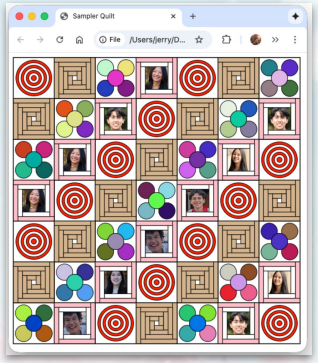
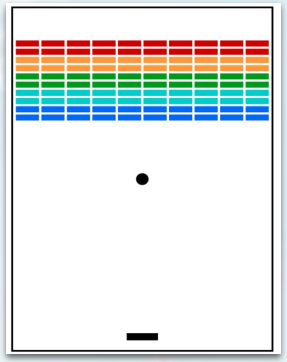
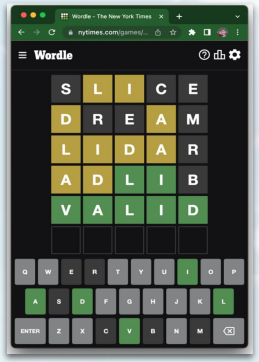
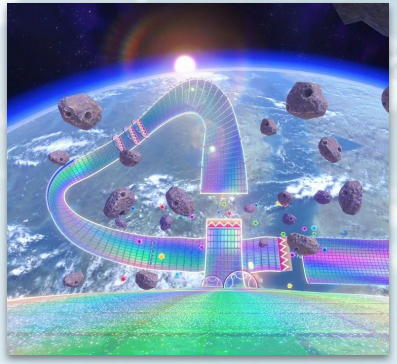
Slides at cs106ax.stanford.edu/lectures.html



Midterm Review

CS106AX Autumn 2025

Slides based on materials by Jerry Cain, Ryan Eberhardt, CS 106 staff

Diego Emilio Padilla, Ben Yan

Stanford | ENGINEERING
Computer Science

💖 Welcome to (almost) Week 6 of CS106AX!

Autumn

Week 6

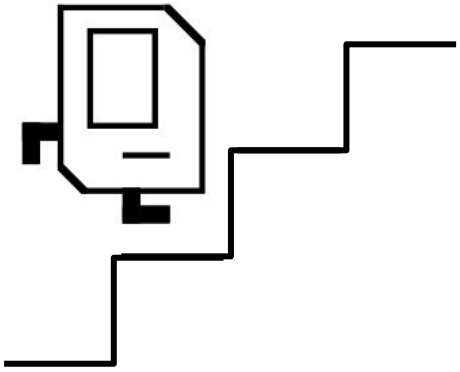
Winter

Hope things are going well!

加油! (add oil, a popular saying)

The Map For Today

1. Quick overview of midterm structure, logistics, what you can expect 📔
2. Recap of course material: **JavaScript (no Python!)**, emphasis on idea over syntax
3. Walk through practice problems and coding exercises together 💖
4. Questions and office hours!



Midterm Logistics

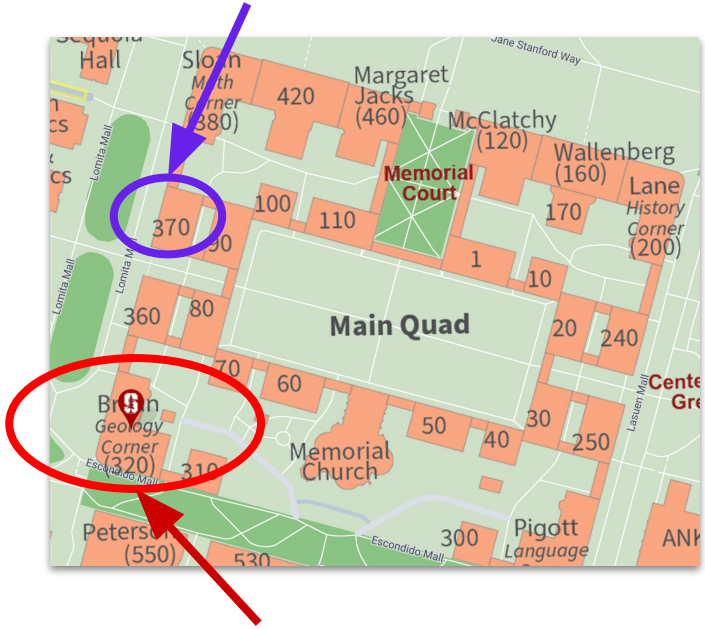
🧭 Place & Time

- ❑ Midterm is **Tuesday, Oct. 28th**, offered at:
  - ❑ **3:30 – 5:30 PM**, Building 320, Room 109
  - ❑ **7:00 – 9:00 PM**, Building 370, Room 370
  - ❑ Can just show up to either time!
- ❑ If you emailed Jerry for an alternative time, you should hear back by tomorrow Monday!

✍️ Mechanics

- ❑ 2 hours long, administered over paper
- ❑ 📚 **Open notes and book**, printed or handwritten, as much as you'd like to bring**; closed computer, no calculator**

🎯 **7 – 9 PM Exam**



🎯 **3:30 – 5:30 PM Exam**

You rock! 

🎯 Midterm Structure

Identical architecture to Practice Midterms, i.e. a star constellation of five problems with topics:

Problem 1	**JavaScript** expressions, statements, methods (short answer, **scoping and trace problems**)
Problem 2 🎨	Using **graphics and animation** (multi-step program, be familiar with mouse event handlers and timers)
Problem 3	**Strings** (write a `function` to solve a string task)
Problem 4	**Arrays** (write a `function` to solve an array task)
Problem 5	Working with **data structures** (write a `function` that handles data stored in JSON / **JS Object Notation** form)

Remember – **this is all stuff you've worked with**, and leveraged beautifully to make programs such as your Breakout! and Wordle games 🔥. **Keep doing what you do.**

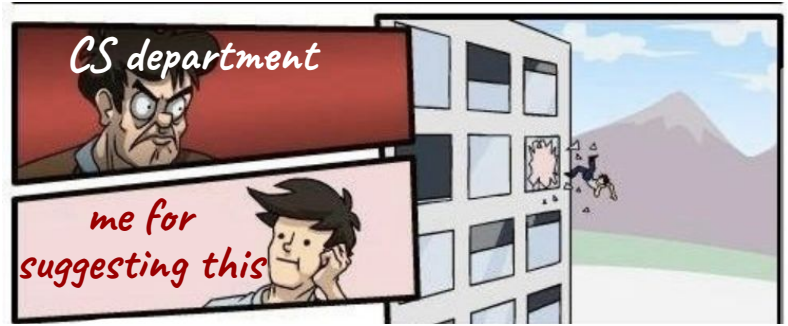
🎯 General Exam Tips and Recs

JavaScript! → ✨ a "vibes-based" language

What I really mean by this: **understand and feel it, <u>don't memorize</u>.**
You just have to practice and keep practicing!

"We don't care about syntax – it doesn't matter!" 🙃

Ok, this isn't entirely true – however …



🎯 General Exam Tips and Recs

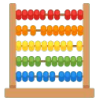
🧮 Broadly, problem-solving ability, logic, and **conceptual understanding** of the course material **is far more important than having perfect syntax**

- ❑ **We won't penalize for minor syntax errors, as long as your intentions are clear**, e.g., misspelling `GRekt`, a missing closing brace in JavaScript *(as long as it's clear whether each line of code is under an if test, while loop, etc.)*
- ❑ Brief commenting is not required, but can help sometimes for partial credit!

⏱️ Try not to panic! *I know – way easier said than done.* But **you can do this!**

- ❑ Don't be afraid to move on + come back re-energized – and **try to attempt every problem**, even if you're not sure how to finish it off.
- ❑ Try not to rely too much on your notes / books / handouts, but it's great if you can **look up things, e.g., method names, effectively when necessary** 📖**.**

✍️ **Practice writing code on paper**, and go in with a strategy (e.g., does it help you to jot down a bit of pseudocode, or sketch your high-level approach?)



JavaScript Expression Reminders

✍️ The first part of the midterm will involve evaluating plain JavaScript expressions, e.g.,

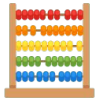
```
2025 > 2020 + 6 || 3 % 3 === 3
```

For each part of the expression, you'll want to be conscious of the type it evaluates to.

- JavaScript obeys the **PEMDAS rule** for arithmetic **order of operations**
  - Parentheses first, then exponents, multiplication / division / remainder operator* (tiebreak, left to right), then addition / subtraction (left to right)

---

* Remainder operator: `n % d` is the remainder when `n` is divided by `d`, and always takes the same sign as `n`. For instance, `6 % 3 ↦ 0, 7 % 3 ↦ 1, -7 % 3 ↦ -1, 4 % 5 ↦ 4`



JavaScript Expression Reminders

✍ The first part of the midterm will involve evaluating plain JavaScript expressions, e.g.,

```
2025 > 2020 + 6 || 3 % 3 === 3
```

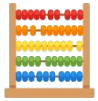
For each part of the expression, you'll want to be conscious of the type it evaluates to.

- JavaScript obeys the **PEMDAS rule** for arithmetic **order of operations**
  - Parentheses first, then exponents, multiplication / division / remainder operator* (tiebreak, left to right), then addition / subtraction (left to right)
- As a general rule, evaluate **arithmetic operators** first (`+`, `-`, `/`, `*`), then **comparison ones** (`<`, `<=`, `>=`, `>`), then **logical / boolean operators** (`&&`, `||`)*

```
2025 > 2020 + 6 || 3 % 3 === 3
2025 > 2026 || 0 === 3
false || false
false
```

---

* Special case: Short-circuiting! Logical evaluations can **end early if result already decided**, e.g, `True || ...`
* Very niche, but logical unary NOT (`!`) is an exception, being evaluated before arithmetic operators



JavaScript Expression Reminders

✍ The first part of the midterm will involve evaluating plain JavaScript expressions, e.g.,

```
2025 > 2020 + 6 || 3 % 3 === 3
```

For each part of the expression, you'll want to be conscious of the type it evaluates to.

- JavaScript obeys the **PEMDAS rule** for arithmetic **order of operations**
  - Parentheses first, then exponents, multiplication / division / remainder operator* (tiebreak, left to right), then addition / subtraction (left to right)
- As a general rule, evaluate **arithmetic operators** first (`+`, `-`, `/`, `*`), then **comparison ones** (`<`, `<=`, `>=`, `>`), then **logical / boolean operators** (`&&`, `||`) *

```
2025 > 2020 + 6 || 3 % 3 === 3
2025 > 2026 || 0 === 3
false || false
false
```

---

* Special case: Short-circuiting! Logical evaluations can **end early if result already decided**, e.g, `True || ...`
* Very niche, but logical unary NOT (`!`) is an exception, being evaluated before arithmetic operators

🎯 JavaScript Expression: Practice!

```
2 + 25 % (4 + 1) * 10 + 3
```

_____

```
"BTS" <= "EXO" && "akb48" === "AKB48"
```

_____

```
"BTS" <= "EXO" || 10 / 0 > 3.14
```

_____

---

* **String comparisons:** A string is considered **"greater"** than another string **if it appears later in a dictionary,** e.g., `"zany" > "azure"`, `"artsy" > "art"` 🎨. A few nuances—number characters before letters, uppercase before lowercase. Really, it depends on ASCII/Unicode ordering!

🎯 JavaScript Expression: Practice!

```
2 + 25 % (4 + 1) * 10 + 3
```

5

2 + 25 % 5 * 10 + 3 → 2 + 0 * 10 + 3 → 2 + 3

```
"BTS" <= "EXO" && "akb48" === "AKB48"
```

false ❌

true && false → false

```
"BTS" <= "EXO" || 10 / 0 > 3.14
```

true ✅

true || 10 / 0 > 3.14 → true

short-circuit evaluation with OR (||), so 2nd operand isn't evaluated (otherwise a crash)

---

* **String comparisons:** A string is considered **"greater"** than another string **if it appears later in a dictionary,** e.g., `"zany" > "azure"`, `"artsy" > "art"` 🎨. A few nuances—number characters before letters, uppercase before lowercase. Really, it depends on ASCII/Unicode ordering!

Function Calls

When tracing functions, take note of **how many / what** parameters it takes, if any, and **if it returns something**, which should be stored inside a variable **where it was called.**

```
function functionName(param1, param2, …){
   list of cool statements in function body
}
```

```
let result = functionName(var1,var2,...); //in other function
```

Function Calls

When tracing functions, take note of **how many / what** parameters it takes, if any, and **if it returns something**, which should be stored inside a variable **where it was called.**

```
function functionName(param1, param2, …){
   list of cool statements in function body
}
```

```
let result = functionName(var1,var2,...); //in other function
```

✍ **Short Example Trace:** What happens when you call **`runBlackJack()`?**

```
function runBlackjack(){
  let card1 = 10, card2 = 11;
  let sum = addNumbers(card1, card2);
  console.log(sum);
}
function addNumbers(num1, num2){
   return num1 + num2;
}
```

Function Calls

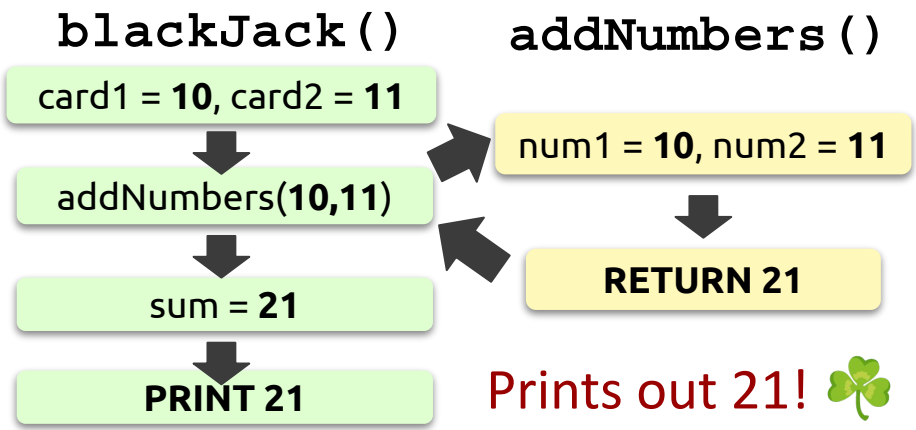
When tracing functions, take note of **how many / what** parameters it takes, if any, and **if it returns something**, which should be stored inside a variable **where it was called.**

```
function functionName(param1, param2, …){
   list of cool statements in function body
}
```

```
let result = functionName(var1,var2,...); //in other function
```

✍ **Short Example Trace:** What happens when you call `runBlackJack()`?

```
function runBlackjack(){
  let card1 = 10, card2 = 11;
  let sum = addNumbers(card1, card2);
  console.log(sum);
}
function addNumbers(num1, num2){
   return num1 + num2;
}
```



JavaScript: Scope Rules

Variables live inside the **block, or pair of braces, in which they're declared.**

Scope of each variable

floors

num

floorsAbove

numHouses

floors

```
function Crothers(){
    let floors = 4;
    for (let num = 0; num < floors; num++){
        console.log("Welcome to floor " + num + "!");
        let floorsAbove = num – floors – 1;
    }
    console.log(floorsAbove);  // Error! ⚠️
}

function Wilbur(){
  let numHouses = 8;
  function Okada(){
    let floors = 3;  //  Okada's version, not Crothers's
    console.log("One of  " + numHouses + "great houses!");
  }
}
```

Nested functions inherit the variables of their parent functions!

⏳ Function: Returning in Different Places

A **`return`** statement **terminates execution of the current function** immediately. Control is given back to the calling function, to which a return value may be sent 📨.

```
function avgUnitsPerClass(numUnits, numClasses){
    if (numClasses === 0){
        return 0;    // can stop tracing here if 0 classes! like summer vacation
    }
    return  numUnits / numClasses;  // only happens if non-zero classes!
    return "so many 1-unit wonders 😊";  // never gets to this line!
}
```

🤔 Sometimes, when writing functions, **we can use early returns to neaten logic**:

- ❏ These are called *guard clauses*, intended to handle special cases upfront, e.g., **`width < 0,`** if the side lengths don't form a valid triangle, like **`1,1,7`**
- ❏ If our core logic is solid, but can crash with an extreme or invalid input like dividing by 0, we can take care of it with **`if () {return … }`**, like the above.

⏳ Function: Parameter Passing Notes

Parameters of immutable types (**`Number, String, boolean`**) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` **are passed by 👉 reference / pointer**, so we can modify the same underlying data in memory*.

* However, reassigning the passed-in object reference wouldn't change the original object, since that'll simply make the local reference point at a new object instead.

⏳ Function: Parameter Passing Notes

Parameters of immutable types (**`Number, String, boolean`**) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` **are passed by 👉 reference / pointer**, so we can modify the same underlying data in memory*.

```
function BensLife(){
   let finances = {savings: 0, checking: 1000};
   let EVGR_RENT = 2000;
   console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
   finances.checking –= rent;
   rent *= 2;  😈😈
   return finances.checking >= 0;
}
```

What's printed by a call to BensLife()?

* However, reassigning the passed-in object reference, e.g., **`finances = {savings:...}`**, wouldn't change the original object, since that'll simply make the local reference point at a new object instead.

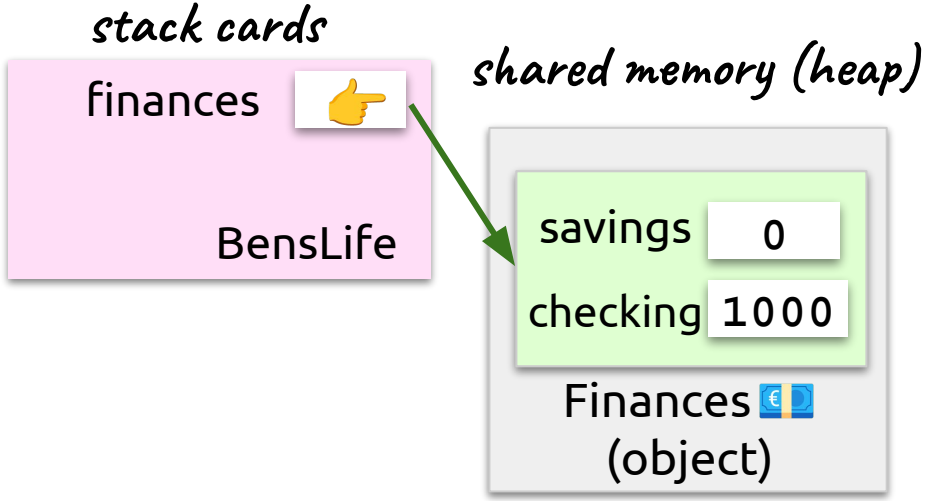
⏳ Function: Parameter Passing Notes

Parameters of immutable types (`Number, String, boolean`) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` are passed by 👉 reference / pointer, so we can modify the same underlying data in memory*.

```
function BensLife(){
  let finances = {savings: 0, checking: 1000};
  let EVGR_RENT = 2000;
  console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
  finances.checking -= rent;
  rent *= 2;
  return finances.checking >= 0;
}
```



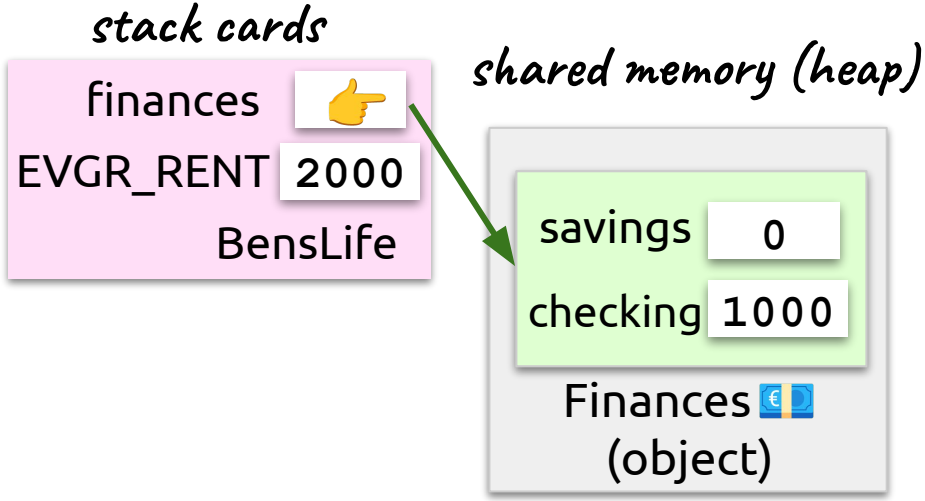
⏳ Function: Parameter Passing Notes

Parameters of immutable types (`Number, String, boolean`) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` are passed by 👉 reference / pointer, so we can modify the same underlying data in memory*.

```
function BensLife(){
    let finances = {savings: 0, checking: 1000};
    let EVGR_RENT = 2000;
    console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
    finances.checking -= rent;
    rent *= 2;
    return finances.checking >= 0;
}
```



⏳ Function: Parameter Passing Notes

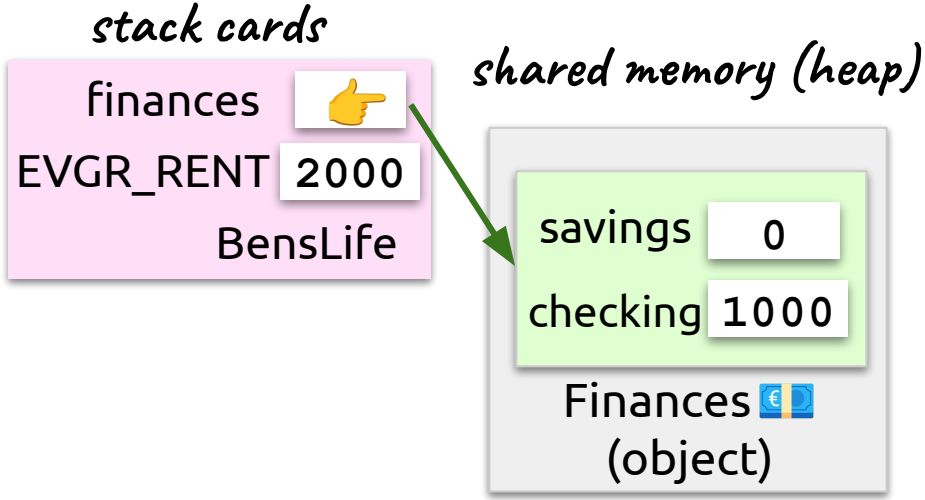
Parameters of immutable types (`Number, String, boolean`) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` are passed by 👉 reference / pointer, so we can modify the same underlying data in memory*.

```
function BensLife(){
    let finances = {savings: 0, checking: 1000};
    let EVGR_RENT = 2000;
    console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
    finances.checking -= rent;
    rent *= 2;
    return finances.checking >= 0;
}
```

👉 *object* **2000**



⏳ Function: Parameter Passing Notes

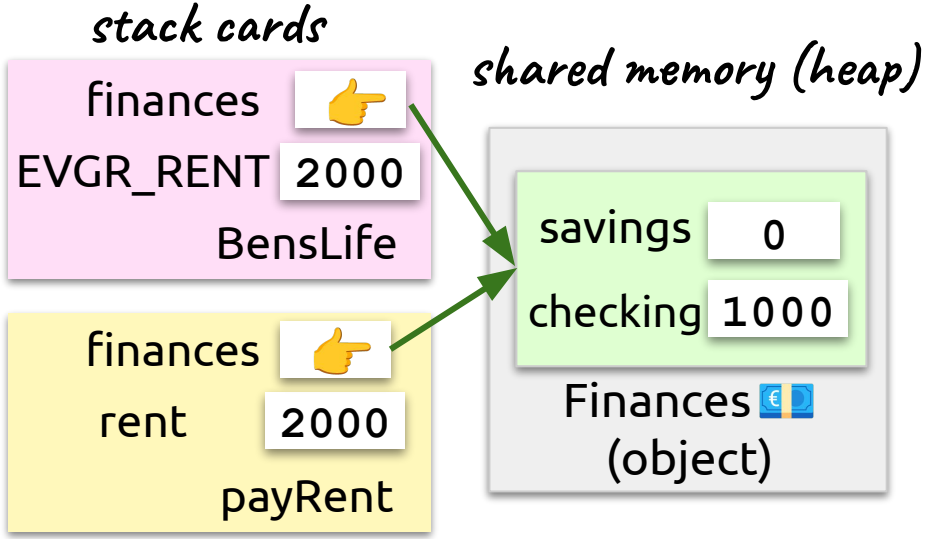
Parameters of immutable types (**`Number, String, boolean`**) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` **are passed by 👉 reference / pointer**, so we can modify the same underlying data in memory*.

```
function BensLife(){
    let finances = {savings: 0, checking: 1000};
    let EVGR_RENT = 2000;
    console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
    finances.checking -= rent;
    rent *= 2;
    return finances.checking >= 0;
}
```

👉 *object* **2000**



📣 *BensLife() calls payRent()!*

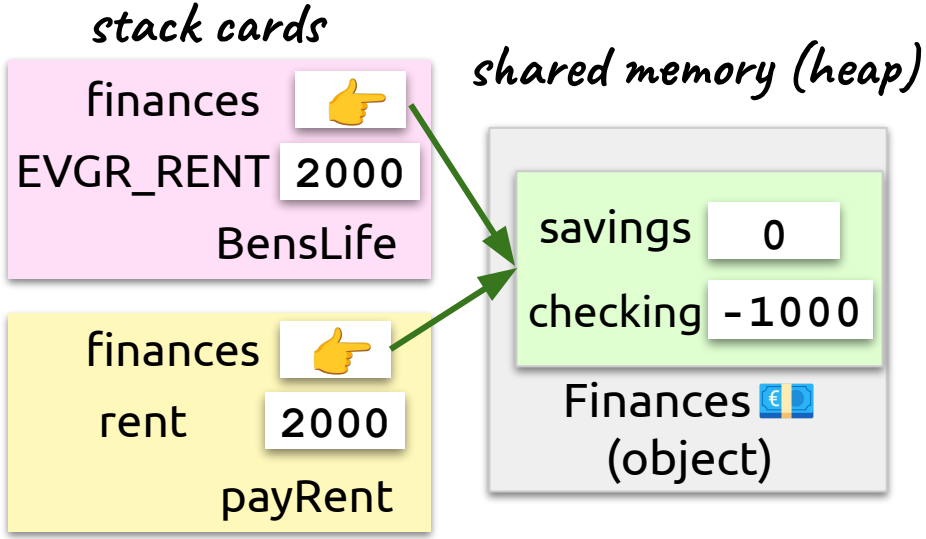
⏳ Function: Parameter Passing Notes

Parameters of immutable types (`Number, String, boolean`) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` are passed by 👉 reference / pointer, so we can modify the same underlying data in memory*.

```
function BensLife(){
    let finances = {savings: 0, checking: 1000};
    let EVGR_RENT = 2000;
    console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
    finances.checking -= rent;
    rent *= 2;
    return finances.checking >= 0;
}
```



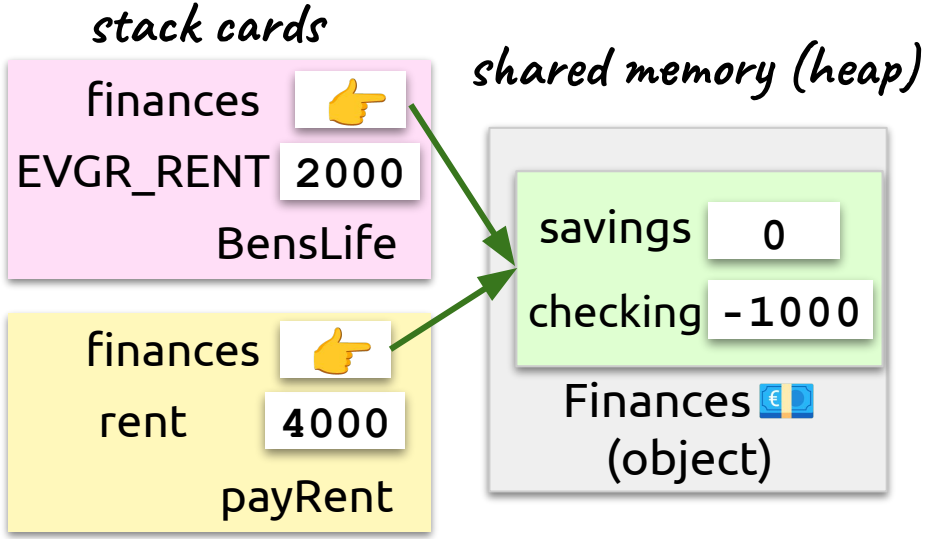
⏳ Function: Parameter Passing Notes

Parameters of immutable types (`Number, String, boolean`) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` are passed by 👉 reference / pointer, so we can modify the same underlying data in memory*.

```
function BensLife(){
    let finances = {savings: 0, checking: 1000};
    let EVGR_RENT = 2000;
    console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
    finances.checking -= rent;
    rent *= 2; 😈😈
    return finances.checking >= 0;
}
// my actual EVGR_RENT doesn't change, omg 🙏
```



⏳ Function: Parameter Passing Notes

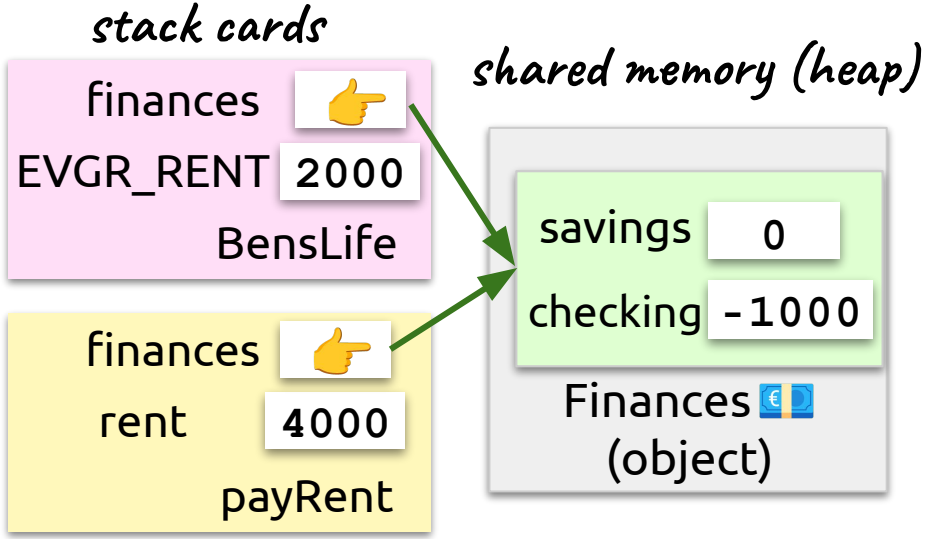
Parameters of immutable types (`Number, String, boolean`) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` are passed by 👉 reference / pointer, so we can modify the same underlying data in memory*.

```
function BensLife(){
    let finances = {savings: 0, checking: 1000};
    let EVGR_RENT = 2000;
    console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
    finances.checking -= rent;
    rent *= 2;
    return finances.checking >= 0;
}
```

false (boolean)



Function: Parameter Passing Notes

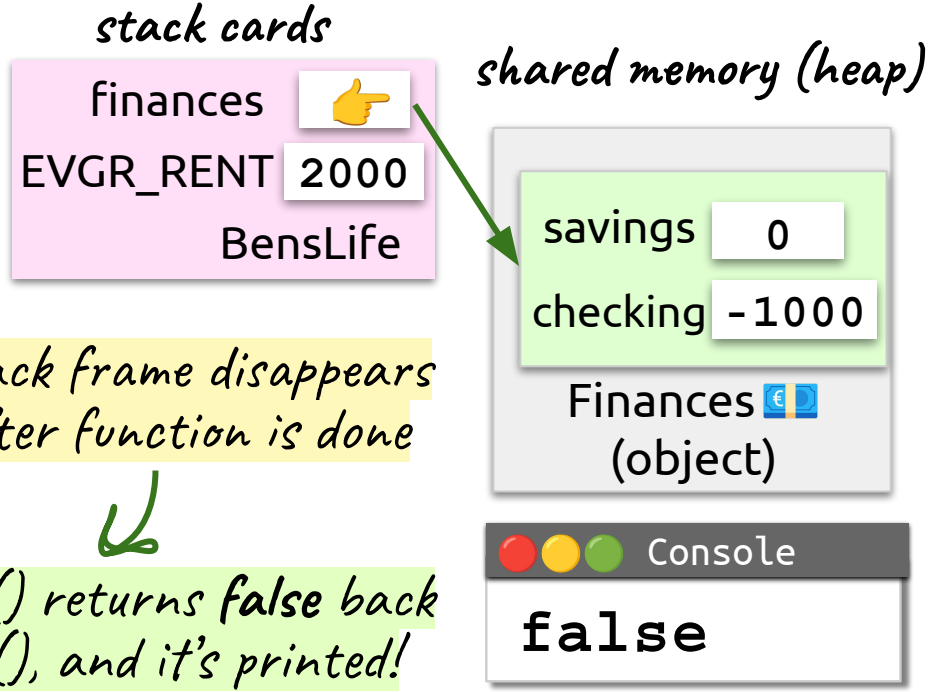
Parameters of immutable types (**`Number, String, boolean`**) get copied when passed. Modifying them won't affect original.

`Array, object, GObjects` **are passed by 👉 reference / pointer**, so we can modify the same underlying data in memory*.

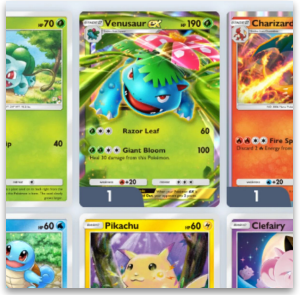
```
function BensLife(){
    let finances = {savings: 0, checking: 1000};
    let EVGR_RENT = 2000;
    console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
    finances.checking -= rent;
    rent *= 2;
    return finances.checking >= 0;
}
```

false (boolean)



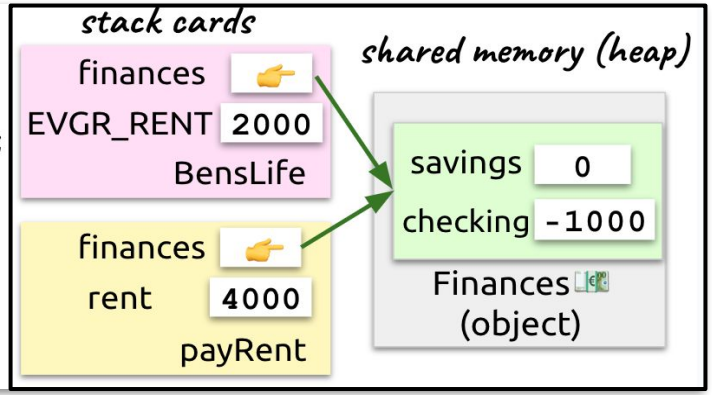
Tracing Tips: Stack Cards



You don't have to rigidly follow this if a different approach works better for you, but this is one way I approach function tracing!

```
function BensLife(){
  let finances = {savings: 0, checking: 1000};
  let EVGR_RENT = 2000;
  console.log(payRent(finances, EVGR_RENT));
}

function payRent(finances, rent){
  finances.checking -= rent;
  rent *= 2;  😈😈
  return finances.checking >= 0;
}
```



Dictionary

stack cards (noun)

For each function, a record table of each variable's current value *(includes parameters!)*

- ❑ Ofc **my handwriting is a lot rougher**, and I'll **cross out old variable values** when updated.
- ❑ For ~~vibes~~ memory-keeping, I often also annotate the code with computed values as I go
- ❑ **Stack cards** can help with making scope clear, and **retrieving variable values quickly**

🔁 For loops, I recommend tracing a few iterations, **and seeing if you can find a pattern**, e.g., if each iteration is just getting the current last digit. **If not, you may have to trace all** 😢.

🖌️ JavaScript Tracing Problems Practice!

Strings in JavaScript

```
let str = "hello";
```

h	e	l	l	o
0	1	2	3	4

Strings are 0-indexed!

Sequence of characters (a single letter, number, symbol i.e. ASCII table) **in order**, enclosed by double quotes

✍️ Methods: What we can do with a string?

`str.length`

Returns length of string

`str.charAt(i)`, or simply `str[i]`

Returns character of string at **index i**

`let substr = str.substring(start, end)`

Extracts and returns substring between index start (inclusive) and end (exclusive), e.g., 🌎 `"earth".substring(0,3)` → `"ear"`

✅ *As a quick check, the substring should be (**end – start**) chars long*

`str.includes(substr)`

true if `str` includes `substr` as a substring (can be 1 char), else **false**

e.g., ✅ `"earth".includes("art")`

`str.indexOf(substr, start)`

Returns index in `str` where first instance of `substr` begins, or –1 if not found.

`start`: Optional, index to start searching

Swankier String Methods

`String.fromCharCode(`*`code`*`)`

Returns the one-character string whose Unicode value is *code*.

***`str`*`.charCodeAt(`*`index`*`)`**

Returns the Unicode value of the character at the specific index.

***`str`*`.toLowerCase()`**

Returns a **copy** of the string converted to lower case.

***`str`*`.toUpperCase()`**

Returns a **copy** of the string converted to upper case.

***`str`*`.startsWith(`*`prefix`*`)`**

Returns **`true`** if the string starts with `prefix`, **`false`** otherwise.

***`str`*`.endsWith(`*`suffix`*`)`**

Returns **`true`** if the string ends with `suffix`, **`false`** otherwise.

***`str`*`.trim()`**

Returns a **copy** of the string with leading and trailing spaces removed.

Char	ASCII Code
"A"	65
"B"	66
...	…
"Z"	90
"a"	97
...	…
"z"	122

In **`JavaScript`**, **all string methods return a new value**, which can be assigned to a variable.

A **reference guide** to some of them!

🔠 String Concatenation

A fancy term for **combining strings end-to-end** with no intervening characters.

- ❏ Uses the ➕ operator, e.g., `"gravity" + "wave" => "gravitywave"`
- ❏ A string and number can be concatenated, `"co" + 24 => "co24"` 🎓
- ❏ When concatenating multiple values in a row, note **order of operations** rules, keep track of type (string or number), and **evaluate two at a time**

```
console.log("messier" + 7 + 80);
```

`=> "messier780"`

```
// "messier" + 7 is "messier7" (string)
// "messier7" + 80 is "messier780" (string)
```

```
console.log(7 + 80 + "messier");
```

`=> "87messier"`

```
// 7 + 80 is 87 (number)
// 87 + "messier" is "87messier" (string)
```

😮 *Different result even though the same 3 values are being added together!*

Common Strings Pattern

Helpful pattern with string functions: given a string, iterate through its characters and build up a **new string** (since strings are **immutable!** can't be changed*)

Awesome String Program 📘

```
let oldStr = some string ...
let newStr = "";
for (int i = 0; i < oldStr.length; i++){
    // build up newStr
}
```

```
/* alternative syntax, directly over chars in forward order
for (let char of oldStr){ ...
*/
```

*String methods such as concatenation, substring, etc. actually create entirely new string objects that we assign to an existing or new variable. String theory! 🧶

🎯 Pattern Example from Lecture

Write a function `removeDoubledLetters` that takes a string and returns a new string with all doubled consecutive letters in the string replaced by a single letter.

```
removeDoubledLetters("bookkeeper") => "bokeper"
removeDoubledLetters("zzzz") => "z"
```

🎯 Pattern Example from Lecture

Write a function `removeDoubledLetters` that takes a string and returns a new string with all doubled consecutive letters in the string replaced by a single letter.

```
removeDoubledLetters("bookkeeper") => "bokeper"
removeDoubledLetters("zzzz") => "z"
```

❓ For this problem, **questions I might ask myself:**

➔ What do I do with each character of the original string?

If it isn't the same as the previous character, add it to the result string

🎯 Pattern Example from Lecture

Write a function `removeDoubledLetters` that takes a string and returns a new string with all doubled consecutive letters in the string replaced by a single letter.

```
removeDoubledLetters("bookkeeper") => "bokeper"
removeDoubledLetters("zzzz") => "z"
```

❓ For this problem, **questions I might ask myself:**

➔ What do I do with each character of the original string?

If it isn't the same as the previous character, add it to the result string

➔ How do I get the previous character?

Go to the index before my current one

🎯 Pattern Example from Lecture

Write a function `removeDoubledLetters` that takes a string and returns a new string with all doubled consecutive letters in the string replaced by a single letter.

```
removeDoubledLetters("bookkeeper") => "bokeper"
removeDoubledLetters("zzzz") => "z"
```

❓ For this problem, **questions I might ask myself:**

➔ What do I do with each character of the original string?

If it isn't the same as the previous character, add it to the result string

➔ How do I get the previous character?

Go to the index before my current one

➔ Is there anything else I need to think about? Any special index cases?

The character at index 0: doesn't have a prior character, but still needs to go into the result string!

🎯 Pattern Example from Lecture

Write a function `removeDoubledLetters` that takes a string and returns a new string with all doubled consecutive letters in the string replaced by a single letter.

```
removeDoubledLetters("bookkeeper") => "bokeper"
removeDoubledLetters("zzzz") => "z"
```

```
function removeDoubledLetters(str) {



}
```

🎯 Pattern Example from Lecture

Write a function `removeDoubledLetters` that takes a string and returns a new string with all doubled consecutive letters in the string replaced by a single letter.

```
removeDoubledLetters("bookkeeper") => "bokeper"
removeDoubledLetters("zzzz") => "z"
```

```
function removeDoubledLetters(str) {
    let result = "";
    for (let i = 0; i < str.length; i++){
        let currLetter = str[i];




    }
    return result;
}
```

💡 *Start with outline to build up a result string and scan the original string!*

🎯 Pattern Example from Lecture

Write a function `removeDoubledLetters` that takes a string and returns a new string with all doubled consecutive letters in the string replaced by a single letter.

```
removeDoubledLetters("bookkeeper") => "bokeper"
removeDoubledLetters("zzzz") => "z"
```

```
function removeDoubledLetters(str) {
    let result = "";
    for (let i = 0; i < str.length; i++){
        let currLetter = str[i];
        // check if not same as previous
        if (currLetter !== str[i - 1]){
            result += currLetter;
        }
    }
    return result;
}
```

Add letter to result if not same as previous!

🎯 Pattern Example from Lecture

Write a function `removeDoubledLetters` that takes a string and returns a new string with all doubled consecutive letters in the string replaced by a single letter.

```
removeDoubledLetters("bookkeeper") => "bokeper"
removeDoubledLetters("zzzz") => "z"
```

```
function removeDoubledLetters(str) {
    let result = "";
    for (let i = 0; i < str.length; i++){
        let currLetter = str[i];
        // check if starting letter or not same as previous
        if (i === 0 || currLetter != str[i - 1]){
            result += currLetter;
        }
    }
    return result;
}
```

✅ *Account for starting letter, and we're done!*

🔢 Arrays in JavaScript

```
let arr = [1,10,100];
```

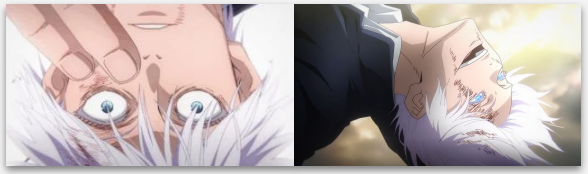
1	10	100
0	1	2

Sequence of elements of any type, with indices starting at `0` and ending at **`arr.length - 1`**, akin to strings.

`arr.length` **3** `arr[0]` **1** `arr[arr.length - 1]` **100**

Two crucial, ~~cursed~~ techniques with an array include:

1. Iterating through its elements in forward order
2. And in reverse order 🙃



1) Forward Iteration ➡️

```
for (let i = 0; i < arr.length; i++){
  let element = arr[i];
   // code to process each element in turn
}
// could also write for (let element of arr){ … }
```

2) Reverse Iteration ⬅️

```
for (let i = arr.length - 1; i >= 0; i--){
  let element = arr[i];
   // code to process each element in
   // turn, starting from the end
}
```

If forward doesn't seem to work well for a given task, give reverse a try!

Array Methods

What we can do with an a ray? ~~steal the moon!~~

🎶🔥 **Hot100 Methods – Top 3** Commonly Used

`arr.push(element,...)`
Adds one or more elements to end of array.

`arr.pop()`
Removes and returns the last element of array.

`arr.shift()`
Removes and returns the first element of array.

`arr.slice(start, finish)`
Similar to **substring** for strings: returns subarray from index *start* to *finish-1*, inclusive.

`arr.unshift(element,...)`
Adds one or more elements to the front of array.

`arr.splice(index, count, ...)`
Removes count elements starting at index, optionally adds new ones.

`arr.indexOf(element)`
Returns first index at which *element* appears, and -1 if not found.

`arr.concat(arr2,...)`
Concatenates one or more arrays into the receiver array **arr**.

`arr.lastIndexOf(element)`
Returns last index at which element appears, and -1 if not found.

`arr.reverse()`
Reverses the elements of **arr** directly.

`arr.sort()`
Sort the elements of **arr** directly, in ascending order.

Arrays Practice!

Objects in JavaScript

✍️ The easiest way to create new aggregates is through **JavaScript Object Notation (JSON).**

🔑 In **JSON,** you specify an object by **listing a collection of key-value pairs**, enclosed in ***curly braces***. Key/names are unique; values are not necessarily unique, and can be of any data type.

```
let bratAlbum = {
 title: "BRAT",
 artist: "Charli XCX",
 streams: 100000,
 criticScores: [96,...,92]
}; // note key:value,
```

Can iterate through object / maps as follows:

```
for (let key in map){
  let value = map[key];
  // code to work with individual key & value
}
```

Recall: Objects are unordered collections!

We can **select, add, or modify a key-value pair** of the object via an expression of the form:

🔑 `objectName.keyName` or 🔑 `objectName["keyName"]`

bratAlbum.streams → `100000`

bratAlbum["streams"] *= 4;

`400000`

🔑 `streams`

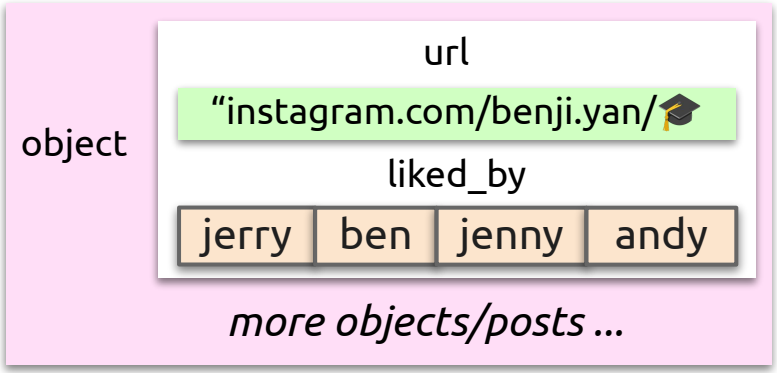
Working with Nested Data Structures

Nested data structures, like lists of lists, list of objects, JavaScript objects with lists / other objects inside as values, oh my, **can be tough!** 😨

📸 BensInstagramCringe

```
let benInstagramPosts = [
    {url: "instagram/benji.yan/🎓",
     likedBy:  ["jerry", "ben", "jenny", "andy"] },

    {url: "instagram/benji.yan/☔",
     likedBy:  ["eugene", "diego"] },
    … more IG posts stored as objects!
]
```



benInstagramPosts (array)

- **Advice:** Think one level of nesting at a time, and at each level (e.g., for an array of objects, `array` → `object` → `key,value pair within object`), it's helpful to know what kind of data type you're working with.
- **After all, an `array` of lists is still an `array` at the end of the day,** as is an `array` of objects, and can be treated as such (e.g., can still loop over elements)

Working with Nested Data Structures

Advice: You may even **name your variables** to make it clear what each nested one is!

printAllCells

```
let ticTacToeBoard = [["❌", " ", " "],
                      ["⭕", "❌", " "],
                      ["⭕", "⭕", "❌"]];
// array of string arrays
```

printAllCells

```
for (let row of ticTacToeBoard){ // row is array
  for (let cell of row){ // cell is string in array
    console.log(cell);
  }
}
```

Working with Nested Data Structures

Advice: You may even **name your variables** to make it clear what each nested one is!

printAllCells

```
let ticTacToeBoard = [["❌", " ", " "],
                      ["⭕", "❌", " "],
                      ["⭕", "⭕", "❌"]];
// array of string arrays
```

printAllCells

```
for (let row of ticTacToeBoard){ // row is array
  for (let cell of row){ // cell is string in array
    console.log(cell);
  }
}
```

printBensInstagramCringe

```
let benInstagramPosts = [
  {url: "instagram.com/benji.yan/🎓",
   likedBy:  ["jerry", "ben", "jenny", "andy"] },

  {url: "instagram.com/benji.yan/☔",
   likedBy:  ["eugene", "diego"] },
  … more IG posts stored as objects!
}
```

printBensInstagramCringe

```
// post is object, keys="url","likedBy"
for (let post of benInstagramPosts){
  let url = post["url"];  // string
  let likedBy = post["likedBy"]; // array of strings
  for (let user of likedBy){ // each user is string
    console.log(user + " liked " + url);
  }
}
```

🖌️ JavaScript Objects / JSON Data Practice!

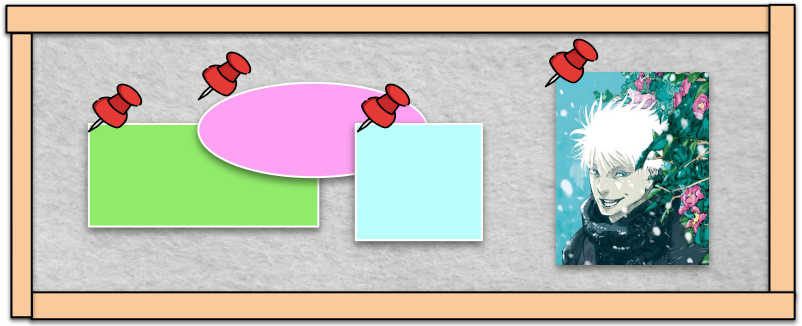
Happy to discuss any questions!

Next Up: Graphics

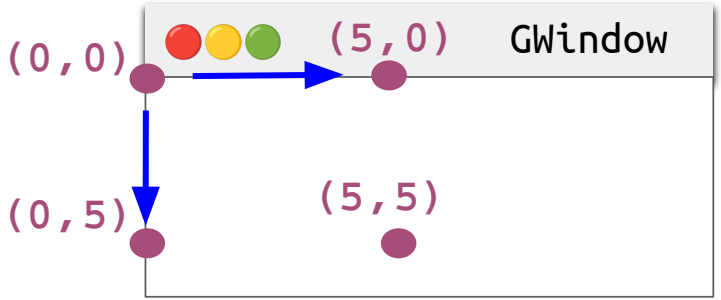
JSGraphics Rundown

You can find documentation in lecture slides / https://jdkula.github.io/jsgraphics-docs!

`GObjects` usually reside in a `GWindow`, which is like a bulletin board for tacking on colorful objects. **Newly added objects go on top if overlapping.**



For most `GObjects`, the location coordinate `(x,y)` is defined as its **upper left** corner (except `GLabel`, where its the bottom left / baseline of text)




```
GWindow(width, height);
GLine(xStart, yStart, xEnd, yEnd);
GOval(x, y, xDiameter, yDiameter);
GRect(x, y, width, height);
GLabel(label, x , y);
```

Constructors!

It's quirky, but unlike in math, the **y coordinates go up** as we go downward.

Graphics Methods Palette

📚 Especially as the midterm is **open notes,** you don't have to memorize these, but having an annotated reference guide may come in handy!

any GWindow()

`gw.add(object)`
`gw.add(object, x, y)`
`gw.remove(object)`
`let obj = gw.getElementAt(x,y)`

any GObject()

`let x = object.getX()`
`let y = object.getY()`
`let width = object.getWidth()`
`let height = object.getHeight()`
`object.setColor(color)`
`object.setLocation(x, y)`
`object.move(dx, dy)`

any GRect() or GOval()

`object.setFilled(fill)`
`object.setFillColor(color)`
`object.setBounds(x,y,width,height)`

any GArc()

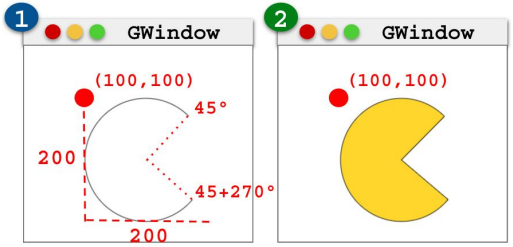
`arc.setStartAngle(start)`
`let angle = arc.getStartAngle()`
`arc.getSweepAngle(sweep)`
`let sweep = arc.getSweepAngle()`
`arc.setFrameRectangle(x,y,xDiameter, yDiameter)`

Notes:

- `gw.getElementAt()` can be useful for **detecting objects a mouse interacted** with.
- *color* is a string, e.g., `"Pink"`, `"White"`
- *fill* is a boolean `(true/false)`
- `.setColor()` specifies only boundary color if `.setFillColor()` is set to a different color.
- **For a `GOval's .setBounds()`, *width* and *height*** are the ***xDiameter*** and ***yDiameter*** .
- `dx` and `dy` are displacements from current coordinates, not the new `x,y` coordinates.
- **This isn't an exhaustive list!**

1
```
let arc = GArc(200, 200,
               45, 270);
gw.add(arc, 100, 100);
```

2
```
arc.setFillColor("Gold");
arc.setFilled(true);
// creates PacMan
```



GCompound

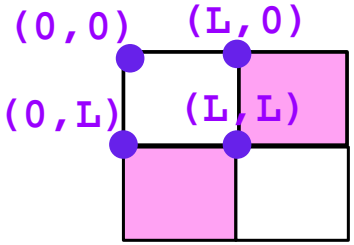
You can add objects to a **`GCompound`** like a **`GWindow`**, but you can also add it and move it the window around, akin to other **`GObjects`**. It's an elegant hybrid, useful for decomposition and when you want to animate objects in unison.

Graphics Tips

The art doesn't have to be perfect, as well it's art!

Embrace your art abilities, and **draw it out!** Sketch what the screen should look like; then, **trace out the objects and coordinates** you need to make that happen.

If you have a plethora of objects (e.g., the lecture checkerboard, the Breakout bricks), you can **work with a mini example** (`N_ROWS,NCOLS=2`), then generalize.



Each board square has coordinate `(row*L,col*L)`

Hollywood Analogy: For working with **animation / event listeners**, consider **variables (actors)** you need ahead of time*, e.g., `vx,vy,movingBall, growingCircle,` then **callback functions** (**TV episodes**) that feature and animate them around.

```
let ball = null; // GOval later
function step(){ // move ball }
function clickAction(e){
   // expand ball for few seconds
}
```

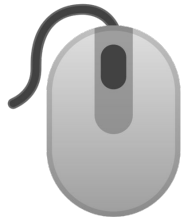
* For variables shared between `step` calls, e.g., same ball, initialize it outside! Leave extra room at the top.
* Callback functions can also introduce new **local variables** / **side characters,** e.g., each click creates a new falling ball.

Don't forget to actually add objects to the screen!

Event-Driven Programming

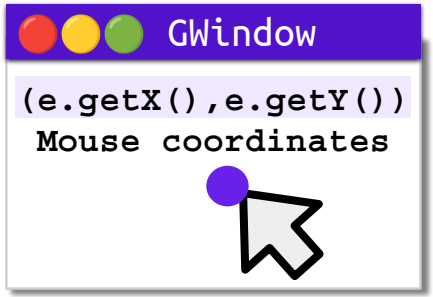
To make your program responsive to mouse interactions, you'll write a **listening function** that ***handles*** a type of user event—which has the general template:

```
let handleEvent = function(e){
    what program should do each time the event occurs
}
gw.addEventListener("click", handleEvent);
```



Last line tells the **GWindow** to **call `handleEvent` each and every time a click** occurs in the window. The **passed-in parameter `e`** provides information on the event, e.g.,

`"click"`	user clicks mouse in window
`"dblclk"`	user double-clicks mouse
`"mousedown"`	user presses the mouse button
`"click"`	user releases the mouse button
`"dblclk"`	user moves the mouse with the button up
`"mousedown"`	user drags the mouse with the button down



Note overlap, e.g., click is mousedown + mouseup, in that order

Timer-Based Animation

We have a step function, and 2 types of timers to animate objects with it:

1. **one-shot** timer, calls the step function **just once** after a specified millisec delay
2. **interval** timer, which calls step **repeatedly** at regular millisec time intervals.

```
let step = function(){ // callback function
    generally, makes some small update to Objects,
    e.g., moves them in x and/or y directions
}
```

1. `setTimeout(step, delay);`
2. `let timer = setInterval(step, delay);`

`setInterval()` returns a value (`timer`) you can later use to stop the timer and thus the animation, by calling `clearInterval(timer)`.

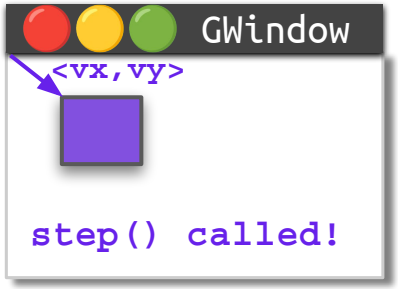
Timer-Based Animation Example

AnimatedSquare.js

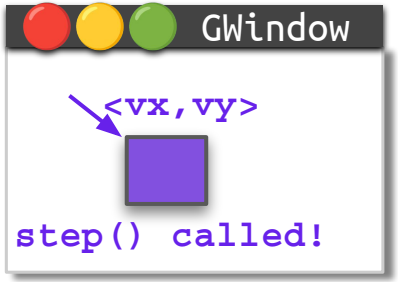
```
function AnimatedSquare() {
  // code to initialize GWindow, GRect square,
  // and x and y velocities dx and dy
  let stepCount = 0;
  let step = function() {
     square.move(dx, dy);
     stepCount++;
     if (stepCount === N_STEPS) clearInterval(timer);
  };
  let timer = setInterval(step, TIME_STEP);
} // say TIME_STEP is 10 milliseconds
```

We can think of ***setInterval*** as a "scheduler" which calls the ***step*** function to move the square every TIME_STEP ms.

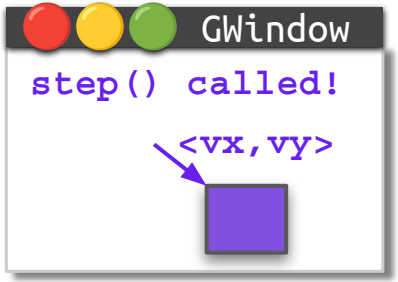
The scheduler is halted when ***clearInterval*** is called 🛑



t=10 ms



t=20 ms



t=30 ms

Happy to discuss any questions!

🎯 Next Up: Graphics practice problem!

🖌️ JS Graphics Practice!

📝 Closing Remarks

📓 **Check out the Handouts Page** on the course website—with 2 practice exams to study from!

🖌️ **Besides the practice exams,** you can look over section handouts and solutions to enrich understanding, as well as course assignment code!

💬 **Please feel free to post on EdStem forum with any questions!** (practice exam/solution clarifications, conceptual questions, exam tips, etc.)

Happy to take questions after the review session!!

You can do this! 💖

Good luck on the midterm!

You're 👋 going 👋 to 👋 be 👋 awesome.

We believe in you!

h	e	l	l	0
c	l	a	s	s
1	0	6	a	x

You win!