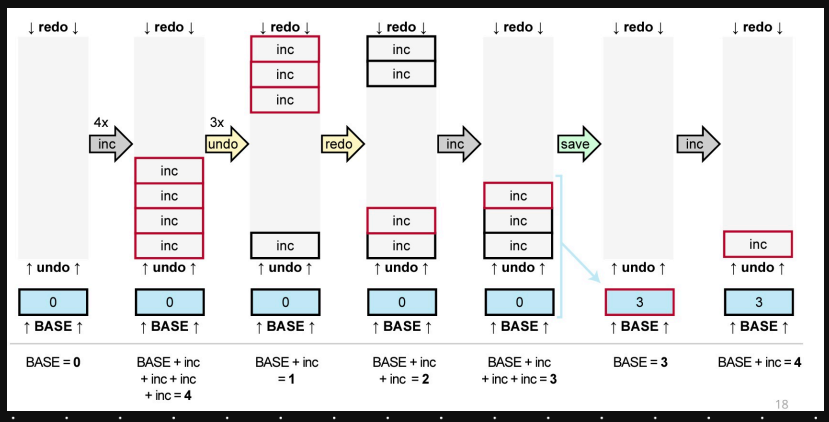
NOTE
undo means undo/redo
unless explicitly stated otherwise

- **benefits** of undo:
  - ↳ recover from errors
  - ↳ enables exploratory learning
    - e.g. trying things w/o knowing consequences + alt solns w/o fear / commitment
  - ↳ eval modifications
- **checkpting**: save curr state so can rollback later
  - ↳ manual undo
- undo design choices:
  - ↳ **undoable actions**
    - all changes to doc (i.e. Model) should be undoable
    - changes to View / doc's interface state should be undoable only if extremely tedious
    - ask for confirmation before doing destructive action that can't be easily undone
  - ↳ **state restoration**
    - UI state after undo/redo
      - → e.g. highlight text, delete, scroll, undo → scroll back to text
    - UI state should be meaningful after redo
      - → restore prior selection of objs
      - → scroll to restored objs
      - → give focus to control where state was restored
  - ↳ **granularity**: extent of which smth should be undone at 1 time
    - **chunk**: conceptual change from 1 state to another
      - → interaction can be divided into undoable chunks
      - → undo reverses 1 chunk
    - for drawing interactions:
      - → drawing line means:
        - mousedown to start line
        - mousemove to define line path
        - mouseup to end line
      - → undo should undo entire line, not just every mouse pos change for each mousemove
        - 1 chunk = mousedown + movemove + mouseup
    - for text interactions, differs btwn editors
      - → chunk can be each char, syntactical unit, based on temporal seq (i.e. edits separated by pause), etc
    - ignore direct manipulation for intermediate states
      - → e.g. ignore mousemove during resize / move states

- delimit chunks on discrete input breaks
  - → e.g. words in text
- chunk all changes from single interface event
  - → e.g. 1 chunk = find + replace all

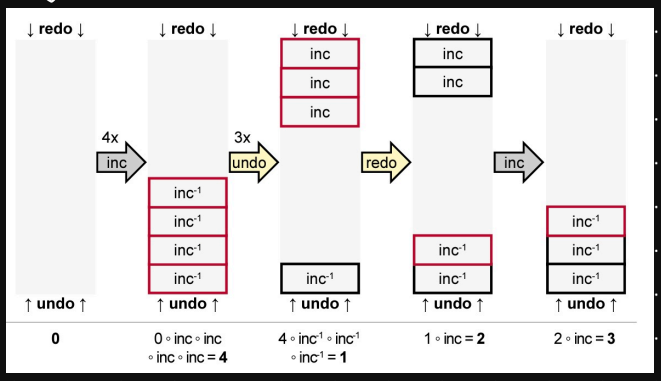
IMPLEMENTATION

- 2 general approaches:
  - ↳ **fwd undo**:
    - start from base doc + maintain list of changes to compute curr doc
    - undo by removing last change from list when computing curr doc
  - ↳ **reverse undo** (more commonly used):
    - apply change to update doc but also save reverse change as separate update
    - undo by applying reverse change to doc
- **change record**: defines single transformation to doc
  - ↳ i.e. state of Model
- both options require 2 stacks:
  - ↳ **undo stack**: all change records, saved as actions are performed
  - ↳ **redo stack**: change records that have been undone
    - reapply them w/ redo
- to implement **fwd undo**:
  - ↳ save **baseline** doc state at some past pt
    - e.g. S^v
  - ↳ save change records to transform baseline doc into curr doc state
    - e.g. $S = (c(b(a(S^v))))$
  - ↳ to undo last action, don't apply last change record
    - e.g. $S' = \text{undo}(c(b(a(S^v)))) = (b(a(S^v)))$
  - ↳ e.g. inc Model



- computes curr state from base val + "do" cmds on undo stack
- undo moves cmd from undo stack to redo stack + forces recompute of state using remaining undo stack
- save resets base val + clears undo + redo stacks
- ↳ **Command interface** has single "do" cmd

- to implement rev undo:
  - ↳ save complete curr doc state
    - e.g. S
  - ↳ save rev change records to return to prev state
    - e.g. {c^{-1}, b^{-1}, a^{-1}}
  - ↳ to undo last action, apply last rev change record
    - e.g. $S' = undo(S) = c^{-1}(S)$
  - ↳ e.g. inc Model



    - computes curr state from prev state combined w/ "do" cmd
    - undo computes new state using rev cmd in undo stack
  - ↳ Command interface has do + undo cmds:
    - e.g.

```
export interface Command {
  do(): void;
  undo(): void;
}


Example Command for increment
{
  do: () => {
    this._count++;
  },
  undo: () => {
    this._count--;
  },
} as Command
```

- rev undo cmd pattern:
  - ↳ user issues cmd:
    - execute cmd to create new curr doc state
    - push rev cmd onto undo stack
    - clear redo state
  - ↳ undo:
    - pop rev cmd from undo stack + execute it to create new doc state
    - push cmd onto redo stack
  - ↳ redo:
    - pop cmd off redo stack + execute it to create new doc state
    - push rev cmd on undo stack
- e.g. text editor cmds + rev cmds (undo/redo cmds)

- Available Commands:

```
insert(string, start)
delete(start, end)
bold(start, end)
normal(start, end)
```

execute	<cmd>	Quick brown	`insert("Quick brown", 0)`
execute	<cmd>	Quick **brown**	`bold(6, 10)`
execute	<cmd>	Quick **brown** fox	`insert(" fox", 11)`
undo	<reverse cmd>	Quick **brown**	`delete(11, 14)`
undo	<reverse cmd>	Quick brown	`normal(6, 10)`
redo	<cmd>	Quick **brown**	`bold(6, 10)`
execute	<cmd>	Quick **brown** dog	`insert(" dog", 11)`

Command	Document	Undo Stack	Redo Stack
insert("Quick brown", 0)	Quick brown	delete(0, 10)	<empty>
bold(6, 10)	Quick **brown**	*normal(6, 10)* delete(0, 10)	<empty>
insert(" fox", 11)	Quick **brown** fox	*delete(11, 14)* normal(6, 10) delete(0, 10)	<empty>
undo	Quick **brown**	normal(6, 10) delete(0, 10)	*insert(" fox", 11)*
undo	Quick brown	delete(0, 10)	*bold(6, 10)* insert(" fox", 11)
redo	Quick **brown**	*normal(6, 10)* delete(0, 10)	insert(" fox, 11)
insert(" dog", 11)	Quick **brown** dog	*delete(11, 4)* normal(6, 10) delete(0, 10)	<empty>

w/ MVC, only Model changes needed to support undo/redo

rev change record implementation options:

- ↳ cmd pattern: save cmd + rev cmd to change state
  - lightweight implementation
- ↳ memento pattern: save snapshots of each doc state
  - snapshots can be complete state or diff from last state
  - more heavyweight implementation
  - e.g. memento uses TS generics (i.e. define fcns, classes, or interfaces w/ placeholders for types)

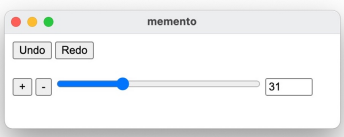
- UndoManager and Memento work with any type of Model state

```
interface Memento<State> { state: State; }
```

generic type

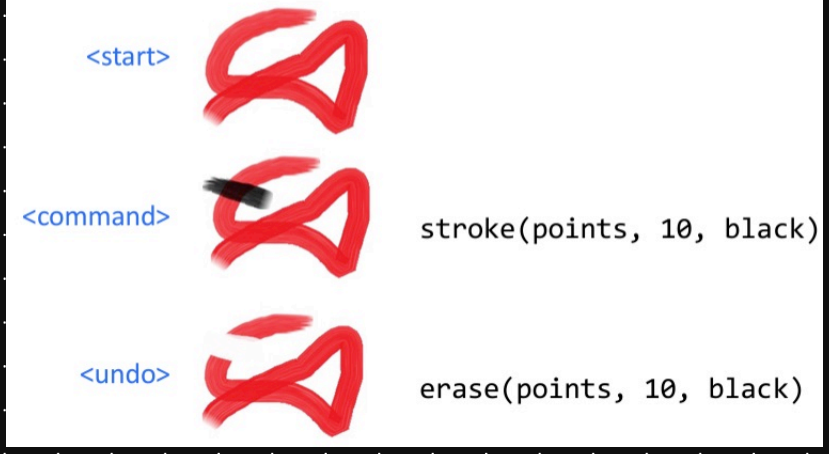
- Demo uses:

```
Memento<number>
```



e.g. bitmap paint app w/ rev cmd undo problem

stroke(points, thickness, colour)
erase(points, thickness)



solns for destructive cmds
- ↳ use fwd cmd undo
- ↳ use rev cmd undo but undo cmd stores prev state for destructive cmds (i.e. memento)
  - might require a lot of mem which is why some apps limit undo stack

undo app scope
- ↳ treat all windows as single model w/ single set of actions that can be undone
  - change window focus when undo/redo triggered
- ↳ have multiple undo + redo stacks, 1 for each window + use whichever has focus
  - more common

ASYNCHRONOUS

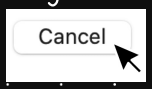
UI RESPONSIVENESS

- **responsive UI** delivers feedback to user in timely manner
  - ↳ doesn't make user wait longer than necessary
  - ↳ communicates state of long tasks
- make responsive UI in 2 ways:
  - ↳ design to meet human expectations + perceptions
  - ↳ load data efficiently so it's avail quickly
- **factors** that affect responsiveness:
  - ↳ user expectations
    - how quickly system should react to complete task
  - ↳ app + interface design
    - interface keeps up w/ user actions, shows app status, + doesn't make user wait unexpectedly
- responsiveness is most important factor in determining user satisfaction, more than ease of learning/use
  - ↳ not just system performance
- **slow performance, but responsive**:
  - ↳ provide feedback to confirm user actions
  - ↳ provide feedback abt what's happening (i.e. curr status)
  - ↳ allow users to perform other tasks while waiting
  - ↳ anticipate user's most common requests
    - e.g. pre-fetch data below curr scroll view
  - ↳ perform low-priority system tasks in bg
- **perceived time**:

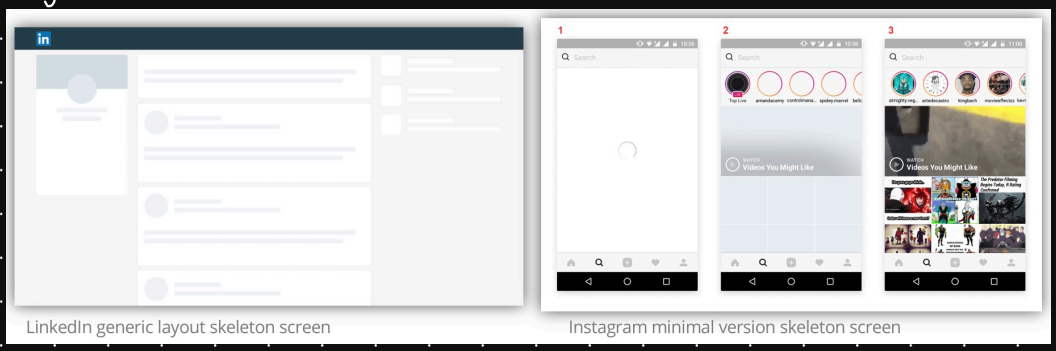
Minimal time to detect a gap of silence in sound	4 ms
Minimal time to be affected by a visual stimulus	**10 ms**
Time that vision is supressed during a saccade	100 ms
Maximum interval between cause-effect events	**140 ms**
Time to comprehend a printed word	150 ms
Visual-motor reaction time to an observed event	**1 s**
Time to prepare for conscious cognition task	**10 s**
Duration of unbroken attention to a single task	6 s to 30 s

  - ↳ min time to be affected by visual stimulus
    - continuous input latency should be < 10 ms
    - e.g. dragging shape + how far cursor is behind shape

↳ max interval btwn cause-effect events
- if UI feedback takes >140ms to appear, cause+effect perception is broken
- e.g. time btwn physically pressing UI button until UI changes



↳ visual-motor rxn time to observed event
- display busy/progress indicators for ops >1s
- present skeleton screen, which is minimal version of interface while real one loads
  - → can be generic layout or minimal version of acc content
  - → advantages:
    - user adjusts to layout they'll eventually see
    - loading process seems faster b/c there's init result
  - → e.g.

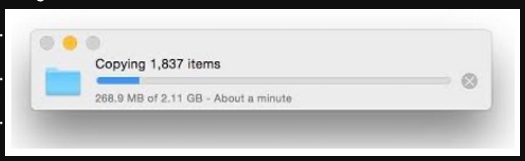


LinkedIn generic layout skeleton screen — Instagram minimal version skeleton screen

↳ time to prep for conscious cognition task
- display fake version of app interface or img of doc on last save, while real one loads in <10s

progress indicator design best practice:
↳ show work remaining, not work completed
↳ show total progress when there's multiple steps (i.e. not each step progress separately)
↳ display finished state at end briefly
↳ show smooth, not erratic, progress
↳ use human, not computer, precision
- e.g. not "243.5s remaining", instead "abt 4 mins remaining"

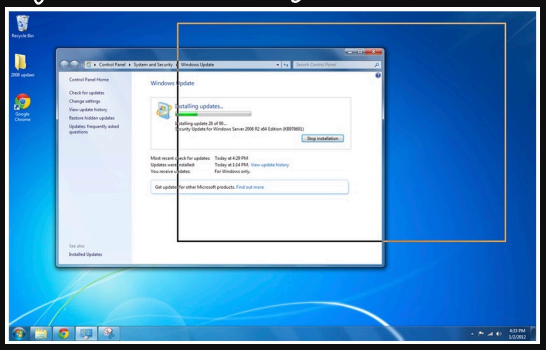
↳ e.g.



responsiveness by progressive loading: provide user w/ some data while loading rest
↳ e.g. word processor shows 1st pg as soon as doc opens
↳ e.g. search fan displays some items as soon as they're found

↳ e.g.

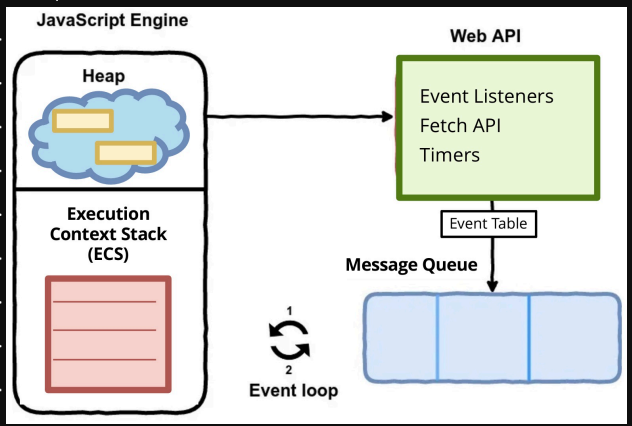
- responsiveness by predicting next op: use periods of low load to precompute responses to highly probable requests
  - ↳ e.g. text search fcn looks for next occurrence of target word while user looks at curr
  - ↳ e.g. web browser pre-downloads linked pgs (i.e. pre-fetch)
- responsiveness by graceful degradation of feedback: simplify feedback for high-computation tasks
  - ↳ e.g. window manager updates window rendering after drag only



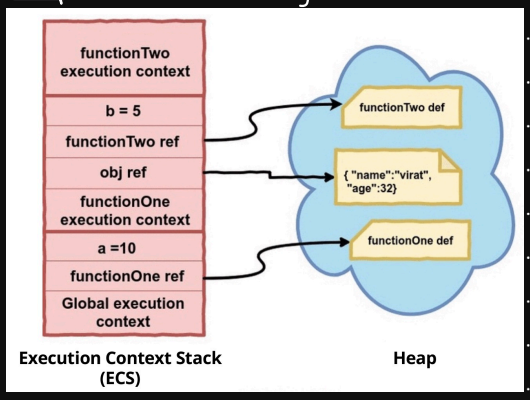
  - ↳ e.g. graphics editor only draws obj outlines during manipulation
- to handle long tasks in UI:
  - ↳ keep UI responsiveness
  - ↳ provide progress feedback
  - ↳ allow task to be paused / cancelled

JRE

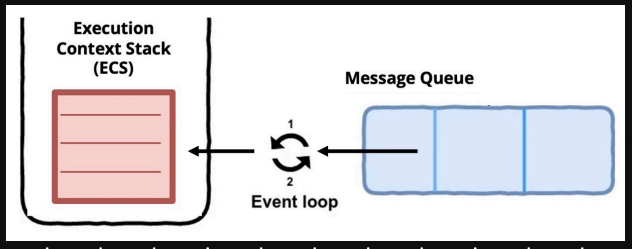
- Java Runtime Environment (JRE) provides necessary runtime support for executing Java apps
  - ↳



- in JS engine:
  - ↳ execution context stack (ECS): code to execute next
  - ↳ heap: fcn defns, objs, etc.



- in web API:
  - ↳ DOM API: handles DOM events
  - ↳ fetch API: loads remote resources
    - e.g. cloud API, server data
  - ↳ timer fcns
    - e.g. setTimeout, setInterval, requestAnimationFrame
- in msg queue (aka callback / event queue):
  - ↳ some web API methods create asynch msgs (i.e. methods are only executed when some event occurs)
    - msgs are gen. to be stored in Event Table
    - when event occurs, method put into Msg Queue
- in event loop:
  - ↳ to execute async method, must be moved to ECS
  - ↳ continually checks if ECS is empty
    - if empty, method moved from msg queue to ECS, then JS engine executes it



- input events are async methods
  - ↳ handle as callbacks bound to DOM elmt
  - ↳ e.g.

```
button.addEventListener("click", () => {
  console.log("👇 click 👇");
  // do something


});
```

FETCH API

- fetch API : interface for fetching resources across network
  - ↳ fetch() fcn starts process of fetching resource
    - returns Promise obj w/ 3 states : pending, resolved, r. rejected
- e.g. thenable chain, where we chain multiple then blocks (each containing an async op)
  - ↳ each then method is associated w/ Promise & invoked once preceding Promise is resolved
  - ↳ rejected Promise is propagated to catch block

```
/**
 * Using Fetch API with "thenable" promises
 * @param url
 */
async function doFetch1(url: string) {
  results.innerText = "Fetching ...";
  console.log("📦 Fetch1 START");
  if (loader) loader.style.display = "flex";
  // fetch is an asynchronous function that
  // returns a "response promise"
  fetch(url)
    // resolved response
    .then((response) => {
      console.log("Fetch1 response", response);
      // the json property is another promise
      return response.json();
    })
    // resolved json data in response
    .then((data) => {
      results.innerText = JSON.stringify(data, null, 2);
      if (loader) loader.style.display = "none";
    })
    // rejected
    .catch((error) => {
      console.error(error);
      results.innerText = "Error fetching data";
      if (loader) loader.style.display = "none";
    });
  console.log("📦 Fetch1 END");
}
```

- e.g. use await keyword to pause execution of fcn until Promise being awaited is resolved

```
/**
 * Using Fetch API with try/catch and await
 * @param url
 */
async function doFetch2(url: string) {
  results.innerText = "Fetching ...";
  console.log("📦 Fetch2 START");
  if (loader) loader.style.display = "flex";

  try {
    // fetch is an asynchronous function that
    // returns a "response promise"
    const response = await fetch(url);
    // resolved response
    console.log("Fetch2 response", response);
    // the json property is another promise
    const data = await response.json();
    // resolved json data in response
    results.innerText = JSON.stringify(data, null, 2);
  } catch (error) {
    // rejected
    console.error(error);
    results.innerText = "Error fetching data";
    if (loader) loader.style.display = "none";
  }
  console.log("📦 Fetch2 END");
}
```

- complex to get process during fetch

WORKER THREADS

when long tasks aren't handled asynchronously, everything else is blocked

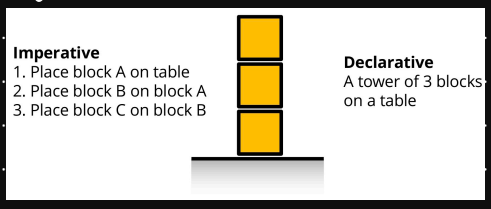
- multi-threading: manage multiple concurrent threads w/ shared resources but executing diff ins
  - ↳ threads can divide computations + reduce blocking
  - ↳ concurrency risks
    - e.g. 2 threads simultaneously update var
- browsers support worker threads, which are bg threads that can run scripts w/o interfering w/ UI
  - ↳ dedicated workers
  - ↳ shared workers
  - ↳ service workers

DECLARATIVE

UI PROGRAMMING PARADIGMS

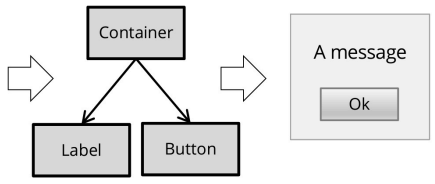
- imperative programming : describe how to achieve result
- declarative programming : describe what result we want

e.g.



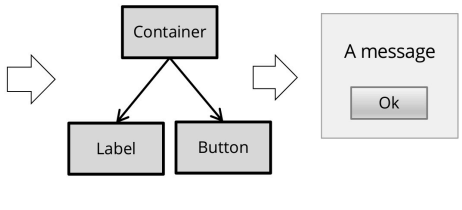
- SK uses imperative paradigm to build UI
  - ↳ write code how to make tree of nodes w/associated events
  - ↳ e.g.

```
const root = new SKContainer()

const l = SKLabel("A message")
root.addChild(l)

const b = SKButton("Ok")
root.addChild(b)
b.addEventListener("action",
 () => doAction() );

setSKRoot(root)
```



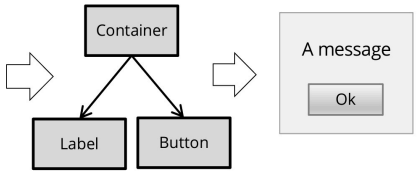
- HTML can be declarative paradigm to build UI
  - ↳ write markup to describe what tree of nodes w/events (i.e. DOM) is
  - ↳ e.g.

```
<div class="container">
  <p>
    A message
  </p>
  <button
    onclick="doAction()">
    Ok
  </button>
</div>
```



- some DOM manipulation methods has mix of declarative & imperative paradigms
  - ↳ HTML to declaratively describe UI layout
  - ↳ TS to imperatively add events or modify layout
  - ↳ e.g.

```
<div class="container">
  <p>
    A message
  </p>
  <button id="b">Ok</button>
</div>

// in main.ts
const b = document.
  querySelector("button#b");
b.addEventListener("click",
  () => doAction() );
```



setup for following demos:

- Simple state

```
// state
let clicked = false;
function setClicked(value: boolean) {
  clicked = value;
  update();
}
```

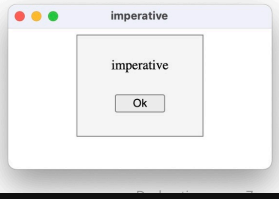
Keeping it simple without MVC, better reactive methods for state are shown in next lecture

- When state changes, app is re-rendered

```
// when state changes, re-render the App
function update() {
  ...
}

// initial render
update();
```

↳ e.g. imperative

```
// create the UI tree for the app
function App() {
  const container = document.createElement("div");
  container.classList.add("container");

  const label = document.createElement("p");
  label.innerText = clicked ? "CLICKED" : "imperative";
  container.appendChild(label);

  const button = document.createElement("button");
  button.innerText = "Ok";
  container.appendChild(button);

  button.addEventListener("click", () => {
    setClicked(true);
  });

  return container;
}

// when state changes, re-render the app
... root.replaceChildren(App());
```



↳ e.g. declarative - html

```
<div class="container">
  <p id="label">???</p>
  <button onClick="setClicked(true)">Ok</button>
</div>


<script>
  // state
  ...

  // when state changes, update the app
  function update() {
    document.querySelector("p#label").innerText =
      clicked ? "CLICKED" : "declarative-html";
  }
</script>
```

but update code is imperative

↳ e.g. declarative-string

```
// create the UI tree for the app
function renderApp(root: Element) {
  root.innerHTML = html`
    <div class="container">
      <p>${clicked ? "CLICKED" : "declarative-string"}</p>
      <button>Ok</button>
    </div>
  `;
  document.querySelector("button")?
    .addEventListener("click", () => {
      setClicked(true);
    });
}

// when state changes, re-render the app
function update() {
  const root =
    document.querySelector("#app")
      as Element;
  renderApp(root);
}
```

but add listener code is imperative

↳ e.g. declarative-htm

```
// create the UI tree for the app
export function App() {
  return html`
    <div class="container">
      <p>${clicked ? "CLICKED" : "declarative-htm"}</p>
      <button onClick=${() => setClicked(true)}>Ok</button>
    </div>
  `;
}

// when state changes, re-render the app
function update() {
  render(html`<${App} />`,
    document.querySelector("#app") as Element);
}
```

all declarative

- HTM is HS tagged template

HYPERSCRIPT

HyperScript (HS) is lower lvl lang used to gen descriptions of UI trees

↳ hyperscript is npm pkg to gen HS

↳ e.g.

```
const msg = "hi hyperscript";
h("div", { class: "container" }, [
  h("p", {}, msg),
  h("button", {}, "Ok"),
]);
```

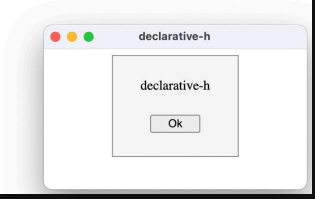
⇨

```
<div class="container">
  <p>hi hyperscript</p>
  <button>Ok</button>
</div>
```

- e.g. declarative-h

```
// create the UI tree for the app
function App() {
  return h("div", { class: "container" }, [
      h("p", null, clicked ? "CLICKED" : "declarative-h"),
      h("button", { onClick: () => setClicked(true) }, "Ok")
  ]);
}

// when state changes, re-render the app
function update() {
  render(App(),
    document.querySelector("#app") as Element);
}
```

all declarative



- HS fcn calls create rep of UI tree, which is JS obj
  - ↳ aka virtual DOM (vdom)
  - ↳ used for 2 purposes:
    - render acc DOM using imperative methods
    - lightweight abstraction of DOM to compare changes
- HS virtual node fcn recursively gens vdom obj
  - ↳

```
export function h(tag, attributes = {}, ... _children) {
  // force children to be an array (possibly empty)
  let children = [].concat(... _children);
  return { tag, attributes, children };
}
```

JavaScript

- e.g. render HS virtual node to DOM elmt

```
function _render(vnode) {
  // if vnode is a string, create text node
  if (typeof vnode === "string") {
    return document.createTextNode(vnode);
  }
  // vnode is a tag, so create corresponding DOM element
  let el = document.createElement(vnode.tag);
  // copy vnode attributes into new DOM element
  let attributes = vnode.attributes || {};
  Object.keys(attributes).forEach((k) => {
    el.setAttribute(k, attributes[k]);
  });
  // recursively render child nodes
  (vnode.children || []).forEach(
    (c) => el.appendChild(_render(c)));
  return el;
}
```

How to "render" **event attributes** like onClick?

JavaScript

- ↳ e.g. rendering HS w/event attr

- Example hyperscript definition with event

```
h("button",
  { onClick: () => (console.log("🔥 CLICKED!")) },
  "Ok");
```

handler

onClick attribute

- Additional render code to "render" event listeners

```
Object.keys(attributes).forEach((k) => {
  if (k.startsWith("on")) {
    // special case for event listener attributes
    const type = k.substring(2).toLowerCase();
    const handler = attributes[k];
    el.addEventListener(type, handler);
  } else {
    // just a normal attribute
    el.setAttribute(k, attributes[k]);}
});
```

- use HS to declaratively build vdom w/ virtual nodes, then call `render` which imperatively builds DOM
- HTM is syntactic sugar for HS
  - ↳ e.g.

- HTM declarative syntax:

```
html`<p style="color: red;">Example B</p>`;
```

- Equivalent hyperscript:

```
h("p", { style: "color: red;" }, "Example B");
```

PREACT AND JSX

- `library`: collection of code (type, objs, fcns, etc.)
  - ↳ use parts of it, typically flexibly
  - ↳ programmer calls lib code
  - ↳ unopinionated collections of code
  - ↳ e.g. Preact, React, Bootstrap
- `framework`: struct to use collection of provided code
  - ↳ can modify + extend it, but must follow rules
  - ↳ framework calls programmer's code
  - ↳ opinionated libs
  - ↳ e.g. Angular, NextJS
- `toolkit`: collection of compatible libs
  - ↳ e.g. Bootscrap
- `tool`: software to perform specific task
  - ↳ e.g. Vite
- key diffs of `Preact` from React:
  - ↳ uses native DOM events
    - React has own synthetic event system
    - e.g. onInput vs onChange, onDblClick vs onDoubleClick
  - ↳ treats Children nodes as native JS arrs
    - React has own obj for managing Children

NOTE
Preacts aims to be largely compatible w/ React API
↳ use preact/compat

↳ supports "class" to set class attr
  - React uses "className"

to setup Preact using Vite:
↳ npm create vite@latest
  - choose Preact
  - choose TypeScript

Preact Node proj similar to vanilla TS, but adds support for .jsx files

JSX describes DOM trees w/ mix of JS + HTML
↳ JS files w/ JSX have .jsx extension
  - TS files have .tsx extension
↳ files w/ JSX compiled into JS w/ HS fcn calls
  - e.g.

```
const msg = "hi JSX";
<div class="container">
    <p>{msg}</p>
    <button>Ok</button>
</div>
```

JSX and HTML look almost the same

```
<div class="container">
  <p>hi JSX</p>
  <button>Ok</button>
</div>
```

JSX is syntactic sugar for HS
↳ JSX compiled into HS
↳ HS used to render
  - h fcn creates vdom obj
  - render fcn creates DOM from vdom

components are building blocks of Preact app
↳ have custom properties (i.e. props)
↳ e.g. custom component

```
const vdom = (
  <div>
    <h1>My component is below here:</h1>
    <MyComponent msg="hi component" />
  </div>
);
```

a custom component with a custom msg "prop"

fcns are most common way to create components
↳ e.g.

```
function MyComponent(props: { msg: string }) {
  return (
    <div class="container">
      <p>{props.msg}</p>
      <button>Ok</button>
    </div>
  );
}
```

a custom "prop"

class components are no longer common
↳ use fcnal components instead
↳ e.g.

```
class MyComponent extends Component<{ msg: string }> {

  constructor(props: { msg: string }) {
    super(props);
  }                                  a custom "prop"

  render() {
    return (
      <div class="container">
        <p>{this.props.msg}</p>
        <button>Ok</button>
      </div>
    );
  }
}
```

- components can be nested like HTML elmts
  - ↳ can have HTML / Component nodes as `children`
  - ↳ enables control over how vdom elmts nested in component should be rendered
  - ↳ `arr of children` is special implicit prop
  - ↳ e.g.

```
function Container(props: { children: any }) {
  return <div class="container">{props.children}</div>;
}                                  render children as they are

const vdom = (
  <Container>
    <p>Text</p>
    <button>Ok</button>
  </Container>
);
```

- TS requires type defn for Component props
  - ↳ best practice is to define `MyComponentProps type`
    - e.g.

```
type NumberBoxProps = {
  num: number;
  colour?: string;
};          optional prop          default prop value

function NumberBox({ num, colour = "grey" }: NumberBoxProps) {
  return
    <div style={`background-color: ${colour};`}>
      {num}
    </div>;
}
```

  - ↳ `destructure` props to avoid props.myprops syntax
    - easier to assign default props vals
    - e.g.

```
type NumberBoxProps = {
  num: number;
  colour: string;
};

function NumberBox({ num, colour }: NumberBoxProps) {
  return
    <div style={`background-color: ${colour};`}>
      {num}
    </div>;
}
```

Without props argument destructuring:

```
function NumberBox(props: NumberBoxProps) {
  return
    <div style={`background-color: ${props.colour};`}>
      {props.num}
    </div>;
}
```

Preact uses std DOM events w/ declarative syntax + we can define events in components

↳ if event handler small, include fcn defn inline

- e.g.

```
const jsx = <button onClick={() => console.log("click")}>
  Click
</button>
```

↳ if event handler is more complex, call handler fcn

- e.g.

```
function handleClick() {
  console.log("click");
}

const jsx = <button onClick={handleClick}>Click</button>
```

↳ event handlers can be passed as props to components

JSX must eval to an expr (i.e. valid unit of code that resolves to a val)

↳ JSX + everything in it are exprs

↳ use {} to insert JS into JSX expr

↳ e.g. insert val of var

```
const msg = "Hello World";
const jsx = <p>{msg}</p>;
```

↳ e.g. iterate thru arr w/ map

```
const items = ["a", "b", "c"];
const jsx = <ul>{ items.map((item) =>
    <li>{item}</li>
  )}</ul>;
```

↳ e.g. conditional logic w/ ternary operator

```
const jsx = <p>{isDone ? "Done" : "Not Done"}</p>
```

2 options for dynamic attrs vals:

↳ template literal

- e.g.

```
function NumberBox({ num, colour }: NumberBoxProps) {
  return <div style={`background-color: ${colour};`}>
          {num}
        </div>;
}
```

template literal with JavaScript expression

↳ HS obj

- e.g.

```
function NumberBox({ num, colour }: NumberBoxProps) {
  return <div style={{backgroundColor: colour}}>
          {num}
        </div>;
}
```

note camelcase used, not - separator

- components must return 1 root node
  - ↳ if we don't have single root node, wrap component nodes in `<Fragment>` node
  - ↳ e.g.

```
function App() {
  return (
    <LeftView />
    <RightView />
  );
}
```

Error:
More than one root node not allowed

```
function App() {
  return (
    <div>
      <LeftView />
      <RightView />
    </div>
  );
}
```

Solution 1:
Insert to make one root node

Issue:
extra in DOM

```
function App() {
  return (
    <Fragment>
      <LeftView />
      <RightView />
    </Fragment>
  );
}
```

Solution 2:
Use special <Fragment> node to make one root node

Note:
<Fragment> is not rendered in DOM

```
function App() {
  return (
    <>
      <LeftView />
      <RightView />
    </>
  );
}
```

Solution 2 Variation:
Use <> and </> as shorthand for <Fragment>

REACTIVE

VIRTUAL DOM RECONCILIATION

- reactivity: auto update of UI due to change in app's state
  - ↳ developer focuses on state of app
  - ↳ framework reflects state in UI
- VDOM is lightweight rep of UI in mem
- to sync VDOM w/ real DOM:
  1) save curr VDOM
  2) components + app state updates VDOM
  3) re-render triggered by framework
  4) compare VDOM before update w/ after update
  5) reconcile diff by identifying set of DOM patches
     - ↳ minimizes DOM manipulations
  6) perform patch ops on real DOM
  7) go back to step 1
- if node type changes, whole subtree rebuilt
  - ↳ e.g.

```
<div><p>Hello</p></div>
  ⇓
<section><p>Hello</p></section>
```

- if node type is same, attrs are compared + updated
  - ↳ e.g.

```
<div class="foo"><p>Hello</p></div>
  ⇓
<div class="foo bar"><p>Hello</p></div>
```

- if node is inserted into list of same node type siblings, all children are updated (if more info isn't provided)
  - ↳ e.g.

```
<ul>
  <li>Apple</li>
  <li>Banana</li>
</ul>
  ⇓
<ul>
  <li>Pear</li>
  <li>Apple</li>
  <li>Banana</li>
</ul>
```

first <li> must have changed Apple to Banana,

second <li> must have changed Banana to Apple,

etc.

- when updating children of same node type., use key prop
  - ↳ each key must be stable + unique

↳ key should be assigned when data is created, not when rendered
- don't use iteration index for key ids when rendering list

↳ e.g.

```
<ul>
  <li key="a">Apple</li>
  <li key="b">Banana</li>
</ul>
```

⇓

```
<ul>
  <li key="p">Pear</li>
  <li key="a">Apple</li>
  <li key="b">Banana</li>
</ul>
```

no other <li> with key-"p", so much be inserting a new child

STATE

approaches to managing state:

↳ useState hook for local component state
- pass state to children

↳ useContext hook to access state w/o passing as props

↳ signals

↳ redux
- not covered

e.g. useState

```
import { render } from "preact";
import {
  useCallback,
  useEffect,
  useLayoutEffect,
  useState,
} from "preact/hooks";

import LeftView from "./Left";
import RightView from "./Right";

import "./style.css";

console.log("counter");

// function useCounter() {
//   const [value, setValue] = useState(0);
//   const increment = useCallback(() => {
//     setValue(value + 1);
//   }, [value]);
//   return { value, increment };
// }
```

```
export default function App() {
  const [count, setCount] = useState(0);   //hook

  // useEffect(() => {
  //   console.log("useEffect");
  // }, [count]);

  // event handler to pass to component
  function handleClick() {
    console.log("click");
    // update state
    setCount(count + 1);
  }

  return (
    <>
      <LeftView count={count} handleClick={handleClick} />
      <RightView count={count} colour="pink" />
    </>
  );
}

render(<App />, document.querySelector("div#app") as Element);
```

prop drilling: pass props (i.e. data) down multiple layers of nested components by passing from parent to children

hook: fcnal methods to compose state & side effects

↳ works by storing data in seq of slots associated w/each component in VDOM tree

↳ calling hook fcn uses ! slot & increments internal slot # counter (i.e. site ordering)
- Preact resets counter before invoking each component so each hook is

associated w/ same slot when component rendered multiple times
  ↳ have specific order of execution
  ↳ hooks can't be called conditionally or within loops

common hooks:
  ↳ useState
    - get & set state
  ↳ useContext
    - access state context w/o prop drilling
  ↳ useRef
    - get ref to DOM node inside fanal component
  ↳ useEffect
    - trigger side-effects on state change
  ↳ useReducer
    - complex state logic similar to Redux

can define & use custom hooks
  ↳ e.g.

- Define a custom counter hook:

```
function useCounter() {
  const [value, setValue] = useState(0);
  const increment = useCallback(() => {
    setValue(value + 1);
  }, [value]);
  return { value, increment };
}
```

dependency argument

- Use the custom counter hook:

```
export default function App() {
  const { value, increment } = useCounter();
  return (
    <>
      <LeftView count={value} handleClick={increment} />
      <RightView count={value} colour="pink" />
    </>
  );
}
```

useContext hook allows us to pass state & other vals to children w/o prop drilling

1) define context obj type

```
export type MyContextType = { colour: string; };
```

2) create shared context obj

```
export const MyContext = createContext({} as MyContextType);
```

3) make context Provider node ancestor of components using context

```
<MyContext.Provider value={{colour: "lightgreen"}}>
   <MyComponent />
</MyContext.Provider>
```

initialize the context object

4) use context in child components

```
import { myContext } from "...";
function MyComponent() {
  const { colour } = useContext(MyContext);
  ...
```

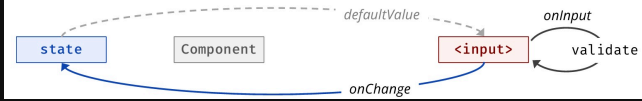
- signals: state management module intro by Preact
  - ↳ can be used w/ React + other frameworks
  - ↳ not included by default
    - npm install @preact/signals
- suggestions for states w/ signals
  - ↳ keep state minimal + well-organized
    - each signal focuses on specific part of state
  - ↳ use signals only where needed
    - too many signals can have performance issues
    - use signals to share state btwn diff parts of app
    - for local app state, use useState hook
  - ↳ keep components small + focused
    - all-in-1 components handling several tasks makes state hard to manage
    - break UI down into components that each manage specific piece of state

EFFECTS

CONTROLLED / UNCONTROLLED COMPONENTS

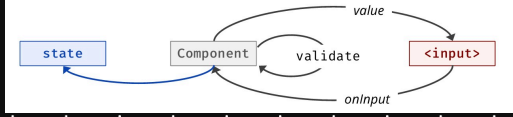
- component data flow: how HTML form elmt vals are associated w/app state
  - ↳ uncontrolled: elmts manage own data vals
    - e.g.

```
<input defaultValue={someValue} onChange={myEventHandler} />;
```



  - ↳ controlled: app state manages elmt data vals thru component
    - e.g.

```
<input value={someValue} onInput={myEventHandler} />;
```



- e.g. uncontrolled component

```tsx
import { appState } from "./state";

export default function Uncontrolled() {
  function handleInput(e: Event) {
    const newValue = (e.target as HTMLInputElement).value;
    // update state
    appState.value = newValue;
  }

  return (
    <>
      <p>Uncontrolled</p>
      <input type="text" onChange={handleInput} />
    </>
  );
}
```

  - ↳ appstate is str signal
  - ↳ onChange event sets state only on focus loss
- e.g. controlled component

```tsx
import { useState } from "preact/hooks";
import { appState } from "./state";

export default function Controlled() {
  const [invalid, setInvalid] = useState(false);

  // regex validation
  const isValid = (text: string) => /^[abc]*$/.test(text);

  function handleInput(e: Event) {
    const el = e.target as HTMLInputElement;
    // validate
    setInvalid(!isValid(el.value));
    // update state (and input value)
    appState.value = el.value;
  }
```

NOTE
Generally use controlled components

```
  return (
    <>
      <p>Controlled</p>
      <input
        type="text"
        value={appState.value}
        class={invalid ? "invalid" : ""}
        onInput={handleInput}
      />
      {invalid && <p class="error">Invalid: use a, b, or c</p>}
    </>
  );
}
```

↳ no need for defaultValue to init to state

↳ onInput calls handler which does validation + updates state

e.g. controlled conditional component

```
import { useEffect, useState } from "preact/hooks";
import { appState } from "./state";

export default function ControlledConditional() {
  // local state for the input value
  const [inputValue, setInputValue] = useState(appState.value);

  // update local state when app state changes
  useEffect(() => {
    setInputValue(appState.value);
  }, [appState.value]);

  // regex validation
  const isValid = (text: string) => /^[abc]*$/.test(text);

  // handler for input changes
  const handleInput = (e: Event) => {
    const el = e.target as HTMLInputElement;
    // Update local state immediately
    setInputValue(el.value);
    // only if valid, update the app state
    if (isValid(el.value)) {
      appState.value = el.value;
    }
  };

  return (
    <>
      <p>ControlledConditional</p>
      <input
        type="text"
        value={inputValue}
        class={!isValid(inputValue) ? "invalid" : ""}
        onInput={handleInput}
        // always leave input field with valid value
        onChange={() => setInputValue(appState.value)}
      />
      {!isValid(inputValue) && (
        <p class="error">Invalid: use a, b, or c</p>
      )}
    </>
  );
}
```

↳ input text changes state only when valid

side effects are changes outside VDOM render

↳ code is run as result of VDOM change

↳ reach out of VDOM tree to mutate state or invoke imperative code (e.g. calling DOM APIs)

↳ e.g. DOM manipulation (e.g. classlist.AddClass), drawing in canvas graphics

context, fetching data
↳ Preact side effect methods:
- useRef hook
- useEffect hook
- useLayoutEffect hook

CANVAS COMPONENT

e.g. canvas

```tsx
import { useRef, useEffect, useLayoutEffect } from "preact/hooks";

type CanvasProps = {
  width?: number;
  height?: number;
};

export function Canvas({ width = 256, height = 256 }: CanvasProps) {
  const canvasRef = useRef<HTMLCanvasElement>(null);

  // drawing
  useLayoutEffect(() => {
    const gc = canvasRef.current?.getContext("2d");
    if (gc) draw(gc);
  }),
    [];

  function draw(gc: CanvasRenderingContext2D) {
    gc.fillStyle = "black";
    gc.fillRect(0, 0, gc.canvas.width, gc.canvas.height);

    gc.fillStyle = "red";

    // 100 random points
    [...Array(100)].forEach((_) => {
      const [x, y] = [Math.random() * width, Math.random() * height];
      gc.beginPath();
      gc.arc(x, y, 5, 0, 2 * Math.PI);
      gc.fill();
    });
  }

  return <canvas ref={canvasRef} width={width} height={height} />;
}
```

- `useEffect` handles side effects after render while `useLayoutEffect` is a pre-render hook for immediate DOM mutations
  ↳ useEffect is async
  ↳ useLayoutEffect is sync
- `ResizeObserver` is HTML DOM feature
  ↳ window resize for elmts

CSS STYLE FOR COMPONENTS

- approaches:
  ↳ global link
  ↳ inline CSS attrs in component render
  ↳ import component specific CSS module file
  ↳ utility classes (e.g. Tailwind)

↳ styled-components pkg in React
- not covered

inline CSS : style attr accepts JS obj to specify CSS props

↳ e.g.

```
"padding: 10px" → { padding: "10px" }
"padding: 10px" → { padding: 10 }
"flex: 1 1 auto → { flex: "1 1 auto" }
```

number variable or constant translated to "px" units

↳ JS var names can't have "-" char so use camelCase
- e.g.

```
"flex-flow: row nowrap" → { flexFlow: "row nowrap" }
```

↳ create const obj + assign in render
- e.g.

```jsx
import { render } from "preact";

import LeftView from "./Left";
import RightView from "./Right";

// global styles (e.g. reset)
import "./style.css";

console.log("style-inline");

// inline styles
const rootStyle = {
  backgroundColor: "whitesmoke",
  display: "flex",
  justifyContent: "center",
  alignItems: "stretch",
  height: "100vh",
};

const containerStyle = {
  margin: "50px",
  flex: "1 1 auto",
  display: "flex",
  flexDirection: "row",
  flexFlow: "row nowrap",
  alignItems: "stretch",
  gap: "50px",
};

export default function App() {
  return (
    // app "root"
    <div style={rootStyle}>
      {/* container */}
      <div style={containerStyle}>
        {/* views */}
        <LeftView />
        <RightView colour="pink" />
      </div>
    </div>
  );
}

render(<App />, document.querySelector("div#app") as HTMLElement);
```

↳ don't provide flexibility w/ rules, selectors, etc.

CSS modules : CSS file w/ ".module.css" extension imported into component tsx file

↳ assign classes in render

↳ class names are locally scoped to component, meaning unique class names are gen

- e.g. if class={style.root}, then class="root_dsfsdf_1" is gen

↳ e.g.

Left.tsx 258 B

```tsx
// app state
import { count, increment } from "./state";

import style from "./Left.module.css";

export default function LeftView() {
  return (
    <div class={style.root}>
      <button onClick={() => increment()}>{count.value}</button>
    </div>
  );
}
```

Left.module.css 193 B

```css
.root {
  padding: 10px;
  border: 1px solid grey;
  background-color: white;
  flex: 1 1 0;
  display: flex;
  align-items: center;
  justify-content: center;
}

button {
  min-width: 80px;
}
```

TAILWIND

Tailwind is utility-first CSS framework

↳ utility classes class names are like style props

↳ no need for media queries, but use size prefix instead

↳ bundler removes all unused CSS for prod

- most Tailwind projs ship <10kB of CSS to client

↳ works best w/ declarative UI

- e.g. Preact components

no predesigned components (e.g. buttons, cards, alerts)

↳ extreme CSS reset called preflight

- we must add styling for everything
- can disable + use browser defaults

e.g. basic styling

↳ default button

```html
<button>Button</button>
```

renders: Button (just clickable text)

↳ style w/ utility classes

```html
<button class="py-0.5 px-2 bg-gray-200
        border border-gray-600 rounded
        hover:bg-gray-300 active:bg-gray-200"
>Button</button>
```

renders: Button

↳ style w/ inline utility classes in std CSS class

- in style.css: (usually in **@layer base** (a Tailwind defined CS layer))

```
button {
  @apply py-0.5 px-2 bg-gray-200
         border border-gray-600 rounded
         hover:bg-gray-300 active:bg-gray-200;}
```

then:

```
<button>Button</button>
```

renders:

Button

setup in tailwind config ts: must add html files to types for processing

```
/** @type {import('tailwindcss').Config} */
export default {
  content: ["./index.html", "./src/**/*.{js,ts,jsx,tsx,html}"],
  theme: {
    extend: {},
  },
  /* remove Tailwind CSS reset */
  // corePlugins: {
  //   preflight: false,
  // },
};

```

Tailwind is mobile-first responsive

↳ provides {screen:} prefix instead of using media queries/rules

- identifies when to use utility class
- default screen is mobile (i.e. no prefix)

↳ e.g.

```
<div class="... flex flex-col sm:flex-row ...">
```

column for mobile

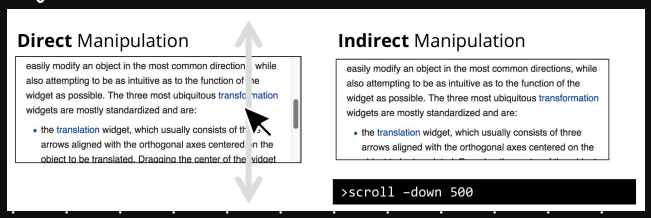
row for tablet and larger

plug-ins register new styles for Tailwind to inject into user's stylesheet (using JS instead of CSS)

DIRECT MANIPULATION

INSTRUMENTAL INTERACTION

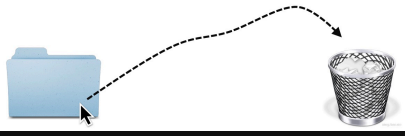
- **direct manipulation**: when virtual rep of obj is manipulated in similar way to real-world obj
  - ↳ e.g.



  - ↳ requires rep of **task objs** (i.e. smth user can manipulate)
    - can be **obj of interest** (e.g. file) or **interface widget** (e.g. cancel button)

- e.g. analogous behaviours

Real World	Direct Manipulation Interface
Object to be discarded	Icon of object to be discarded
Move hand to object	Move pointer to object
Pick up object with hand	Click to acquire object
Waste basket	Waste basket icon
Move to waste basket	Drag to waste basket icon
Release object from hand	Release button to discard object



- **Schneiderman's characteristics** of direct manipulation:
  - ↳ continuous rep of task objs + actions
  - ↳ task objs manipulated by physical actions, not complex syntax
  - ↳ fast, incremental, + reversible actions w/effects on task objs immediately apparent
  - ↳ layered, self-revealing approach to learning

benefit of direct manipulation is that we feel as if we're interacting w/task obj rather than w/interface so we focus on task rather than on tech

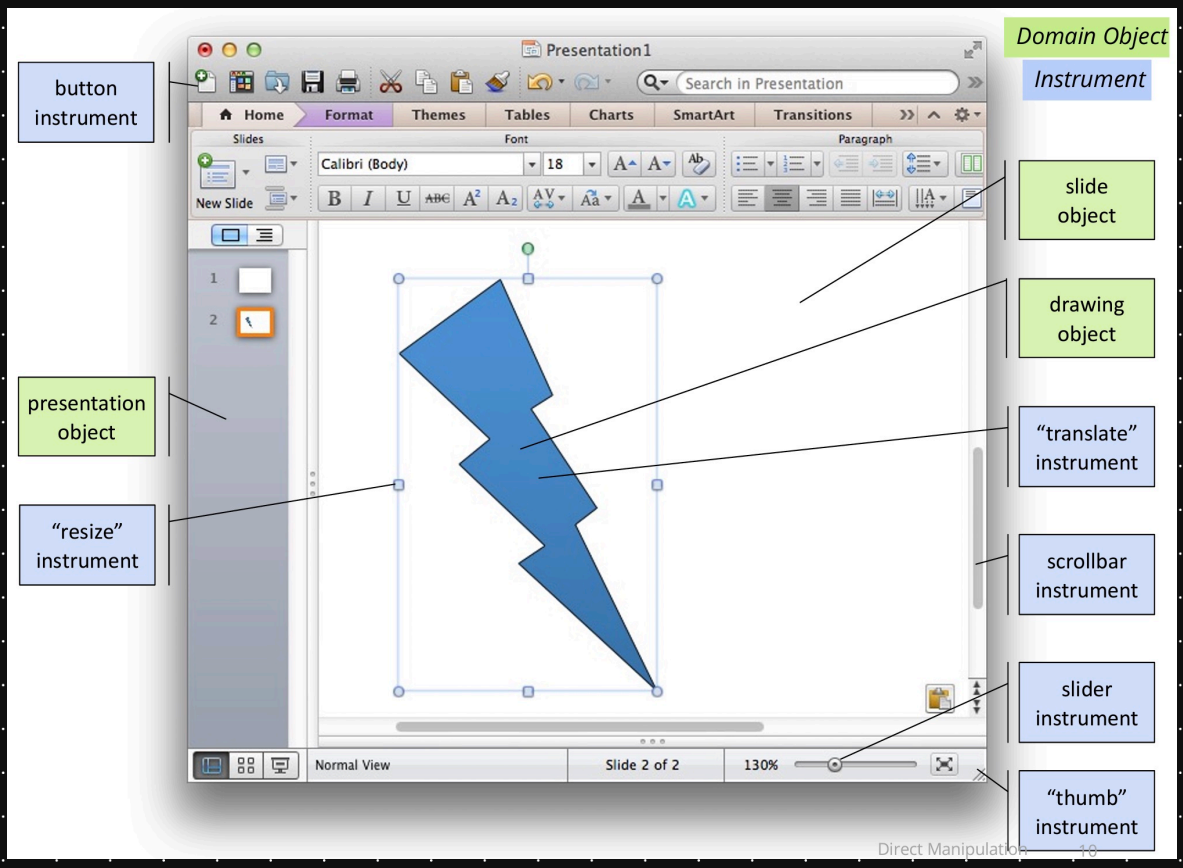
  - ↳ feeling of direct involvement w/world of task objs rather than communication w/intermediary

- **interaction model**: describes how to combine interaction techniques in a meaningful + consistent way
  - ↳ props can be used to eval specific interaction designs

interfaces have:

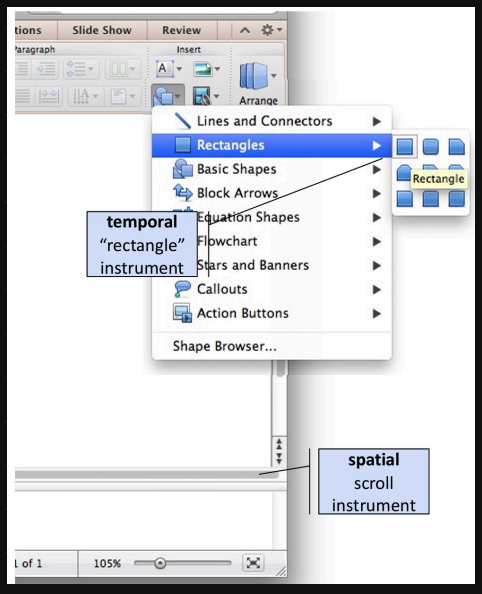
  - ↳ **interaction instruments**: necessary mediator btwn user + domain objs
  - ↳ **domain objs**: thing of interest, data, + associated attrs, which is

manipulated using interaction instrument
↳ e.g.



activation: how instrument is triggered for use
↳ GUI instruments activated in 2 ways:
- spatially has movement cost
- temporally has time cost
  → always has spatial cost as well

↳ e.g.



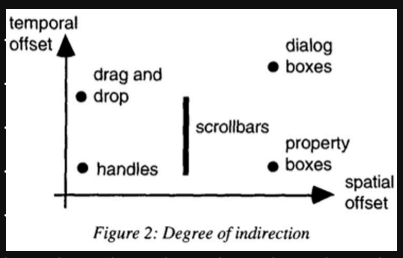
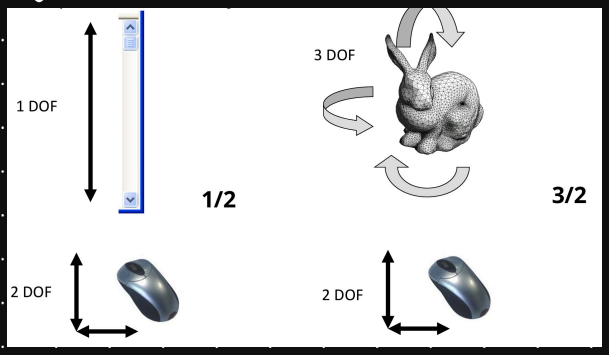
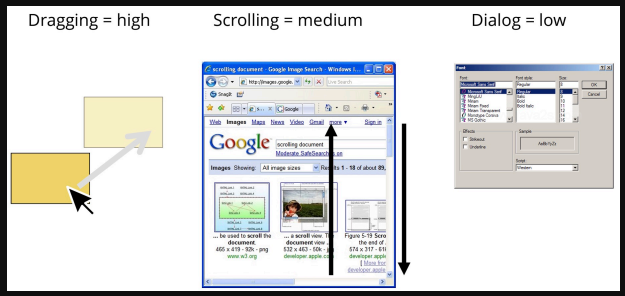
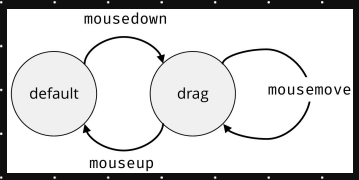
- **reification**: turning concepts into smth concrete
  - ↳ instrument is reification of cmd
    - e.g. scrollbar reifies cmd to move content inside window
- **meta-instrument**: instrument that acts on instrument
  - ↳ i.e. other instrument becomes obj of interest
  - ↳ e.g. paper (obj), pencil (instrument), pencil sharpener (meta-instrument)
- **obj reification**: turning attrs of primary obj into other objs of interest
  - ↳ e.g. colour swatch, font styles
- 3 props to describe instruments:
  - ↳ **deg of indirection**: spatial / temporal offset btwn instrument + action on obj
    - 2d measure of dist from instrument to obj
    - 

Figure 2: Degree of indirection
  - ↳ **deg of integration**: match btwn input device to instrument
    - ratio of **deg of freedom (DOF)** of instrument over DOF captured by input device
    - captures suitability of device
    - e.g.


  - ↳ **deg of compatibility**: similarity of action on device / instrument to response of obj
    - more similar is better b/c it's more intuitive
    - e.g.


direct manipulation interface allows user to directly act on set of objs in UI
- ↳ i.e. direct manipulation is form of instrumental interaction
- ↳ ideally:
  - low indirection (low spatial + temporal offsets)
  - high integration (1:1)
  - high compatibility
- ↳ ideal lvl of direct manipulation is when instruments are visually indistinguishable from objs they control
  - actions on instrument/obj entities are analogous to actions on similar objs in real word + preserve conceptual linkage btwn instrument + obj

dragging
- ↳ mousedown on shape starts drag
  - calc **offset** from pos to shape frame-of-ref (e.g. when dragging circle, mouse doesn't snap immediately to centre of circle)
- ↳ mousemove to drag
- ↳ mouseup to end drag
- ↳ state machine:



e.stopImmediatePropagation() to prevent further propagation of curr event in capture + bubble phases

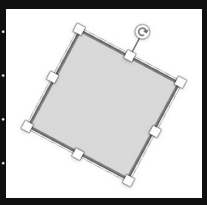
most UI toolkits have built-in support for **drag + drop**
- ↳ drag + drop interaction:
  1) mousedown on src obj/data
  2) drag src obj onto target obj
  3) mouseup to drop obj/data on target
- ↳ e.g. drag file to upload

HTML drag + drop API enables drag+drop interactions in browser apps
- ↳ make elmt draggable w/ attr `<div draggable="true" ...>`
- ↳ manage drag op by handling drag+drop events
  - dragstart
  - drag
  - dragenter
  - dragleave
  - dragover
  - drop
  - dragend
- ↳ provides **direct manipulation feedback**
  - visual indication of elmts that can be dragged

- shows possible drop targets
  - ↳ use `e.preventDefault()`

TRANSFORMABLE

- transformable shape: direct manipulation widgets used to transform obj
  - ↳ scale by dragging corner + side handles
  - ↳ rotate by dragging rotation handle
  - ↳ translate by dragging shape
  - ↳ e.g.



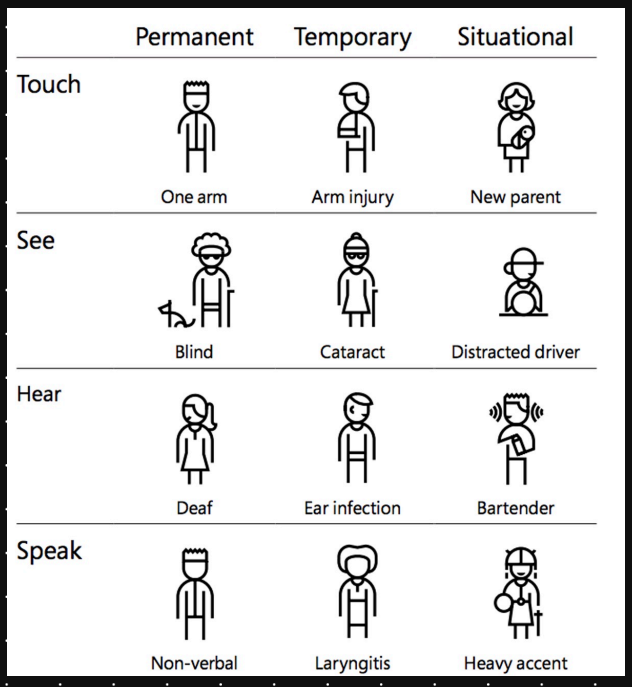
ACCESSIBILITY

TYPES OF DISABILITIES

- accessibility: developing content to be as accessible as possible, no matter an individual's physical + cognitive abilities + how they access UI
  - ↳ aka. a11y
  - ↳ parallel w/ physical accessibility
- access to UI is human right
- ppl have range of ability dimensions (i.e. characteristics + factors that influence capabilities)
  - ↳ e.g.

Age	Physical	Lived Experience
Gender	Culture	Emotional
Cognitive	Language	Spiritual

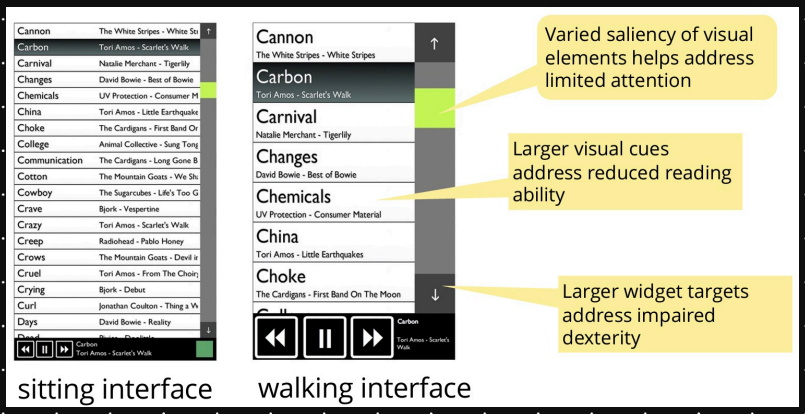
  - ↳ "avg person" is statistical construct
- types of disabilities:



- disabilities can be temp / situational
  - ↳ e.g. sick / injured, driving, underwater diving

ACCESSIBILITY APPROACHES

- to address situational walking impairment where there's:
  - ↳ reduced dexterity + motor control
  - ↳ reduced cognitive ability
  - ↳ e.g.

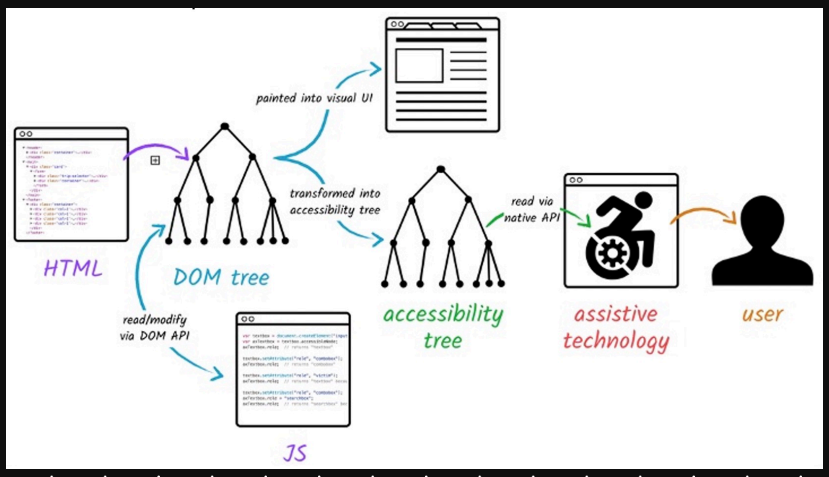


- interfaces for age-related impairments:
  - ↳ keep info simple (cognitive)
  - ↳ high contrast colours, large text, + large icons (vision)
  - ↳ large widget + button sizes (motor)
- must also consider chronic + long-term disabilities
  - ↳ 10% to 20% of pop
- modern OS support for accessibility:
  - ↳ control cursor from keyboard (motor)
  - ↳ adjust acceleration, tracking, + precision (motor)
  - ↳ speech dictation (visual / motor)
  - ↳ magnify portions of screen, adjust elmt sizes / font sizes, + provide full voice dictation (visual)
  - ↳ captions / subtitles (audial)
- interface enhancements for visual impairments:
  - ↳ zoom screen / specific area
  - ↳ inc font size
  - ↳ high contrast colours, dark mode, remove animations
  - ↳ screen reader, voice input
  - ↳ real world magnifier
- interface enhancements for hearing impairments:
  - ↳ show audio alerts visually
    - e.g. vibrate, flashlight alarm
  - ↳ realtime audio processing to filter bg noise + amplify another voice
  - ↳ monitor audio for certain sounds + send alert
    - e.g. baby crying
- interface enhancements for motor impairments:
  - ↳ sticky, slow, + filter keys
  - ↳ reduce key repeat rate
  - ↳ eye tracking
  - ↳ voice input
  - ↳ physical switches + "puffers"
  - ↳ brain-computer interfaces (BCI)

- interface enhancements for **cognitive impairments**:
  - ↳ word prediction, grammar, + spell check
  - ↳ text-to-speech
  - ↳ augmenting text w/ icons + pictures
  - ↳ slow down interface
    - avoid sudden state changes
    - reduce / remove unnecessary animations
      - → e.g. animations
    - elim time sensitive actions
- **curb cut** effect: laws + programs designed to benefit vulnerable groups often end up benefiting all of society
  - ↳ e.g. vid closed-captioning
- legal obligations in Canada:
  - ↳ **Accessible Canada Act** (since 2019)
    - applies to gov + federally regulated organizations
    - expected to use **Web Content Accessibility Guidelines (WCAG)**
    - fines up to $250K
  - ↳ **Accessibility for Ontarians w/ Disabilities Act** (since 2005)
    - applies to all Ontario gov websites + public / private entities w/ 50+ employees
    - must adhere to WCAG 2.0
    - fines up to $100K

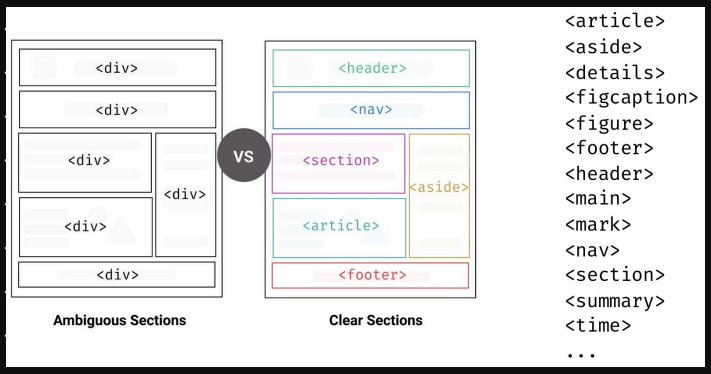
IMPLEMENTING ACCESSIBLE INTERFACES

- browser gens **accessibility tree** from DOM w/ accessibility-related info for most HTML elmts
  - ↳ name, description, role, state
  - ↳



  - ↳ in Accessibility tab in DevTools
- **semantic HTML**: semantic elmts clearly describe content meaning
  - ↳ aka **Plain Old Semantic HTML (POSH)**
  - ↳ don't use + for everything

↳ e.g.



semantic HTML w/Preact components:

↳ avoid `<div>` for component root if semantic elmt possible

- use Fragment instead of `<div>` if not

↳ some components can have diff semantic roles

- e.g. could be in main content + aside
- use `as Element` pattern
  - → e.g.

```tsx
type CalloutProps = {
  as: any;
  children: ComponentChildren;
};

export default function Callout({
  as: Element = "div",
  children,
}: CalloutProps) {
  return <Element className="callout">{children}</Element>;
}


// try changing the `as` prop to aside, blockquote, h1, etc.
// then check the rendered HTML
render(
  <Callout as="aside">as-element content</Callout>,
  document.body
);
```

`skip link`: add link to top of pg so screen readers can skip to content

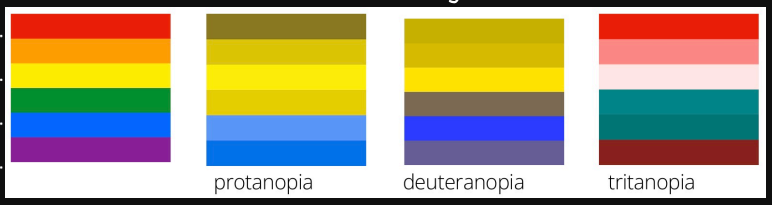
↳ e.g.

```html
<a href="#maincontent" class="skip">Skip to main content</a>
<header>
...
</header>
<section id="maincontent">
```

CSS to hide until focused

```css
.skip {
  position: absolute;
  left: -9999px;
}

.skip:focus {
  position: static;
}
```

- Accessible Rich Internet Applications (ARIA): spec to add semantics to elmts
  - ↳ use ARIA roles only if not possible to use HTML semantic elmt
    - e.g. toolbar, tooltip, feed, math, presentation, note
  - ↳ use ARIA states + props for attrs not supported on HTML elmt
    - e.g. aria-required, aria-checked, aria-disabled
  - ↳ ARIA attrs only change accessibility tree
- types of colour blindness:
  - ↳ dichromacy: 1 type of cone missing
    - protanopia: missing red cones
    - deuteranopia: missing green cones
    - tritanopia: missing blue cones
  - ↳ monochromacity: 2 or 3 types of cones missing
  - ↳



- colour perception: human ability to discriminate colours depends on context
  - ↳ harder to tell 2 colours apart when:
    - colours are pale
    - obj too small / thin
    - colour patches far apart
- colour contrast: ratio of perceived luminance for 2 colours
  - ↳ e.g.

1.0 (on white)	8.6 (blue on white)
1.4 (green on white)	21.0 (black on white)
4.0 (red on white)	

  - ↳ WCAG guidelines:
    - min (lvl AA) is at least 4.5
    - enhanced (lvl AAA) is at least 7.0
- basic ally testing methods:
  - ↳ disconnect mouse + try to use app
    - tab, shift + tab to focus elmt
    - enter to activate elmt
    - arrow keys
  - ↳ test w/ screenreader
  - ↳ ally linters / checkers

COMPUTER VISION

- p5.js is JS lib for creative coding w/ focus on accessible + inclusive coding
  - ↳ simplified + enhanced HTML Canvas API
- p5.js architecture:
  - ↳ runs in global / instance mode
    - global: all props + methods are attached to window
    - instance: all props + methods are bound to p5 obj
  - ↳ implied callback fcns
    - setup() runs when program starts
    - draw() executed 60 times / sec by default
  - ↳ implied event fcns
    - e.g. keyPressed() when key is pressed
  - ↳ global vars
    - e.g. mouseX, mouseY for curr mouse pos
    - e.g. key for last key pressed

> NOTE
> p5.js is imperative

- computer vision: broad class of algos that allow computers to make intelligent assertions abt digital imgs + vids
  - ↳ for human computer interaction, computer is controlled using vid cam
  - ↳ e.g. simple computer vision algo

```
for each video frame:
    identify a "special" pixel
    based on some criteria
    use that pixel's location
    to control the computer
```