Fun, Friendly Computer Science

@mercedescodes | @mercedes | mercedesbernard.com

Hello! Today we're here to talk about Fun, Friendly Computer Science. This talk is going to be computer science quick hits. We're going to cover 11 topics in about 55 minutes. So because we don't have very much time, this talk will mainly focus on fundamental and introductory object-oriented programming concepts. But there's a whole world of other CS things to learn if you want to explore more.

If you have some familiarity with loops, array, and classes, you'll be able to follow along with this talk!

My name is Mercedes Bernard. My pronouns are she/her. And I'm a senior software engineer and engineering manager with a digital consultancy in Chicago called Tandem.

Why?

My background is in traditional, CS. I have a BS in CS. But the longer I'm in this industry, the more I realize that there is a lot that I learned that I never use. If you came into software from a non-traditional path, you may never have had the chance to learn this stuff. But you'll still be interviewed on it.

My goal with this talk is to show you that these topics that are used in interviews and sometimes used for gatekeeping aren't intimidating and also aren't really that important because a) you probably already know it and just don't have the words to explain it and b) you really don't use it very often.

You'll walk away from this talk with a high level understanding of a bunch of different topics as well as metaphors that you can use to explain them and examples that you can refer to later if you need them.

If you already know everything in this talk, that's great! But you'll probably still find something useful in explaining this to those you mentor or teach.

mercedesbernard.com/speaking/fun-friendly-cs

@mercedescodes | @mercedes | mercedesbernard.com

If you are someone who likes to reference slides or speaker notes while I'm talking, I've posted the slides for this talk here. I also tweeted out a link right before I got started so you can also find it on my Twitter profile.

github.com/mercedesb/fun-friendly-cs

There will be code samples in this talk and you can find them all in this repo. Be sure to check out the commits because each commit corresponds to one of the topics we'll cover here today.

All of the code you'll see today is written in vanilla js. This was an intentional choice. JS is not exactly known for being object oriented but I wanted to show that CS is more a way of thinking than it is a specific language or framework. There are some languages that are more functional in nature where you would have to force OO behavior, but anything is possible in code.

Vanilla JS also seemed the most accessible of my language options as opposed to .NET or Ruby. Most folks have dabbled a little and even if you only know a framework, you'll still be able to get by. If you've never used JS before, a lot of the syntax is easy to follow.

Test files are probably the most useful and where you should start to understand what the example is trying to show. But not everything has tests because in some cases, like set theory, the tests were no more valuable than the code.

Agenda

1. Concepts

- Big O notation
- Set theory
- Recursion

2. Data structures

- Linked list
- Stack
- Queue
- Tree

3. Principles of OO programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Concepts

Big O Notation

Cooking with ratios is when you don't memorize recipes but instead memorize the ratios of ingredients. The amount of ingredients you need changes *in proportion* to how much of the food you want to make. For example, cupcakes follow a 4:3:2:1 ratio.

Big O Notation measures relative complexity of a function or algorithm. Most often this is measuring running time but it could also be used to measure memory consumption, stack depth, and other resources. We're going to focus on running time since in an interview, that's what they're usually asking about. The actual input size is unimportant because we want to measure the proportional complexity of the logic.

This measurement is language/hardware/time agnostic and is relative to the code's input size.

We also talk about it according to worst case scenario. Sometimes in different sort and search algorithms you get lucky and the collection is nearly sorted or the object is near the beginning of your iteration but when we're talking about complexity, we want to plan for the worst case scenario.

O(1)

```
combineButterAndSugar() {
  return this.beatWithElectricMixer(
  [
    {
      ingredient: this.recipeRatios.butter,
      amount: this.numberOfBatches *
this.recipeRatios.butter.number
    },
    {
      ingredient: this.recipeRatios.sugar,
      amount: this.numberOfBatches *
this.recipeRatios.sugar.number
    }
  ],
  3));
}
```
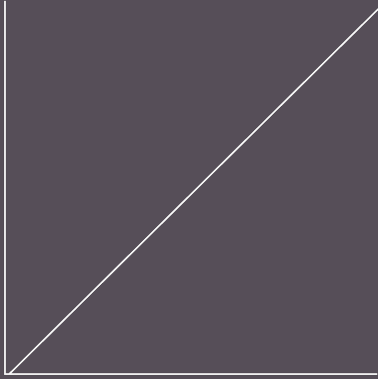
O(1) = Constant running time regardless of input size

O(n)

```
addEggs() {
  const steps = [];
  const amount = this.numberOfBatches *
this.recipeRatios.eggs.number;

  for (let egg = 0; egg < amount; egg++) {
    steps.push(
     this.beatWithElectricMixer(
     [
       { ingredient: this.recipeRatios.eggs,
amount: 1 },
       { ingredient: "butter mixture" }
     ],
     1));
  }

  return steps;
}
```

O(n) = Running time proportional to input size and running time increases linearly

$O(n^2)$
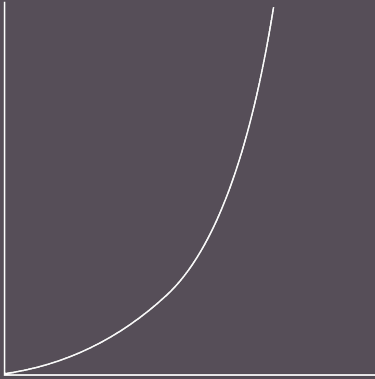
```
combineFlourMixtureAndMilkAndButterMixture() {
  const steps = [];

  // abbreviated for slides

  for (let batch = 0; batch < this.numberOfBatches; batch++)
  {
    for (let part = 0; part < this.numberOfBatches; part++)
    {
      // abbreviated for slides
      steps.push(
        this.beatWithElectricMixer(
          [// butter mixture and milk ]
          ,1)
      );
      steps.push(
        this.beatWithElectricMixer(
          [// butter mixture and flour mixture],
          1)
      );
    }
  }

  return steps;
}
```

@mercedescodes | @mercedes | mercedesbernard.com

O(n^2) = Running time proportional to the square of the input size. This is common in nested iterations.

$O(2^n)$
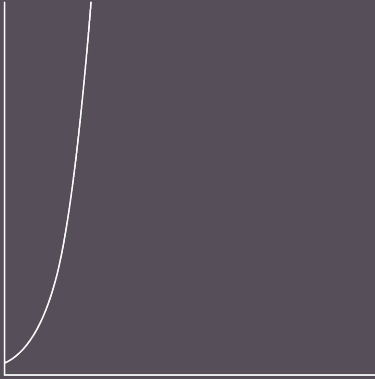
```
fibonacciFrosting() {
  const fibNum = this.calculate(this.numberOfBatches);
  return `Icing the fibonacci number ${fibNum} on all
of the cupcakes.`;
}

calculate(number) {
  if (number <= 1) return number;
  return (
    this.calculate(number - 1) +
    this.calculate(number - 2)
  );
}
```

@mercedescodes | @mercedes | mercedesbernard.com

$O(2^n)$ = Running time grows exponentially with the size of the input. For example, calculating Fibonacci recursively
$O(\log n)$ = This is kinda the opposite of $O(2^n)$.

Big O Uses

Interviews
Comparing the performance of 2 possible solutions
Having a shared language

Agenda

1. Concepts

- ~~Big O notation~~
- Set theory
- Recursion

2. Data structures

- Linked list
- Stack
- Queue
- Tree

3. Principles of OO programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Set theory

Instagram accounts with cute dogs

@tinyheelercrew @yukimomon @fomo.the.cat
@thegraypotato_ @_merlintheaussie_ @maple.cat
@wolfgang2242 @henrythecoloradodog @basilfold
@greysonthebearwolf @wat.ki @madfluffs

Instagram accounts with cute cats

@mercedescodes | @mercedes | mercedesbernard.com

Venn diagrams are a great way to think about set theory. A set is a data structure similar to arrays or lists but it is an **unordered collection** of objects with **no duplicates**.

Sets are more interesting for what we can do on them. These operations form the basis of set theory.

Union: X ∪ Y

All of the cute accounts

Instagram accounts with cute dogs

@tinyheelercrew   @yukimomon   @fomo.the.cat

@thegraypotato_   @_merlintheaussie_   @maple.cat

@wolfgang2242   @henrythecoloradodog   @basilfold

@greysonthebearwolf   @wat.ki   @madfluffs

Instagram accounts with cute cats

Union - X ∪ Y The stuff that exists in X OR Y

Intersection: X ∩ Y

Accounts featuring cute dogs and cats

Instagram accounts with cute dogs

@tinyheelercrew   @yukimomon   @fomo.the.cat

@thegraypotato_   @_merlintheaussie_   @maple.cat

@wolfgang2242   @henrythecoloradodog   @basilfold

@greysonthebearwolf   @wat.ki   @madfluffs

Instagram accounts with cute cats

Intersection X ∩ Y - The stuff that exists in X AND Y

Difference: X - Y

Accounts featuring only cute dogs (no cats allowed)

Instagram accounts with cute dogs

@tinyheelercrew
@thegraypotato_
@wolfgang2242
@greysonthebearwolf

@yukimomon
@_merlintheaussie_
@henrythecoloradodog
@wat.ki

@fomo.the.cat
@maple.cat
@basilfold
@madfluffs

Instagram accounts with cute cats

Difference - X - Y The stuff that only exists in X

Relative Complement: Y \ X

Accounts featuring only cute cats (no dogs allowed)

Instagram accounts with cute dogs

@tinyheelercrew   @yukimomon   @fomo.the.cat

@thegraypotato_   @_merlintheaussie_   @maple.cat

@wolfgang2242   @henrythecoloradodog   @basilfold

@greysonthebearwolf   @wat.ki   @madfluffs

Instagram accounts with cute cats

@mercedescodes | @mercedes | mercedesbernard.com

Relative complement Y \ X (same as Y - X) The stuff that only exists in Y

Symmetric Difference: X △ Y

Accounts featuring either dogs or cats, but not both (no cross species friendships here)

Instagram accounts with cute dogs

@tinyheelercrew
@thegraypotato_
@wolfgang2242
@greysonthebearwolf

@yukimomon
@_merlintheaussie_
@henrythecoloradodog
@wat.ki

@fomo.the.cat
@maple.cat
@basilfold
@madfluffs

Instagram accounts with cute cats

@mercedescodes | @mercedes | mercedesbernard.com

Symmetric difference (disjunctive union) X △ Y
Same as (X ⟍ Y) ∪ (Y ⟍ X).
The stuff that exists in only X and the stuff that exists in only Y but none of the stuff that exists in both

Set Theory Uses

Set theory is the foundation of relational databases
Website filters

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- Recursion

2. Data structures

- Linked list
- Stack
- Queue
- Tree

3. Principles of OO programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Recursion

Russian nesting dolls are a great metaphor for recursion, because each doll is the same except for its size. The dolls continue to open until you get to the smallest child which does not open. When you reach the smallest child, your reverse the process, closing each doll one by one in reverse order.

Recursion is the process in which a function calls itself directly or indirectly. And the function that is doing this calling of itself is called a recursive function. Everything that you do recursively you can also do in a loop.

When writing a recursive function, we don't want it to continue calling itself infinitely so we have to set up a condition where it exists the nesting and returns a finite value. This is called the **base case**. The smallest doll in Russian nesting dolls is like the base case.

When you're writing a recursive function, the base case is usually the easiest place to start.

```
count() {
  return this.countNestedDolls(this.bigDoll);
}

countNestedDolls(doll) {
  const child = doll.open();

  // base case
  if (!child) {
    return 1;
  } else {
    return this.countNestedDolls(child) + 1;
  }
}
```

Recursion Uses

Navigation paths on a website

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- Linked list
- Stack
- Queue
- Tree

3. Principles of OO programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Data Structures
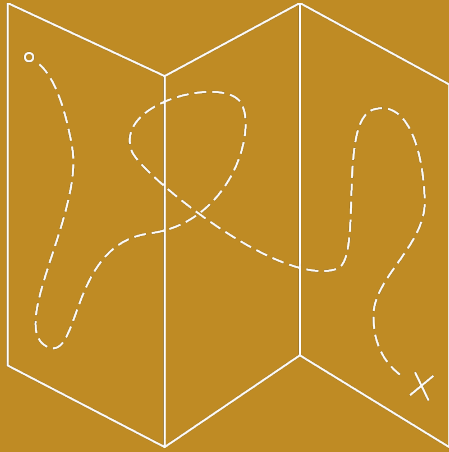
Linked List

When you are following a scavenger hunt, you start with the first clue and must follow each clue sequentially one at a time. Because you have no way of knowing where the nth clue is when you start, you must follow the hunt from clue to clue to clue.

A scavenger hunt is a good metaphor for a linked list. A linked list is a data structure characterized by sequential data access and no random access. This is unlike arrays. In arrays, because all of the data is stored in contiguous locations in memory, you can do simple addition and subtraction to find the memory location of the data at the nth node. For example, array[3] is just 3 memory
locations from array[0]. Linked lists on the other hand, do not need contiguous memory allocated. Each node in a linked list points to where the next node is located.

Because of the memory allocation, a linked list will have no wasted space because it can grow and shrink dynamically whereas traditionally an array always needs to be allocated to maximum size (or copied to a new array if it needs to grow). We tend to forget this when working with Javascript since JS Arrays behave more like ArrayList.

However, because a linked list node needs to store its data and a reference to the next node, it does take up a bit more space than an array of the same data.

Linked lists have fast insertion and deletion if its at the head or tail. But to insert into the middle of a linked list is O(n) because we need to **loop through the list** to the place we want to insert, update the previous nodes pointer to the inserted node and

update the inserted nodes pointer to the next node.

```
add(data) {
  const newNode = new LinkedListNode(data);

  if (this._head === null) {
    this._head = newNode;
  } else {
    this._tail.next = newNode;
  }
  this._tail = newNode;
  return newNode;
}
```

```
add(data) {
  const newNode = new LinkedListNode(data);

  if (this._head === null) {
    this._head = newNode;
  } else {
    this._tail.next = newNode;
  }
  this._tail = newNode;
  return newNode;
}
```

```
add(data) {
  const newNode = new LinkedListNode(data);

  if (this._head === null) {
    this._head = newNode;
  } else {
    this._tail.next = newNode;
  }
  this._tail = newNode;
  return newNode;
}
```

```
add(data) {
  const newNode = new LinkedListNode(data);

  if (this._head === null) {
    this._head = newNode;
  } else {
    this._tail.next = newNode;
  }
  this._tail = newNode;
  return newNode;
}
```

```
remove() {
  if (this._head === null) return undefined;

  let removed = this._head;
  this._head = this._head.next;
  return removed;
}
```

```
remove() {
  if (this._head === null) return undefined;

  let removed = this._head;
  this._head = this._head.next;
  return removed;
}
```

```
insert(data, index) {
  if (index < 0) return undefined;

  const newNode = new LinkedListNode(data);

  if (index === 0) {
    newNode.next = this._head;
    this._head = newNode;
  } else {
    let current = this._head;
    let previous = null;

    let i = 0;
    while (current !== null && i < index) {
      previous = current;
      current = current.next;
      i++;
    }

    if (current !== null) {
      previous.next = newNode;
      newNode.next = current;
    } else if (previous === this._tail) {
      this.add(data);
    }
  }
}
```

```
insert(data, index) {
  if (index < 0) return undefined;

  const newNode = new LinkedListNode(data);

  if (index === 0) {
    newNode.next = this._head;
    this._head = newNode;
  } else {
    let current = this._head;
    let previous = null;

    let i = 0;
    while (current !== null && i < index) {
      previous = current;
      current = current.next;
      i++;
    }

    if (current !== null) {
      previous.next = newNode;
      newNode.next = current;
    } else if (previous === this._tail) {
      this.add(data);
    }
  }
}
```

```javascript
insert(data, index) {
  if (index < 0) return undefined;

  const newNode = new LinkedListNode(data);

  if (index === 0) {
    newNode.next = this._head;
    this._head = newNode;
  } else {
    let current = this._head;
    let previous = null;

    let i = 0;
    while (current !== null && i < index) {
      previous = current;
      current = current.next;
      i++;
    }

    if (current !== null) {
      previous.next = newNode;
      newNode.next = current;
    } else if (previous === this._tail) {
      this.add(data);
    }
  }
}
```

```
insert(data, index) {
  if (index < 0) return undefined;

  const newNode = new LinkedListNode(data);

  if (index === 0) {
    newNode.next = this._head;
    this._head = newNode;
  } else {
    let current = this._head;
    let previous = null;

    let i = 0;
    while (current !== null && i < index) {
      previous = current;
      current = current.next;
      i++;
    }

    if (current !== null) {
      previous.next = newNode;
      newNode.next = current;
    } else if (previous === this._tail) {
      this.add(data);
    }
  }
}
```

Linked List Uses

@mercedescodes | @mercedes | mercedesbernard.com

When you don't know the size of the data ahead of time and you don't need efficient random access.

Implementing your own stack or queue… but would you really? Probably not since most languages and frameworks have these data structures built for you already. We don't want to reinvent the wheel.

Blockchain

Doubly linked list
Playlist

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- ~~Linked list~~
- Stack
- Queue
- Tree

3. Principles of OO programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
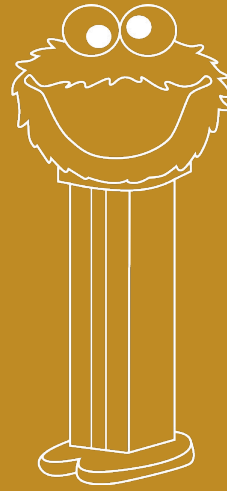
Stack

A PEZ dispenser is wonderful visualization of a stack. When you fill it with candy, you are pushing the candy from the top down and when you eat the candy, you pop a piece of the top of the literal stack of pieces one at a time.

A stack data structure behaves the same way. Stacks are characterized by last in, first out (LIFO) data access. When implemented as a linked list, adding and removing from the stack are O(1).

```javascript
push(data) {
  const newNode = new LinkedListNode(data);
  if (this._head) {
    newNode.next = this._head;
  }
  this._head = newNode;
  return newNode;
}

pop() {
  const popped = this._head;
  this._head = this._head.next;
  return popped;
}
```

Stack Uses

When you want to enforce LIFO data access.
Browser back button
Undo/redo feature
Checking for balanced delimiters in a string (parentheses, quotes, HTML tags, etc)

Agenda

1. Concepts
- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures
- ~~Linked list~~
- ~~Stack~~
- Queue
- Tree

3. Principles of OO programming
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
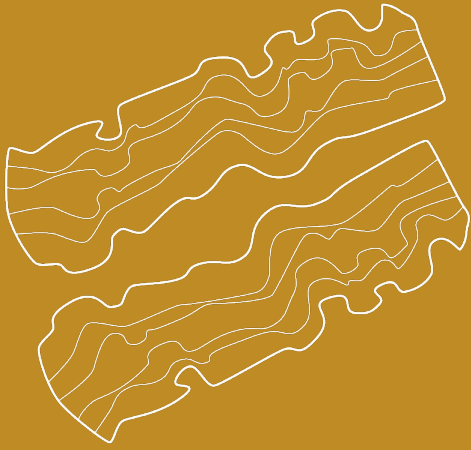
Queue

Waiting in line, or as the British call it queuing up, is a literal
visualization of a queue. The first person in line is the first person out of line. And new
folks joining the line (should) always join at the back.

A queue data structure behaves the same way. Queues are characterized by first in,
first out (FIFO) data access. When implemented as a linked list, enqueuing and
dequeuing from the queue are O(1).
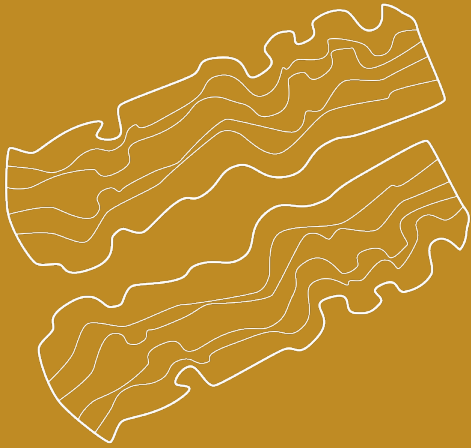
```
enqueue(data) {
  const newNode = new LinkedListNode(data);

  if (this._head === null) {
    this._head = newNode;
  } else {
    this._tail.next = newNode;
  }
  this._tail = newNode;
  return newNode;
}

dequeue() {
  const dequeued = this._head;
  this._head = this._head.next;
  return dequeued;
}
```

Queue Uses

When you want to enforce FIFO data access.
Processing jobs asynchronously but in the order they were scheduled

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- ~~Linked list~~
- ~~Stack~~
- ~~Queue~~
- Tree

3. Principles of OO programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Tree
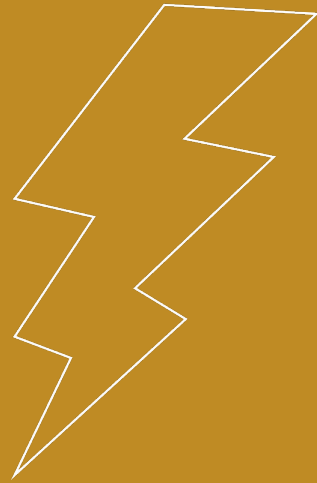
If you're a big Harry Potter fan, you probably enjoyed the end of the fourth book that featured a giant maze full of magical obstacles. The first person to the center of the maze won the Triwizard cup. When you are completing a maze, you are often faced with choices about whether to go left or right or in some cases continue straight down the center path.

Modeling these choices can be done using a tree data structure. Unlike linked lists, stacks, or queues, trees are useful to model hierarchical data. Think file systems or organization charts. In a tree data structure, there is a root node that has 1 to many children (in much the same way that a linked list stores a reference to the next node in the list).

To access other nodes in the tree, you must traverse through the nodes.

Harry Potter & the Triwizard Maze

start

Harry

Cedric

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24
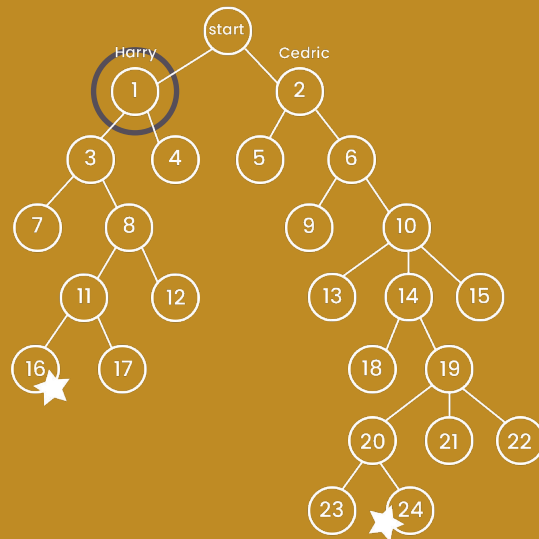
Breadth First Search

Breadth first search is when you traverse the nodes one level at a time, visiting all the nodes on level 1, then level 2 and so on. In the case of a maze, BFS is the worst way to get to the center quickly.

Breadth First Search

Breadth First Search

Breadth First Search

Breadth First Search

Breadth First Search

Breadth First Search

Breadth First Search

Depth First Search

Depth first search is when you traverse the nodes as deep as possible from one side of the tree to the other. In a maze, DFS is the best way to get to the center quickly but you can see here that the villain in the book laid a trap for Harry in the center of the maze and purposefully set up the maze so that even Cedric's best route would be worse than Harry's worst route.

Depth First Search

Depth First Search

Depth First Search

Depth First Search

Depth First Search

Depth First Search

@mercedescodes | @mercedes | mercedesbernard.com

Depth First Search
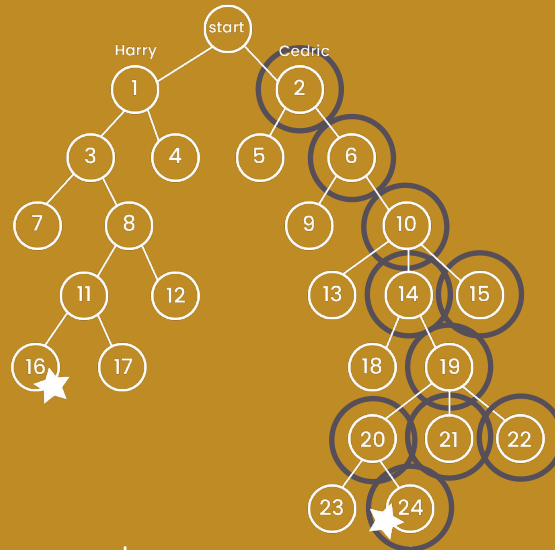
Depth First Search

Depth First Search

Tree Uses

There are many many types of trees that are out of scope for this talk but the most common you'll hear about are binary trees (of which there are also many types). Binary trees are trees where the nodes have at most 2 children.

Binary trees on their own aren't super useful but there are many types of binary trees that have further constraints making them more useful. For example, a binary search tree.

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- ~~Linked list~~
- ~~Stack~~
- ~~Queue~~
- ~~Tree~~

3. Principles of OO programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Principles
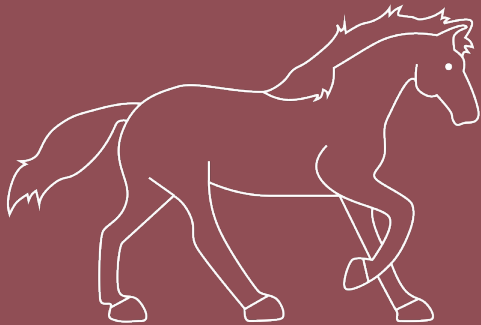
Encapsulation

Let's pretend that we're going to code a video game where the user had to move the horse. A horse has 4 gaits, a walk, trot, gallup, or canter.

Walking: 4 beat pace, left hind leg, left front leg, right hind leg, right front leg
Trot: 2 beat pace, left hind leg + right front leg, right hind leg + left front leg
Canter: 3 beat pace, left hind leg, right hind leg + left front leg, right front leg
Gallop: 4 beat pace, left hind leg, right hind leg, left front leg, right front leg

So everytime the horse needs to move, we don't want the object telling the horse to move to tell it which leg to move and how quickly. This is incredibly error prone and could result in the horse ending up in a broken state. They could have too many legs in the air or they could be moving at the wrong pace. Instead, we'd define 4 methods, one for each gait, so the horse can handle moving its own legs at the appropriate pace. Those 4 methods that *encapsulate* the state of each leg while it's moving.

Encapsulation is when you hide an object's state from other objects. You create a public interface (a method) for other objects to interact with your object and mutate its state.
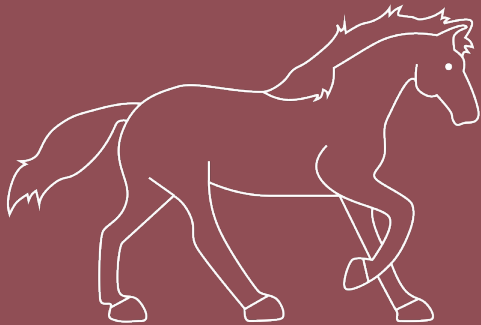
```
canter(steps) {
  let time = 0;
  let distanceTraveled = 0;

  for (let i = 0; i < steps; i++) {
    this[backLeftLegPosition] = LegPosition.UP;
    this[backLeftLegPosition] = LegPosition.DOWN;

    this[backRightLegPosition] = LegPosition.UP;
    this[frontLeftLegPosition] = LegPosition.UP;
    this[backRightLegPosition] = LegPosition.DOWN;
    this[frontLeftLegPosition] = LegPosition.DOWN;

    this[frontRightLegPosition] = LegPosition.UP;
    this[frontRightLegPosition] = LegPosition.DOWN;

    time += 0.27; // 3.667 steps / second
    distanceTraveled += 6; // 22 ft / second
  }

  return { steps, time, distanceTraveled };
}
```
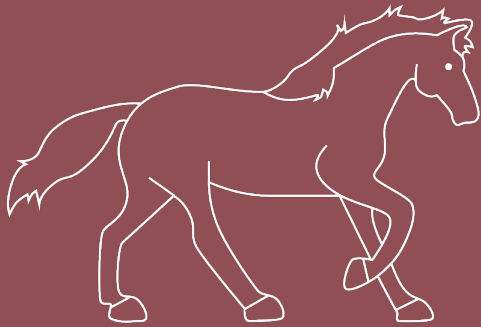
```
const GiddyUp = new Horse();
const steps = // read from input;
GiddyUp.canter(steps);
```

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- ~~Linked list~~
- ~~Stack~~
- ~~Queue~~
- ~~Tree~~

3. Principles of OO programming

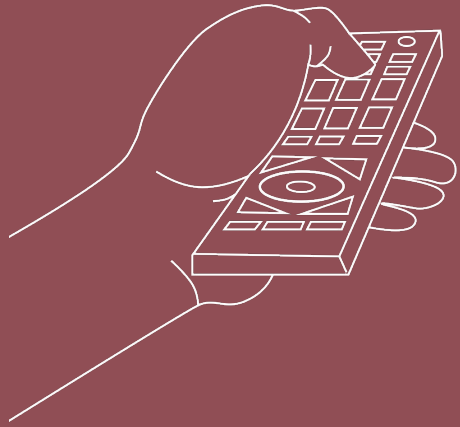- ~~Encapsulation~~
- Abstraction
- Inheritance
- Polymorphism

Abstraction

When you push a button on a remote control, something happens on your TV. The volume goes up, the channel changes, etc. There are few, finite things you can do with a remote. And you don't need to know how it works in order to make your change happen. For example, you don't need to know how infrared light works, how the binary is encoded or decoded, or how the microprocessor in your TV carries out the action.

This is an *abstraction*. You push a button and a thing happens. And you have only a few buttons to choose from.

Abstraction is an extension of encapsulation. Abstraction refers to hiding all the internal implementation details of a class and providing very few, clear mechanisms for other objects in the code to interact with each other.

You can create different abstractions… think the knobs on a TV from a 50s that required you to get up to change the channel

```
const Remote = new RemoteControl();
let tvState = Remote.power();
renderTV(tvState);
tvState = Remote.turnUpVolume();
renderTV(tvState);
tvState = Remote.turnUpVolume();
renderTV(tvState);
tvState = Remote.turnDownVolume();
renderTV(tvState);
```
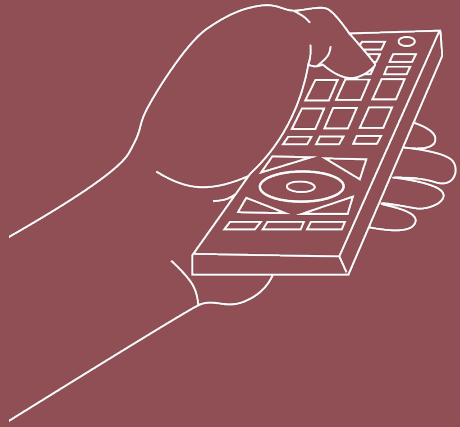
```
export default class RemoteControl {
  // ... abbreviated code for slides
  turnUpVolume() {
    return handleButtonClick(VolumeUp, this.television);
  }
}

function handleButtonClick(button, television) {
  const encodedData = encodeButtonPressIntoBinary(button);
  return sendBinaryDataAsInfraredLight(encodedData,
television);
}

function sendBinaryDataAsInfraredLight(binaryData,
television) {
  const infraredLight =
convertBinaryToInfraredLight(binaryData);
  return television.handleRemoteControlClick(infraredLight);
}

function encodeButtonPressIntoBinary(button) {
  // ... button data is encoded into binary and returned
}

function convertBinaryToInfraredLight(binaryData) {
  // ... binary data is converted and returned
}
```
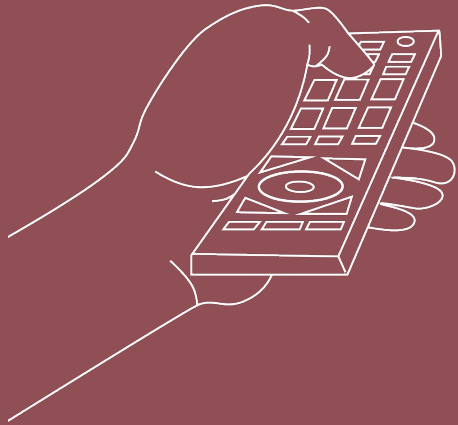
Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- ~~Linked list~~
- ~~Stack~~
- ~~Queue~~
- ~~Tree~~

3. Principles of OO programming

- ~~Encapsulation~~
- ~~Abstraction~~
- Inheritance
- Polymorphism

Inheritance

All garden plants need sun, soil, and water to survive. And if you want to learn how to garden, you'll be reading about the plant's care instructions that describe how to help your plant stay healthy and thrive.

If we were going to code up an app that displayed each plant's care instructions, we can use inheritance to share the code accesses each plant's needs and prints them out in a human understandable format.

Inheritance supports reusability in programming. A child class (or sub class) can inherit all of the fields/methods/properties from another class (called the base or super class) and then implement its own that differ or are in addition to what exists in the base class.

```javascript
export default class Plant {
  // ... abbreviated code for slides
  get sun() {
    return this[Sun];
  }

  get shade() {
    return !this[Sun];
  }

  get plantingInstructions() {
    return `Planting instructions: ${this.soilNeeds}`;
  }

  get careInstructions() {
    return `Sunlight needs: ${this.lightNeeds}
<br/>Watering instructions: ${this.waterNeeds}`;
  }

  learnHowToGarden() {
    return `${this.name}\n
      ${this.plantingInstructions}\n
      ${this.careInstructions}`;
  }
}
```

```
export default class Geranium extends Plant {
  constructor() {
    super();
    this.name = "Geranium";
    this.sun = true;
    this.wet = false;
    this.lightNeeds = "4 - 6 hours of direct
sunlight per day.";
    this.soilNeeds =
      "Plant in a pot with soil-less potting
mixture and good drainage.";
    this.waterNeeds =
      "Water thoroughly and allow to soil to
completely dry between waterings.";
  }
}
```

```
const Plants = [new Geranium(), new Begonia(), new Coleus()];
Plants.forEach(plant => plant.learnHowToGarden());
```

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- ~~Linked list~~
- ~~Stack~~
- ~~Queue~~
- ~~Tree~~

3. Principles of OO programming

- ~~Encapsulation~~
- ~~Abstraction~~
- ~~Inheritance~~
- Polymorphism
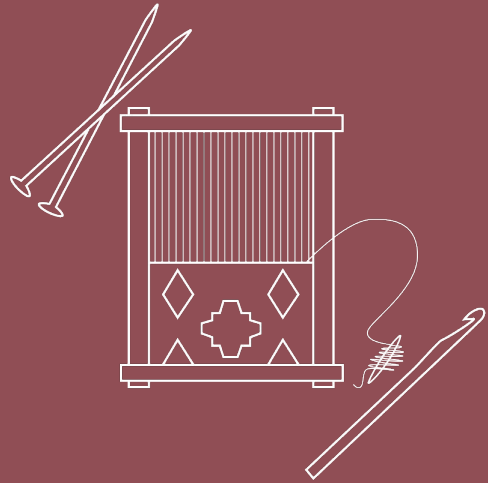
Polymorphism

If you are an avid crafter, you may know multiple different crafts in which you could create a fabric. And depending on the type or style of fabric you want, you would use a different craft. But at the end of the day, you want to do the same thing: create fabric. If we were to model that programmatically, we would take advantage of polymorphism.

Polymorphism is an object oriented concept where you can use multiple classes in exactly the same way so that their concrete class doesn't matter. Inheritance is one way to achieve polymorphism. But other ways can include duck typing (defining the same method signature on multiple classes) or using interfaces (if you are using a statically typed language that supports them).

```
createFabric(numberOfRows) {
  let fabric = [];
  for (let row = 0; row < numberOfRows; row++) {
    if (row % 2 === 0) {
      fabric.push(this.stitchRow(Knit));
    }
    if (row % 2 > 0) {
      fabric.push(this.stitchRow(Purl));
    }
  }
  return fabric;
}

stitchRow(stitch) {
  let row = "";
  for (let stitch = 0; stitch < this.rowLength; stitch++) {
    if (stitch === Knit) {
      row += this.knit();
    } else if (stitch === Purl) {
      row += this.purl();
    }
  }

  row += "Turn.\n";
  return row;
}
```
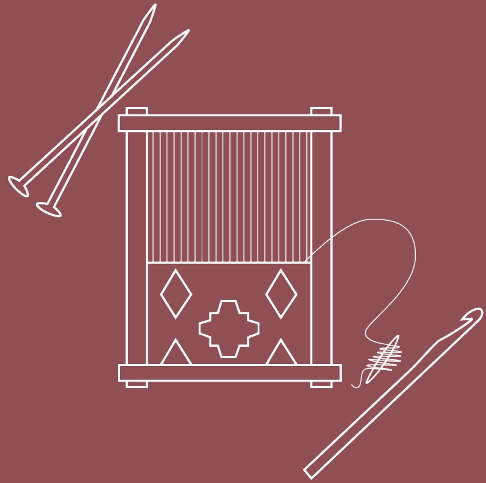
In this code example, we use duck typing to achieve polymorphism where each yarn craft class defines a `createFabric` method, allowing us to use them all exactly the same way without caring about which concrete class we're using at that moment.

```javascript
createFabric(numberOfRows) {
  let fabric = [];
  for (let row = 0; row < numberOfRows; row++) {
    fabric.push(this.weaveRow(row));
  }
  return fabric;
}

weaveRow(rowNumber) {
  let row = ""
  for (let stitch = 0; stitch < this.rowLength; stitch++) {
    const isEvenRow = rowNumber % 2 === 0;
    const isEvenStitch = stitch % 2 === 0;

    if ((isEvenRow && isEvenStitch) || (!isEvenRow &&
!isEvenStitch)) {
      row += this.weaveWeftOverWarp();
    } else if ((isEvenRow && !isEvenStitch) || (!isEvenRow &&
isEvenStitch)) {
      row += this.weaveWeftUnderWarp();
    }
  }

  row += "Turn.\n";
  return row;
}
```
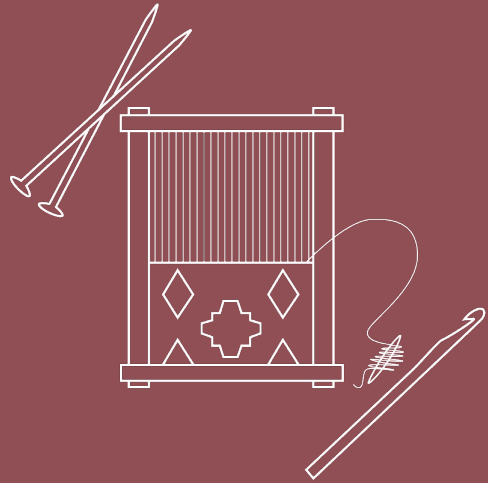
```
createFabric(numberOfRows) {
  let fabric = [];
  for (let row = 0; row < numberOfRows; row++) {
    fabric.push(this.stitchRow());
  }
  return fabric;
}

stitchRow() {
  let row = "";
  for (let stitch = 0; stitch < this.rowLength; stitch++) {
    row += this.singleCrochet();
  }
  row += "Turn.\n";
  return row;
}
```
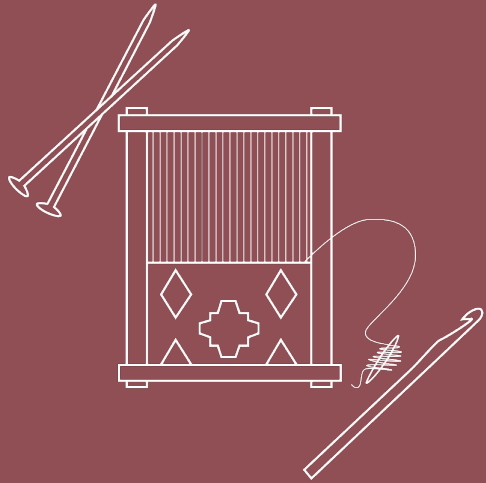
```javascript
let YarnCraft;
const craft = // read from input;
const rowLength = // read from input;
const numberOfRows = // read from input;

if (craft === "knit") {
  YarnCraft = Knitting;
} else if (craft === "crochet") {
  YarnCraft = Crocheting;
} else if (craft === "weave") {
  YarnCraft = Weaving;
}

const chosenCraft = new YarnCraft(rowLength);
const fabric = chosenCraft.createFabric(numberOfRows);
```
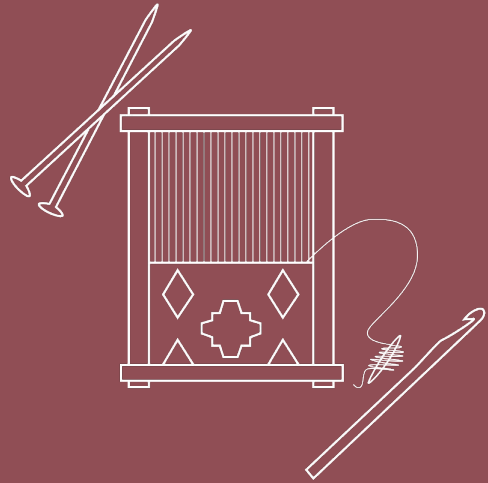
Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- ~~Linked list~~
- ~~Stack~~
- ~~Queue~~
- ~~Tree~~

3. Principles of OO programming

- ~~Encapsulation~~
- ~~Abstraction~~
- ~~Inheritance~~
- ~~Polymorphism~~
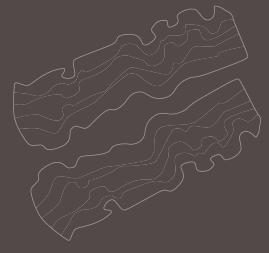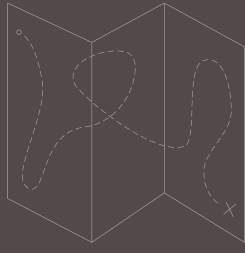
mercedesbernard.com/speaking
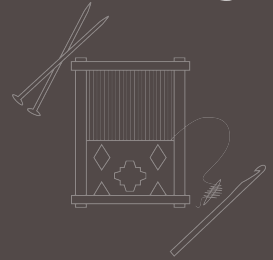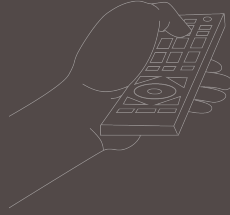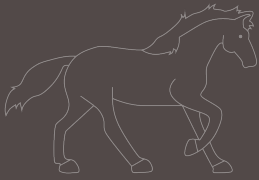
@mercedescodes | @mercedes | mercedesbernard.com

Thank you