Subtyping and Events

Lecture 19 - COMP110 - Spring 2018

HACK110
THIS Friday 7pm to 7am – Sitterson Hall

- 75 fellow 110ers have RSVPed… you should come, too!
- Sessions on: Game Dev, Web Dev, iOS App Dev, what's next in CS
- Free food through the evening, Sticker Badges of Honor for Demoers, 110 T-Shirts
- Hack on a *solo* or *pair* project of your own design
  - We'll e-mail out final project guidelines to everyone RSVPed
- If you have not RSVPed and want to come, RSVP! We will e-mail everyone later this week with final details:
  - https://goo.gl/forms/N1OOVfCMk1Y014bG3

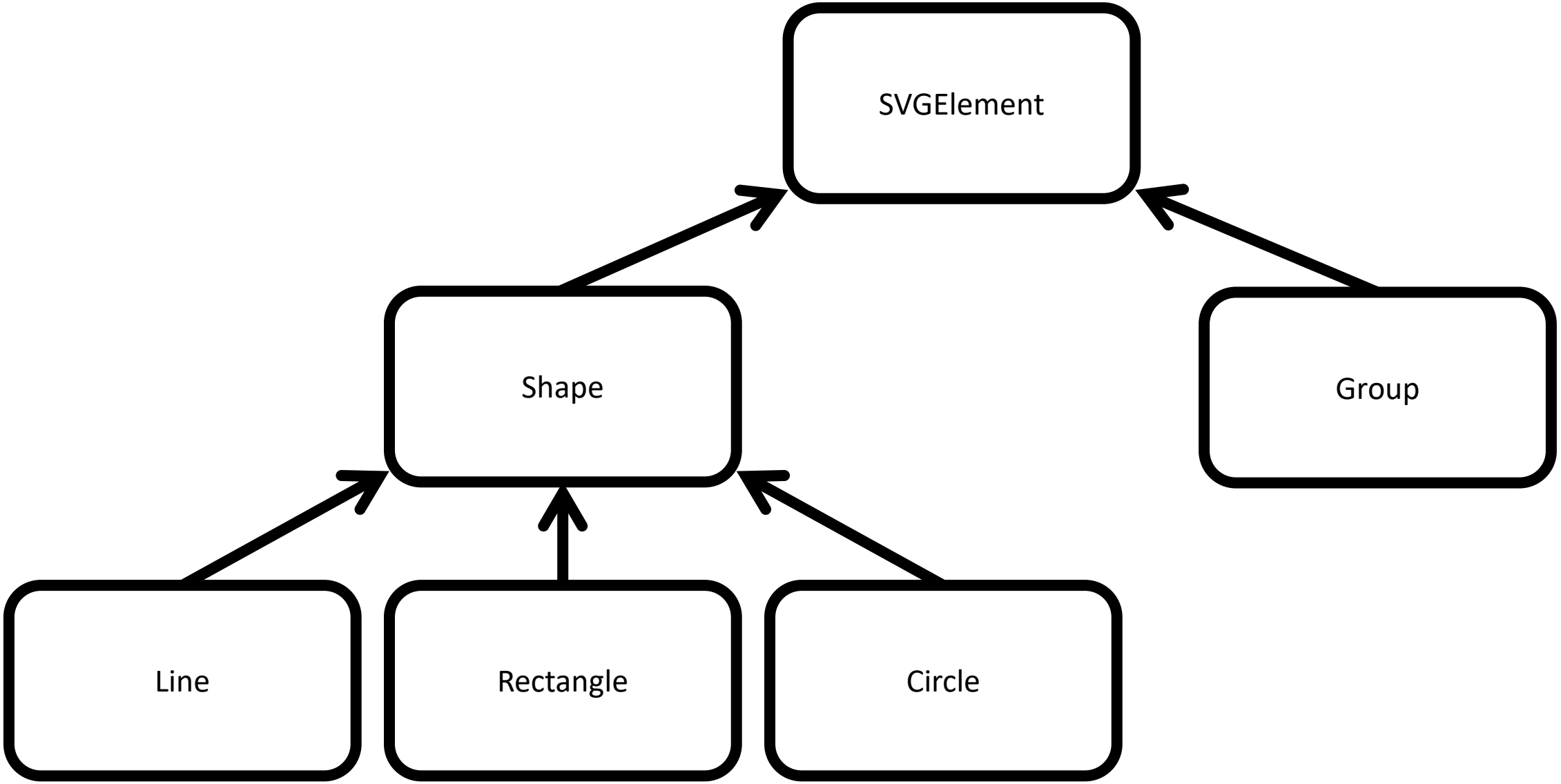
Assignments Out

- **WS4 - Arrays and while Loops – Due tonight at 11:59pm**

- **PS5 - Algo Rhythms**
  - **Part 1 Written - Due Thursday 4/5 at 11:59pm**

  - **Part 2 Program - Due Sunday 4/8 at 11:59pm – Grader up soon, but you can write your own tests in the mean time to convince yourself of correctness!**

Graphics Library Type Hierarchy



A class can *extend* another class to establish a more specific type.

- Classes can form a class hierarchy
  - Like a taxonomy in biology

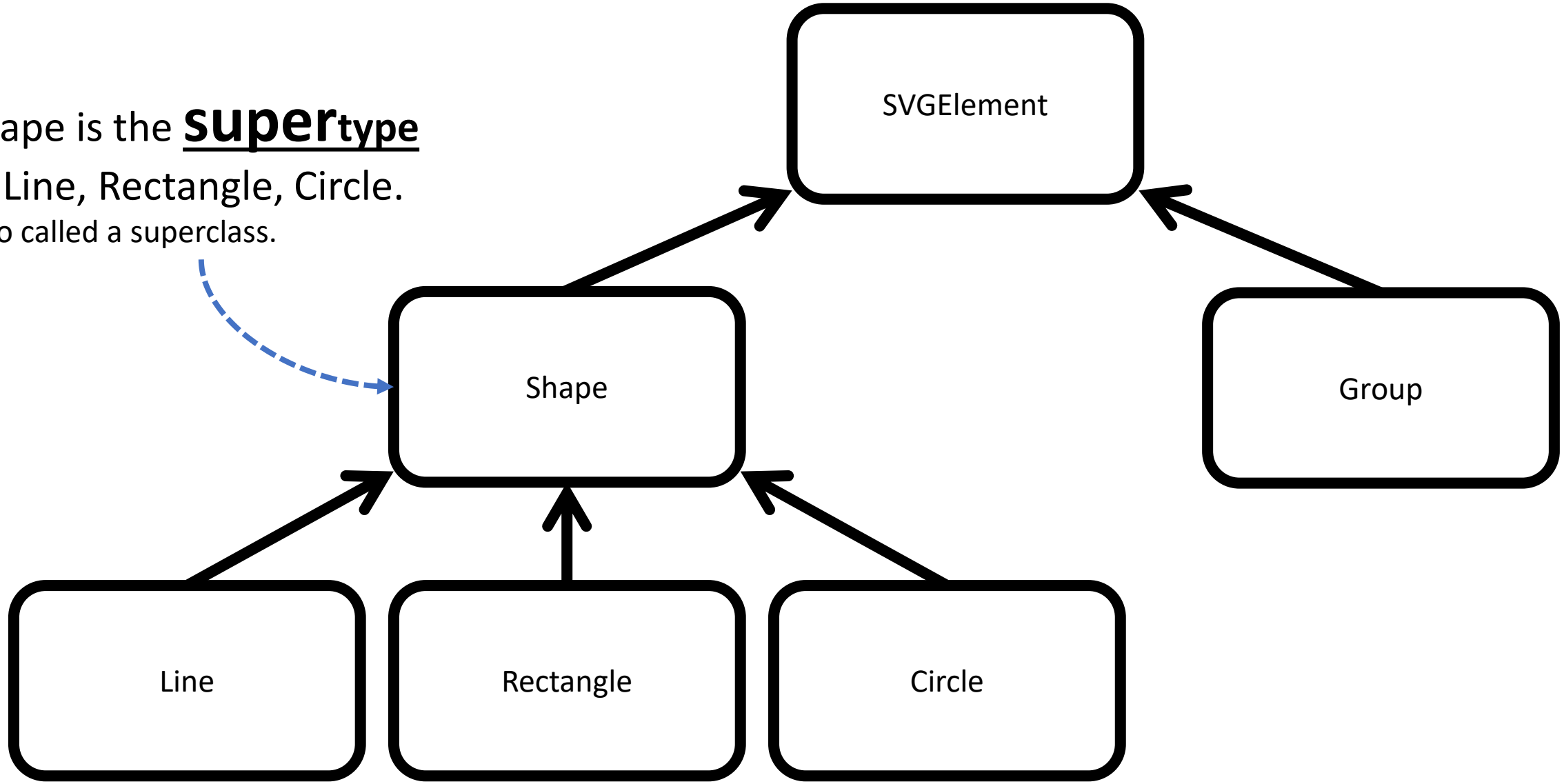
- When a class (A) is extended by another class (B)...
  - an object of type B is an object of type A
  - an object of type A is not necessarily of type B

- **`class Square extends Rectangle`**
  - "A Square is a rectangle, but a Rectangle is not necessarily a Square."

Terminology

Shape is the **supertype** of Line, Rectangle, Circle.

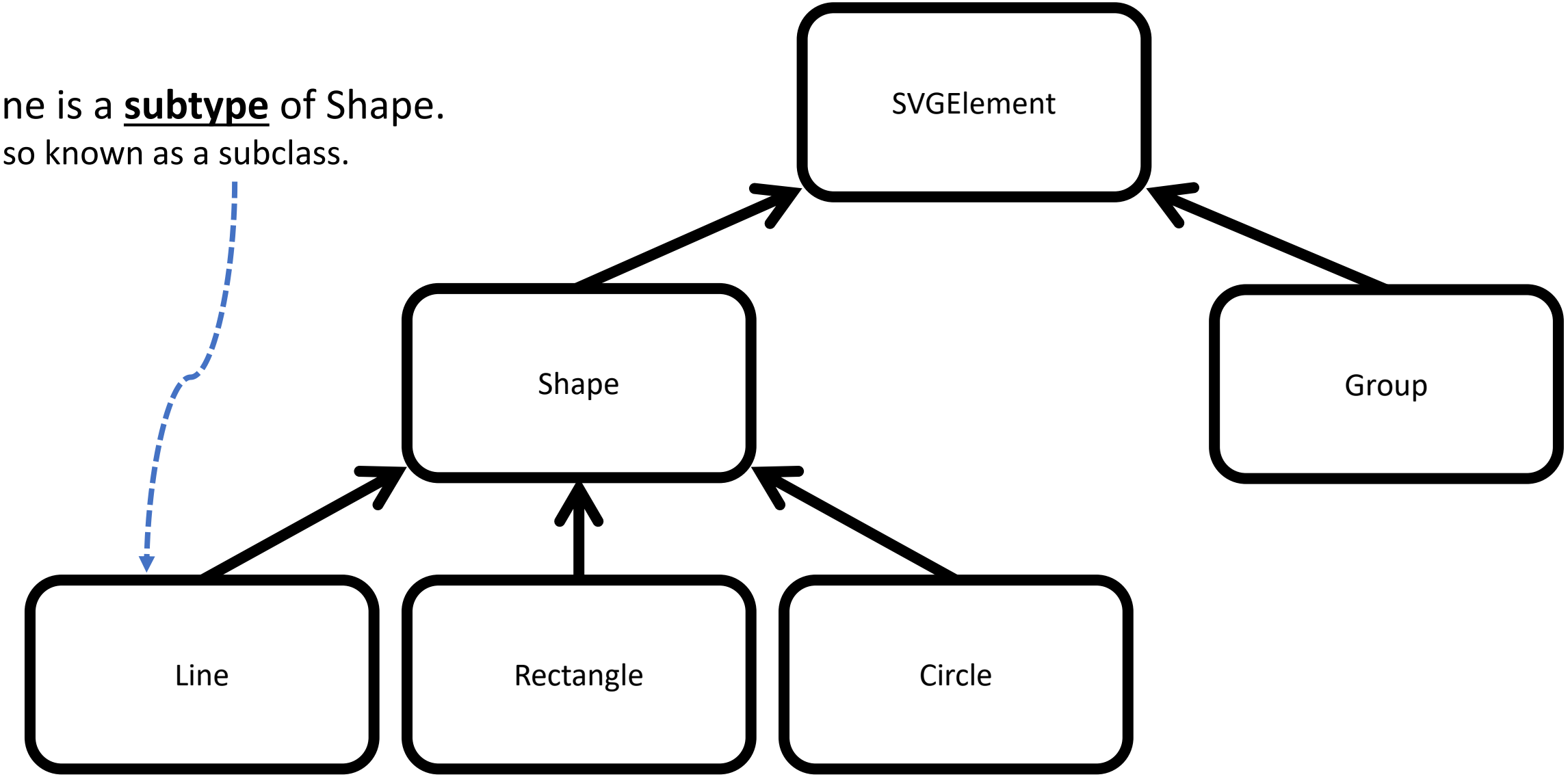
Also called a superclass.



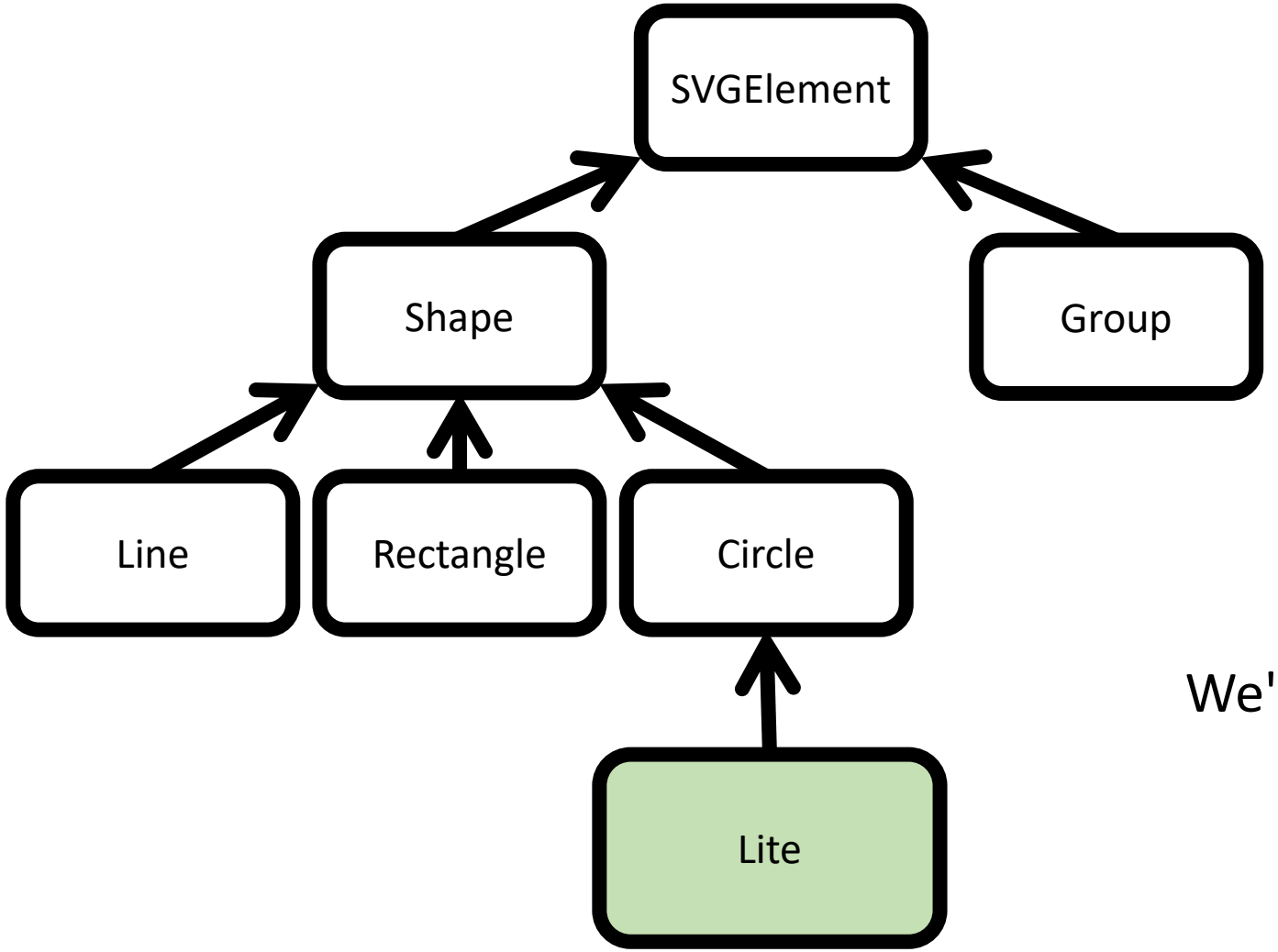
Terminology

Line is a **subtype** of Shape.

Also known as a subclass.



We can extend built-in classes.



We're going to *extend* the hierarchy today!

Defining a Subclass

```
class <Subclass> extends <Superclass> {

}
```

- When defining a new class, you can extend an existing class by using the keyword **`extends`** followed by the superclass' name

- When a class is extended, the subclass automatically has the properties, constructor, and methods of the superclass.

- When extending a class, you can:

1. Introduce additional, new properties or methods to the subclass.
   - This works no differently than adding them to a standalone class!

2. *Override* existing methods or constructors of the superclass.
   - There are a few nuances in doing this that we'll explore today.

- You cannot remove properties, constructor, or methods.

Follow-along: Declare a Lite subclass of Circle

- In lec19 / 00-subtypes-script.ts

- We're going to introduce the notion of a "Lite"
  - A Lite will be a Circle, but with some additional capabilities

- Let's begin by establishing Lite extends Circle *without* any new capabilities:

```
export class Lite extends Circle {

}
```

Hands-on #1

- In the **main** function of 00-subtypes-script.ts declare a new named **lite**

1. Initialize it to a **`new Lite`** object - Hint: it has **`Circle`**'s constructor!
    1. r: 30
    2. cx: 100
    3. cy: 100
2. Add the lite to the **`scene`** Group

- Save, refresh, and when you click a lite should appear. Check-in on PollEv!

```
// TODO: Add a Lite
let lite = new Lite(30, 100, 100);
scene.add(lite);
```

Adding Methods to a Subclass

- Methods can be added to a subclass the same as to a regular class.

- In doing so, *only* objects of the subclass' type have the method
  - You are not changing the superclass in any way.

- Let's add a method to our Lite that sets its fill to a random color. Then, before adding it to our, let's call it!

```
class Lite extends Circle {

    fillRandom = (): void => {
        this.fill = new Color(Math.random(), Math.random(), Math.random());
    }

}
```

Subclass Constructor Rule:
*A subclass constructor must call **`super`** Constructor First*

- When defining a subclass constructor, there is something you must *always* do: call the superclass' constructor first.

- Why? The superclass' constructor is responsible for initializing the properties *it* defines.

- How? Inside of a subclass constructor, a special keyword named **`super`** refers to the superclass' constructor.
  - You must call it with the correct arguments.

Adding a constructor to Lite

- Suppose we wanted all Lites to have a random starting fill color.
- We can ensure this by declaring a constructor for Lite (below).
- Notice the first thing we needed to do was call the super class' (Circle's) constructor!

```
class Lite extends Circle {

    constructor(r: number, cx: number, cy: number) {
        super(r, cx, cy);
        this.fillRandom();
    }
    // Elided
}
```

How can we change colors when the Lite is clicked?

- Shapes have ***"event handlers"*** that get called for certain "events" like mouse clicks!

- An event handler is *just a **type** of function* that is registered to be called when a specific event occurs.
  - Events: click, double click, mouse move, key down, key up, and so on.

```
interface MouseEventHandler {
   (event: MouseEvent): void;
}
```

- The **MouseEvent** class is standard in web browsers.
  - Each has a list of properties describing the event ("what x coordinate did the click occur at?")
  - You do not need to memorize this, it can be searched for as needed

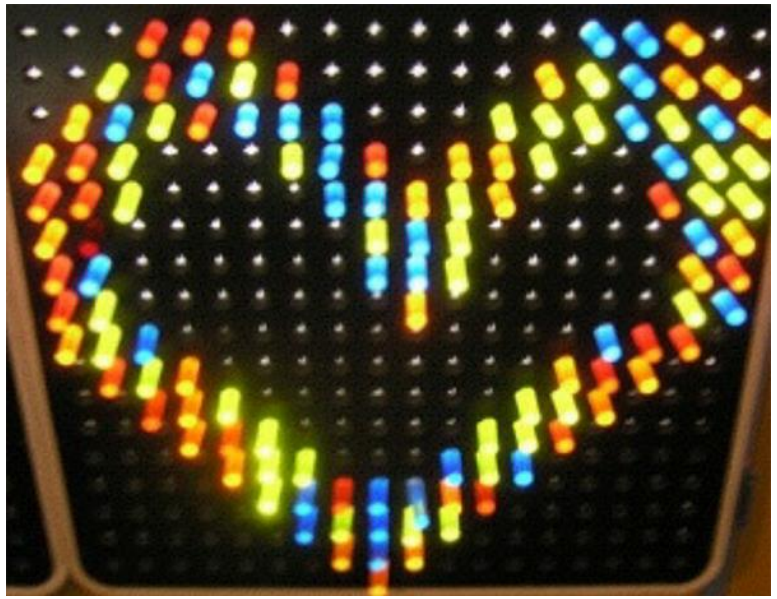
Follow-along: Adding an **`onclick`** handler

- We will override the built-in `onclick` Mouse Event Handler and add our own.

```
export class Lite extends Circle {

    constructor(r: number, cx: number, cy: number) {
        super(r, cx, cy);
        this.fill = new Color(0.3, 0.3, 0.3);
    }

    onclick = (e: MouseEvent): void => {
        this.fillRandom();
    }

    /** Elided **/
}
```

- This special method of Shapes is called each time you click on the Shape

How can we make a "Lite Brite" App?



Aside: A second style of **loop** statement…

- The **while** loop statement is the most flexible and powerful kind of loop at our disposal.
- However, it's easy to accidentally write infinite loops.
- Today we'll learn the **for** loop statement.

#TBT: Writing a **`while`** loop that repeats a specific number of times.

- Repeating a task a specific number of times is a **very** common task in computing.
- You will see this all semester.
- Three keys:
  1) Declare a counter variable and initialize it to 0.
  2) The loops test will check that the counter variable is less than the # of times you want to repeat
  3) **Don't forget!** The last step of the repeat block is incrementing your counter variable.

1

```
let i = 0;

while (i < ____) {

  // Do Something Useful

  i = i + 1;

}
```

2

3

The **for** Loop Statement

- General form:

```
for ( <variable initialization> ; <boolean test> ; <variable modification> ) {

   <repeat block>
}
```

1. Counter variable is initialized

2. Boolean test is evaluated
   True? – **3.** Repeat block is entered and runs.
   4. *Then*, counter variable modified.
   Finally, loop back to step #2.

False? – 5. Skip repeat block and loop is complete.

The **for** Loop Statement

- General form:

```
for ( let i = 0 ; i < 10 ; i++ ) {

   <repeat block>
}
```

1. Counter variable is initialized

2. Boolean test is evaluated
   True? – **3.** Repeat block is entered and runs.
   4. *Then*, counter variable modified.
   Finally, loop back to step #2.

False? – 5. Skip repeat block and loop is complete.

Follow-along: **for** Loop Example

- Open lec19 / 01-for-loops-app.ts

```
print("For Loop Examples");
for (let i = 0; i < 10; i++) {
        print(i);
}
```

Hands-on: Write your own for loop

- At TODO#2 – Write a for loop that will individually print each of the words in the words array, in reverse order (i.e. fox, brown, quick, the).

  If you add additional words to this array, your for loop should still work.

- Check-in when you have a solution: comp110.com/pollunc

What's so great about a **for** `loop`?

- Special syntax for the common while loop pattern using a counter variable
  - *But to the computer, each is exactly the same!*

- For us as human programmers, the **`for`** loop syntax has two benefits:

  1. You are *much* less likely to accidentally write an infinite loop

  2. The counter variable is only defined within the for-loops repeat block
     - Kind of like a function's parameter is only accessible inside of the function body.
     - This means you can have a sequence of for loops that each use, say ***i***, as the counter variable.

- Generally, once the syntax is familiar, for-loops are less human-error prone

Hands-on: Draw a 20x20 grid of lights

- At TODO#1 in 02-lite-brite-script.ts:

1. Write a *nested for* loop
   - Form reminder: `for (let i = 0; i < n; i++) { ... }`
   - Rather than i as a counter variable name, use:
     - **row** for the outer loop
     - **column** for inner loop
   - Each loop's condition should be while its counter is less than ROWS, COLUMNS respectively

2. Inside the innermost loop
   - Declare a Lite variable named lite.
   - Initialize it with:
     - **radius** of RADIUS
     - **cx** of **START_X + DIAMETER * column**,
     - **cy** of **START_Y + DIAMETER * row**
   - Add it to the **scene** Group

3. Save & refresh and you should see a 20x20 grid of lites. **Check-in!**

Project Goal: Update a Lite's state over time

- We'd like our Lite to update and change as time passes

- Let's add an **`update`** method to Lite that just calls **`fillRandom`** for now...

```
update = (): void => {
    this.fillRandom();
}
```

How can we call update() on every Lite?

- Let's store each of our Lite objects in a global array of Lites

```
/* Global Variables */
let lites: Lite[] = [];
```

```
// TODO #1: Write your nested for loop here
for (let row = 0; row < ROWS; row++) {
    for (let column = 0; column < COLUMNS; column++) {
        let lite = new Lite(RADIUS, START_X + column * DIAMETER, START_Y + row * DIAMETER);
        scene.add(lite);
        lites[lites.length] = lite;
    }
}
```

Let's update all Lite objects every second

- First we'll setup a function outside of main called `tick`. It will loop through every Lite in the lites array and call its update method.

```
let tick = (): void => {
    for (let i = 0; i < lites.length; i++) {
        lites[i].update();
    }
};
```

- Then, we'll set an interval of 1 second to call tick

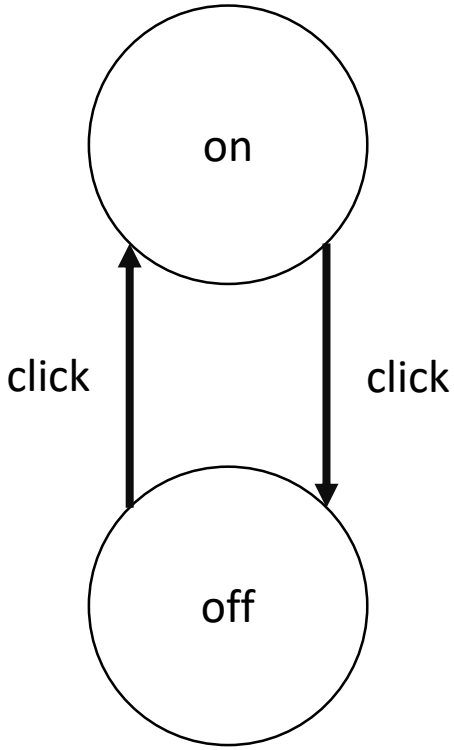
```
// At the end of the the main function:
setInterval(tick, 1000);
```

Project Goal: What if we wanted a Lite to be able to toggle between **on** (lit) and **off** (unlit)?

- How can each Lite keep track of whether it is currently **lit**?

- With a property! Let's add a boolean **on** property to Lite.

```
export class Lite extends Circle {
      on: boolean = false;
      // …
```



Hands-on: Toggling States

1. In the `onclick` Handler, assign to **`this.on`** the *negated* value of **`this.on`**
   - Hint: How do you negate or *not* a boolean value?

2. Change the `update` method to have the following logic:
   - If this.on is true, then call the fillRandom method
   - If this.on is falsle, then set this.fill to be UNLIT

3. Save, refresh, and see if you can toggle your Lites on and off. Check-in when working.

```
onclick = (e: MouseEvent): void => {
    this.on = !this.on;
}
```

```
update = (): void => {
    if (this.on) {
        this.fillRandom();
    } else {
        this.fill = UNLIT;
    }
}
```

Lite Brite's Object-Oriented Programming Takeaways

- Notice each circle is an *instance, or object,* of the Lite *class*

- Each and every Lite object has its own individual properties:
  - on (boolean)
  - fill (Color)
  - etc

- We added *methods* to the Lite class which gave each Lite new capabilities (fillRandom, update, onclick)
  - A lite could call a method *on itself* but calling `this.<method>(…)`

A **Type** tells you ***what*** it can do, ***not how*** *it does it*

- The type of a value tells you how you are allowed to interact with that value in code

- If it is a primitive value type (number, string, boolean)
  - Then you can use primitive operations

- If it is an array of some other type
  - Then you can use the index operator a[<index>]

- If it is a function-type value
  - Then you know what *parameters it requires* and it's *return type*
  - You may not know *what function will actually be called when you call it*
    - *(As shown in the PollEverywhere example. Also fundamental to how filter/map/reduce work.)*

- If it is an object-type value
  - Then you know its class which tells you its properties, constructor, and methods
  - You also know what methods it has, their parameters, and their return type
  - You may not know what method will *actually be called when you call it*