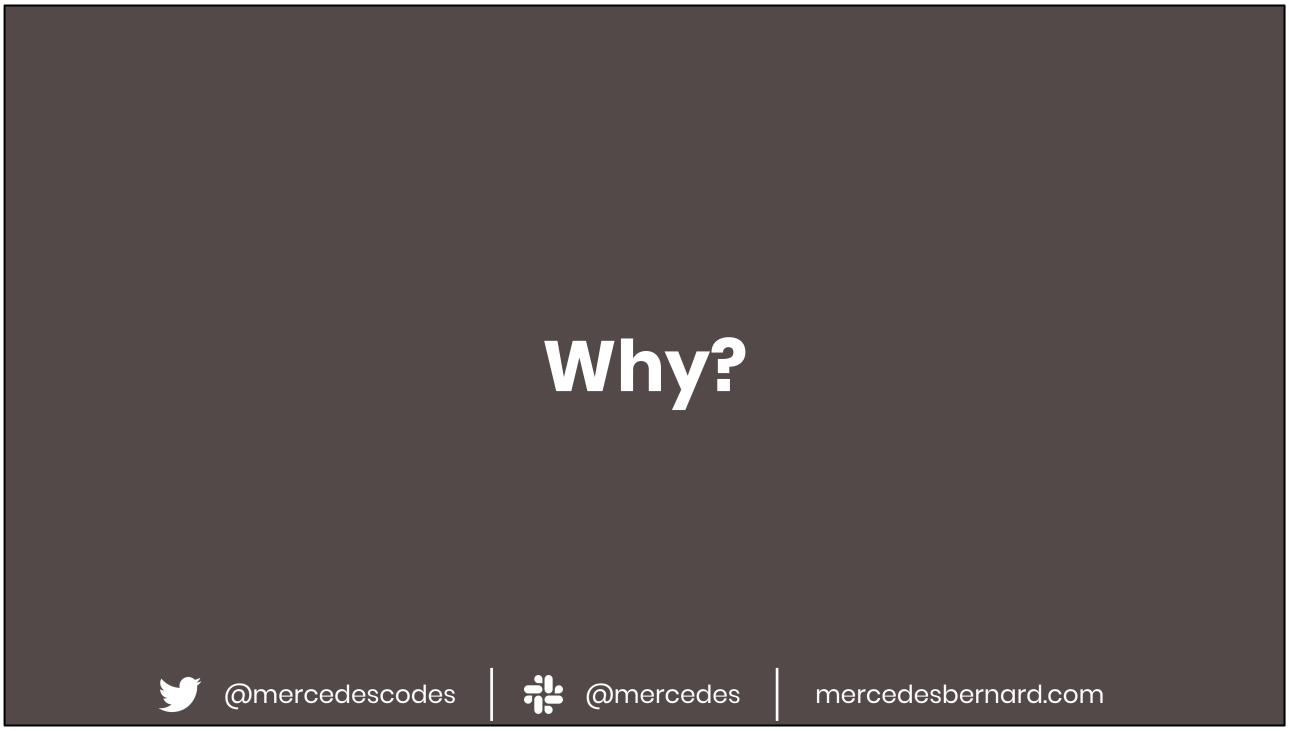
Fun, Friendly Computer Science

@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?

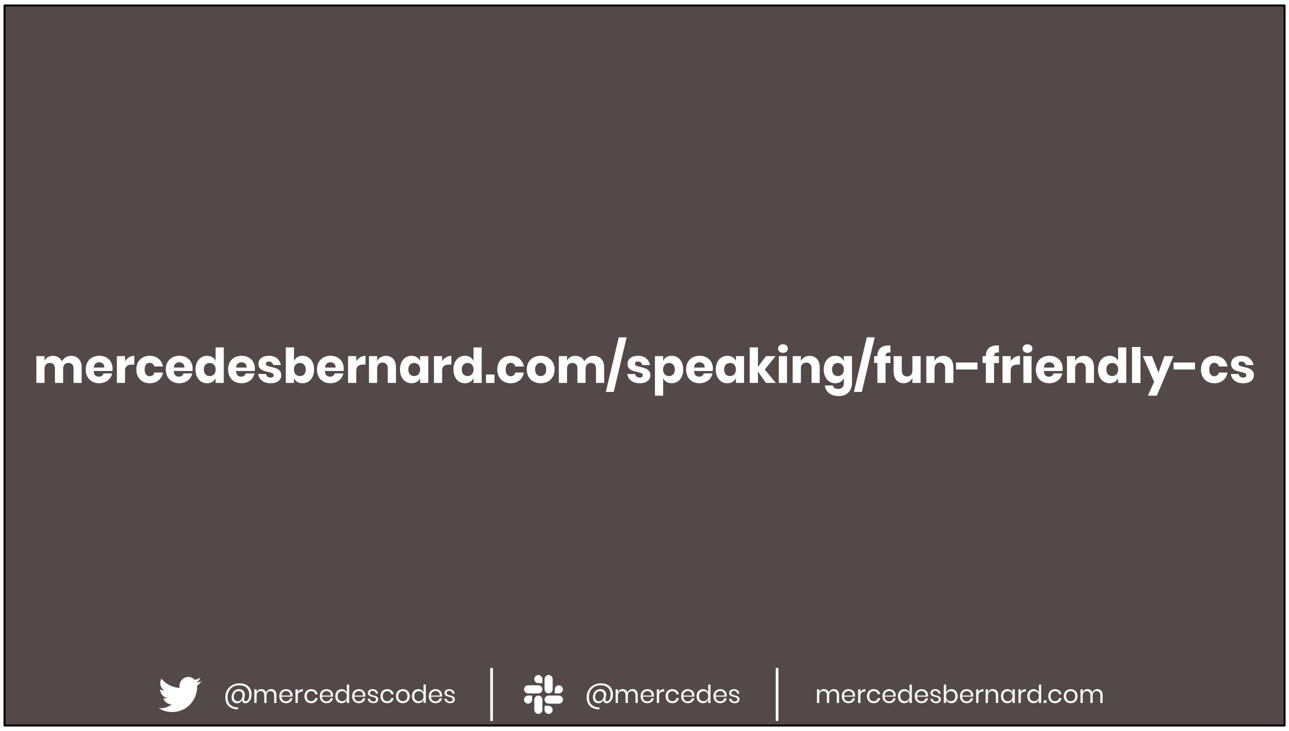
@mercedescodes | @mercedes | mercedesbernard.com

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-js

@mercedescodes | @mercedes | mercedesbernard.com

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.

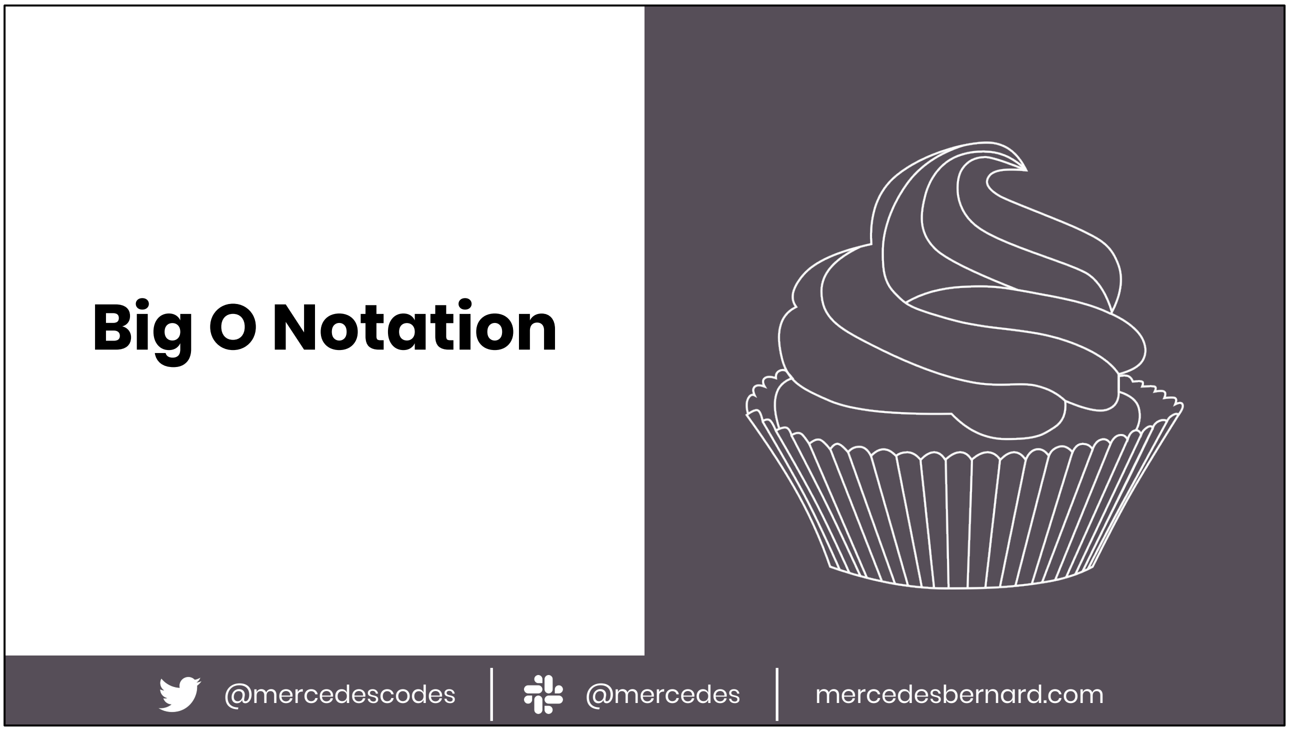
Test files are probably the most useful and where you should start to understand what the example is trying to show. But not everything has tests because in some cases, like set theory, the tests were no more valuable than the code.

Agenda

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

2. **Principles of OO programming**
   - Encapsulation
   - Abstraction
   - Inheritance
   - Polymorphism

Concepts



Big O Notation

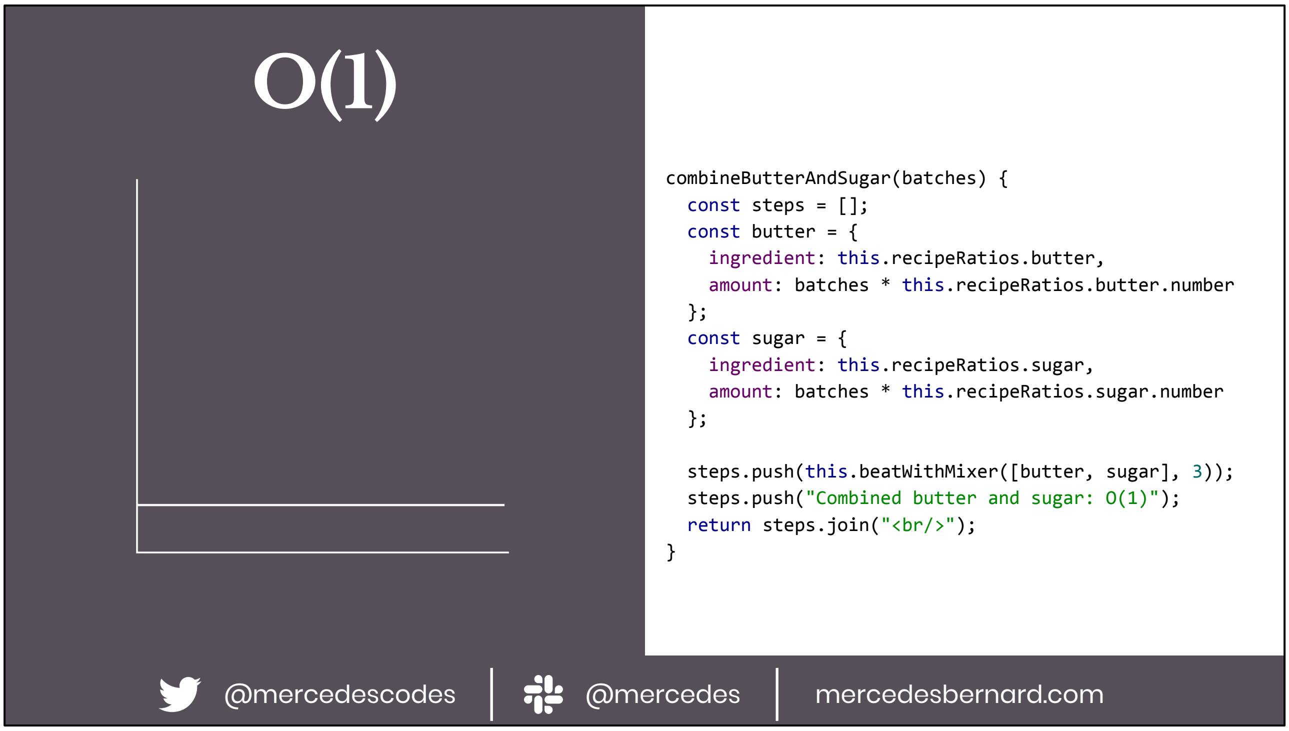
@mercedescodes | @mercedes | mercedesbernard.com

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.

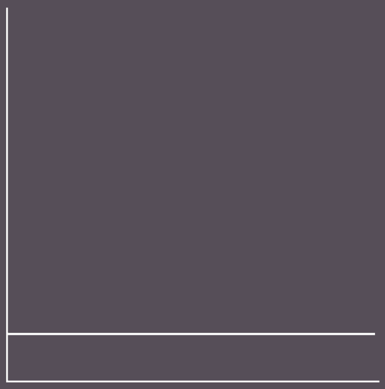
Big O Notation measures relative complexity of a function or algorithm. Most often this is measuring running time but it could also be used to measure memory consumption, stack depth, and other resources. We're going to focus on running time since in an interview, that's what they're usually asking about.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



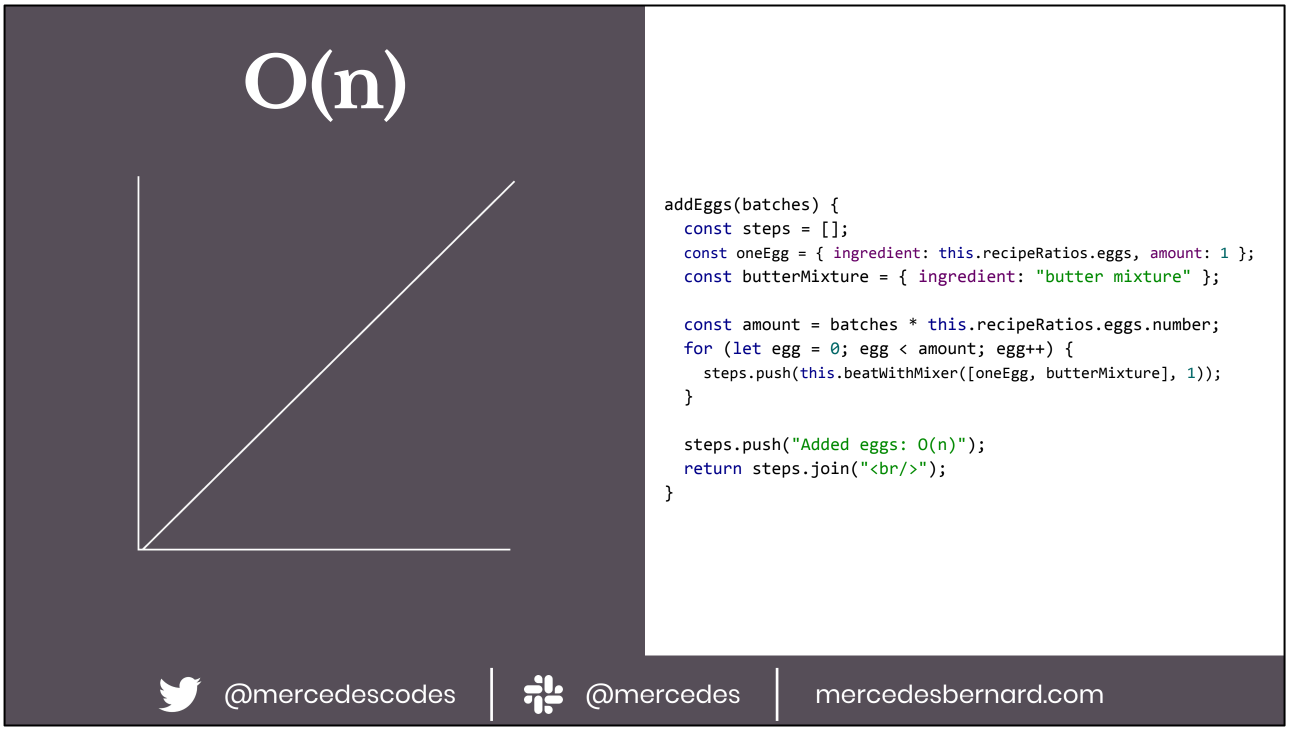
O(1)



```
combineButterAndSugar(batches) {
  const steps = [];
  const butter = {
    ingredient: this.recipeRatios.butter,
    amount: batches * this.recipeRatios.butter.number
  };
  const sugar = {
    ingredient: this.recipeRatios.sugar,
    amount: batches * this.recipeRatios.sugar.number
  };

  steps.push(this.beatWithMixer([butter, sugar], 3));
  steps.push("Combined butter and sugar: O(1)");
  return steps.join("<br/>");
}
```

@mercedescodes | @mercedes | mercedesbernard.com

O(1) = Constant running time regardless of input size

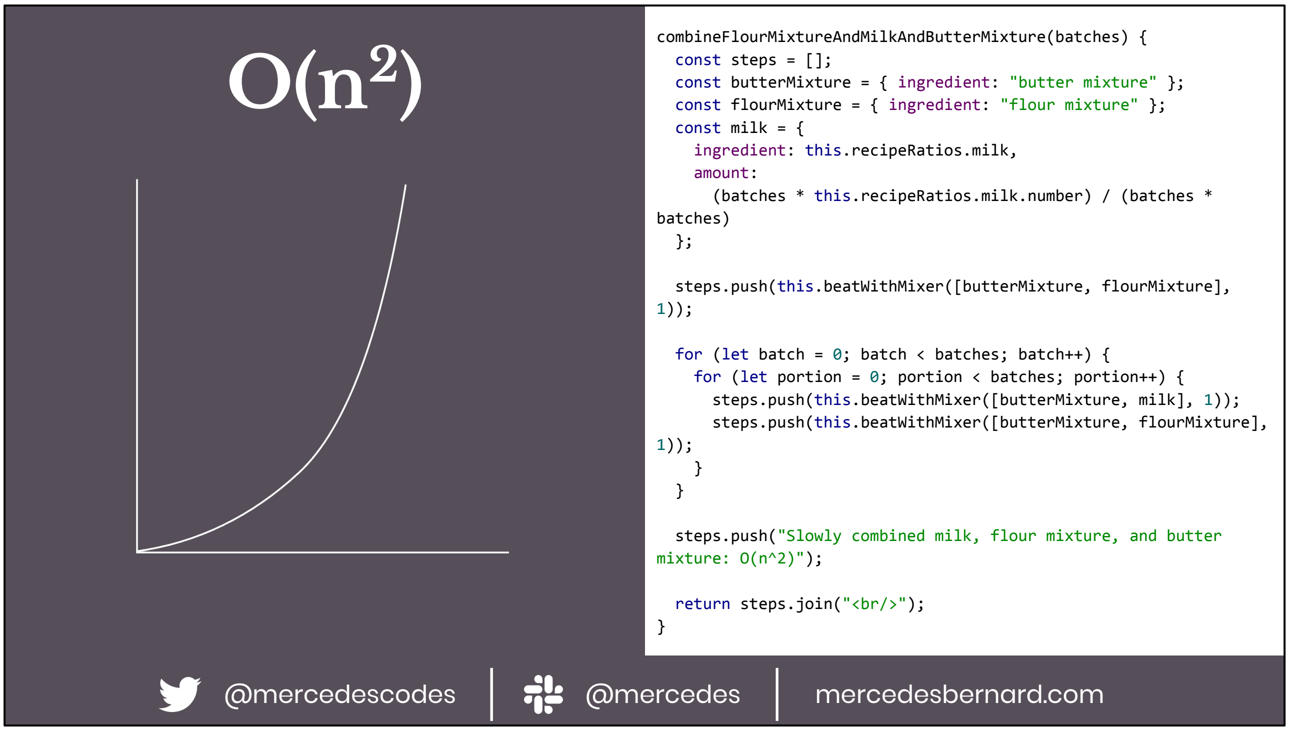


O(n)

```
addEggs(batches) {
  const steps = [];
  const oneEgg = { ingredient: this.recipeRatios.eggs, amount: 1 };
  const butterMixture = { ingredient: "butter mixture" };

  const amount = batches * this.recipeRatios.eggs.number;
  for (let egg = 0; egg < amount; egg++) {
    steps.push(this.beatWithMixer([oneEgg, butterMixture], 1));
  }

  steps.push("Added eggs: O(n)");
  return steps.join("<br/>");
}
```

@mercedescodes | @mercedes | mercedesbernard.com

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

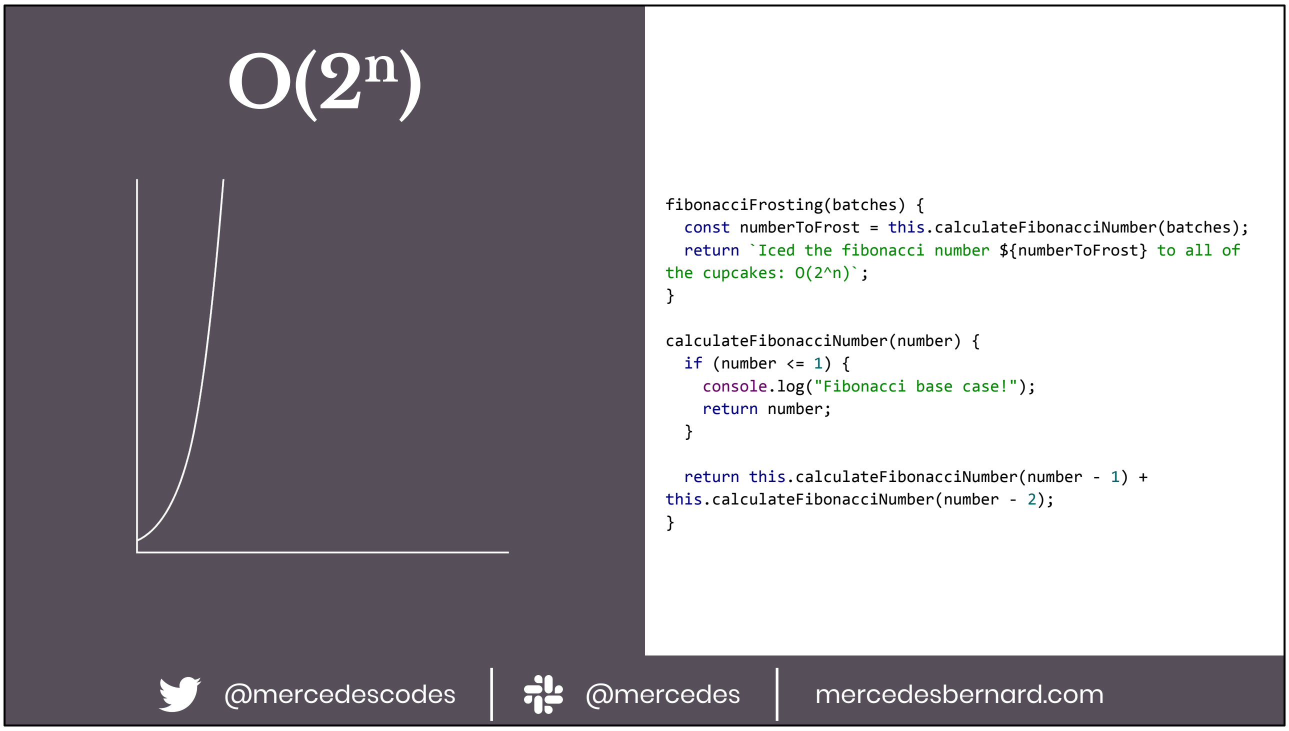


$O(n^2)$

```
combineFlourMixtureAndMilkAndButterMixture(batches) {
  const steps = [];
  const butterMixture = { ingredient: "butter mixture" };
  const flourMixture = { ingredient: "flour mixture" };
  const milk = {
    ingredient: this.recipeRatios.milk,
    amount:
      (batches * this.recipeRatios.milk.number) / (batches *
batches)
  };

  steps.push(this.beatWithMixer([butterMixture, flourMixture],
1));

  for (let batch = 0; batch < batches; batch++) {
    for (let portion = 0; portion < batches; portion++) {
      steps.push(this.beatWithMixer([butterMixture, milk], 1));
      steps.push(this.beatWithMixer([butterMixture, flourMixture],
1));
    }
  }

  steps.push("Slowly combined milk, flour mixture, and butter
mixture: O(n^2)");

  return steps.join("<br/>");
}
```

@mercedescodes | @mercedes | mercedesbernard.com

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



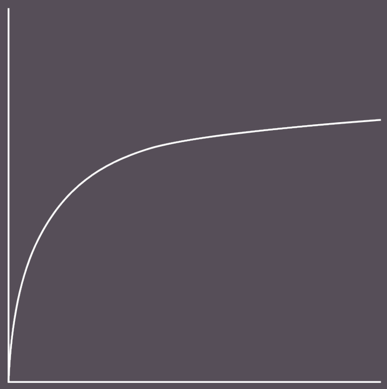
O(2^n)

```
fibonacciFrosting(batches) {
  const numberToFrost = this.calculateFibonacciNumber(batches);
  return `Iced the fibonacci number ${numberToFrost} to all of
the cupcakes: O(2^n)`;
}

calculateFibonacciNumber(number) {
  if (number <= 1) {
    console.log("Fibonacci base case!");
    return number;
  }

  return this.calculateFibonacciNumber(number - 1) +
this.calculateFibonacciNumber(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(logn)



Divide and conquer algorithms such as binary search

O(log n) = This is kinda the opposite of O(2^n).

Big O Uses

@mercedescodes | @mercedes | mercedesbernard.com

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

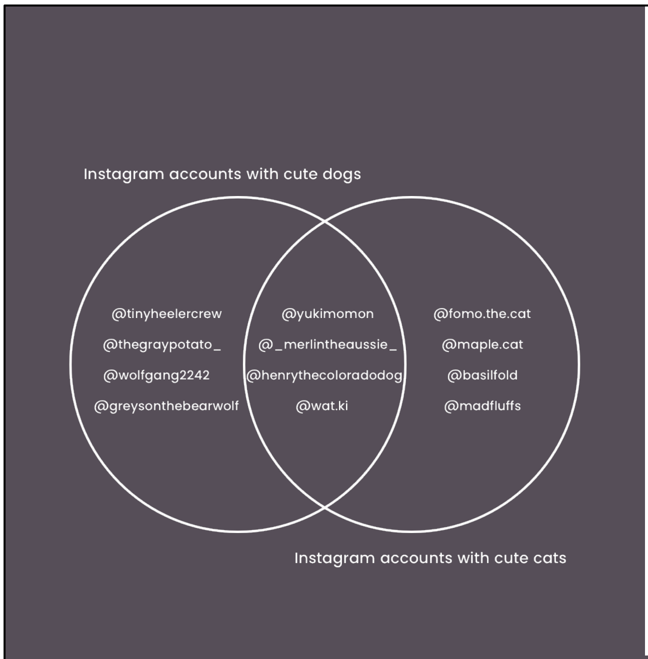
Agenda

1. Concepts

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

2. Principles of OO programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism



Set theory

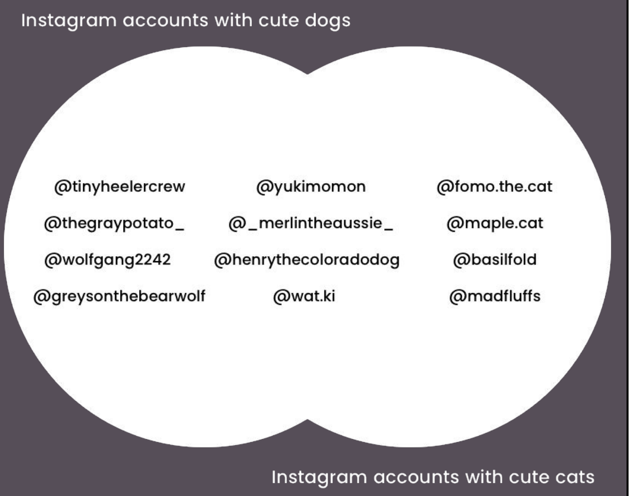
@mercedescodes | @mercedes | mercedesbernard.com

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

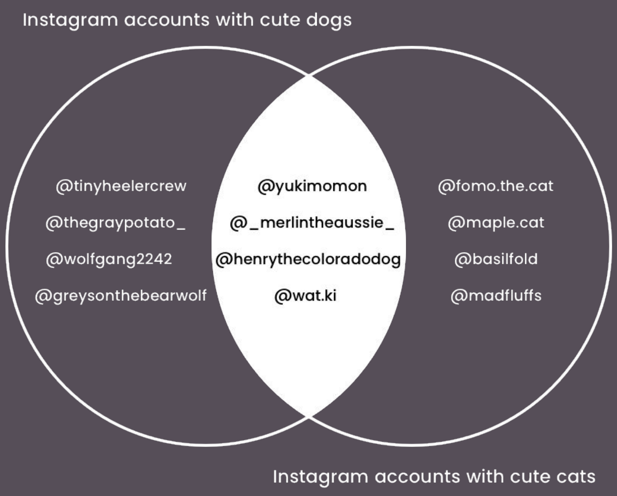


@mercedescodes | @mercedes | mercedesbernard.com

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

Intersection: X ∩ Y

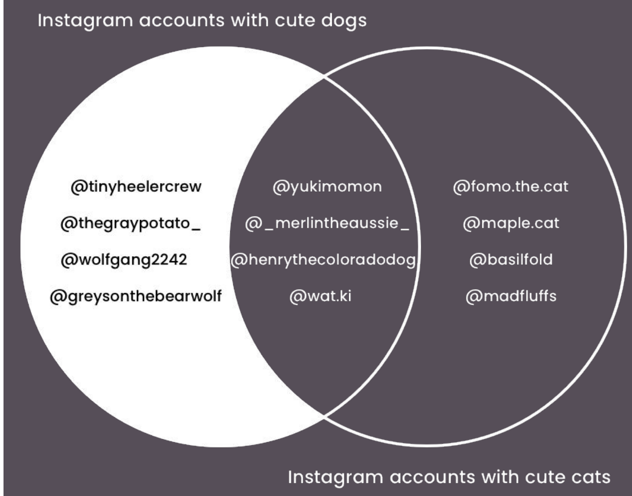
Accounts featuring cute dogs and cats



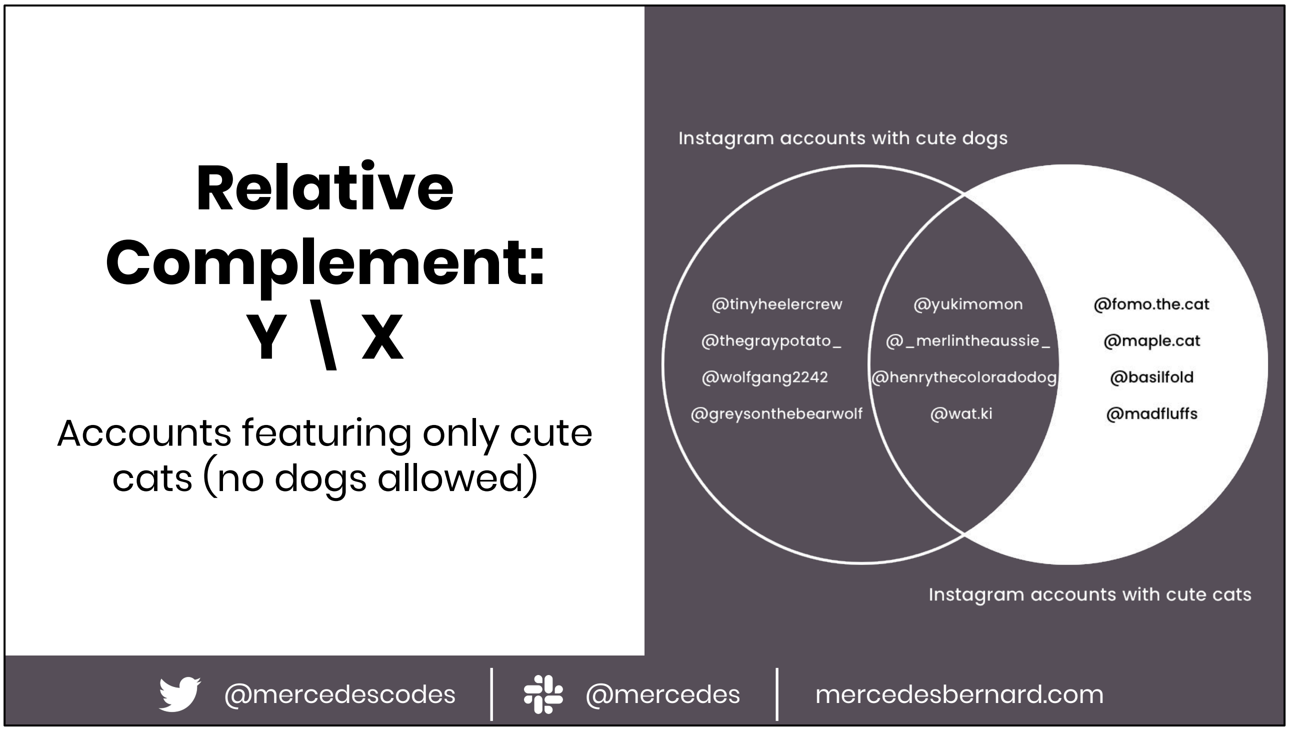
Intersection X ∩ Y - The stuff that exists in X AND Y

Difference: X - Y

Accounts featuring only cute dogs (no cats allowed)

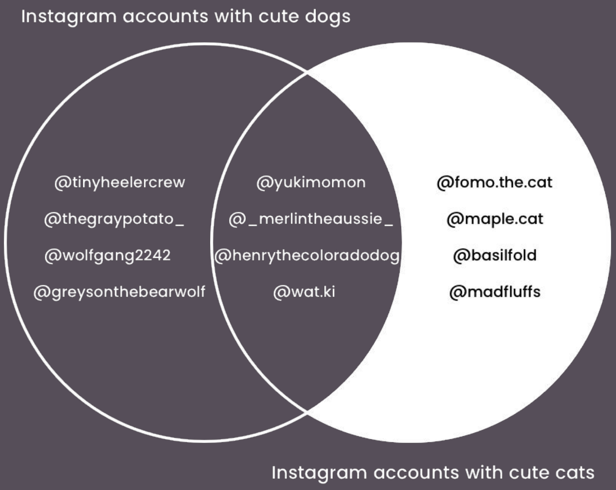


Difference - X - Y The stuff that only exists in X



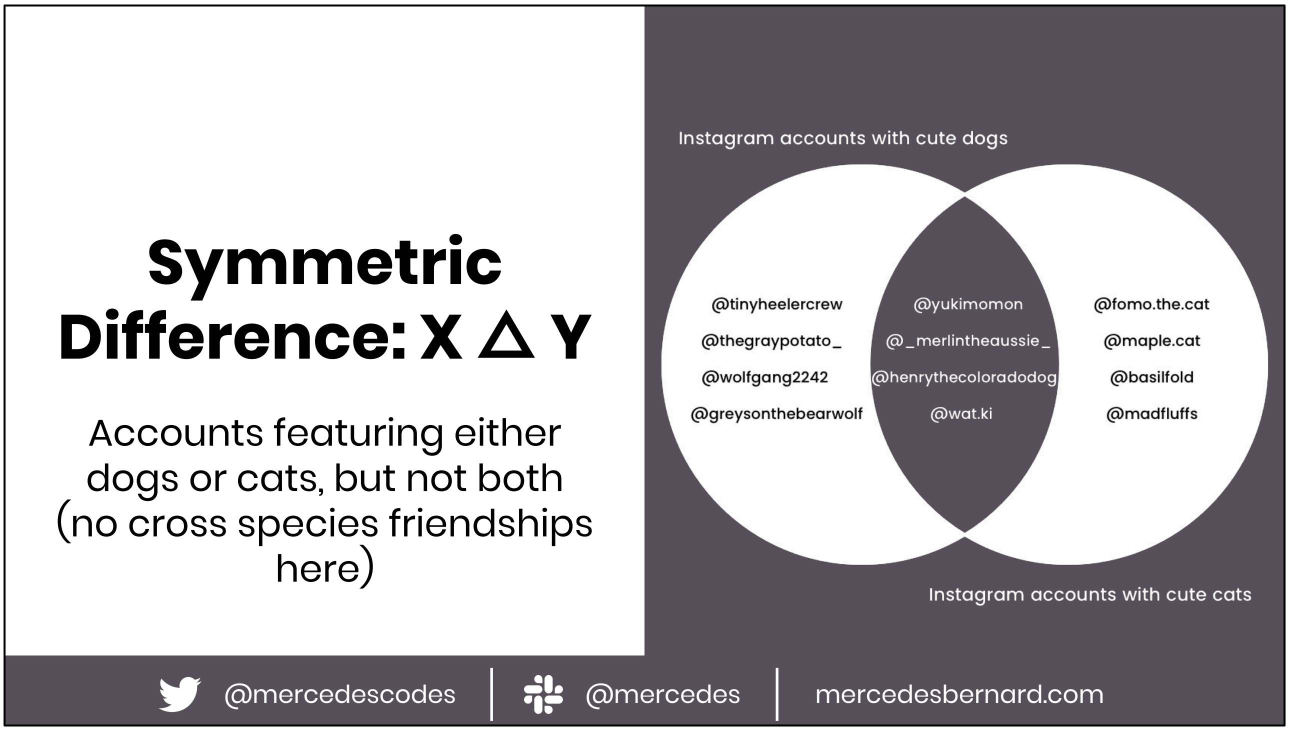
Relative Complement: Y \ X

Accounts featuring only cute cats (no dogs allowed)



@mercedescodes | @mercedes | mercedesbernard.com

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



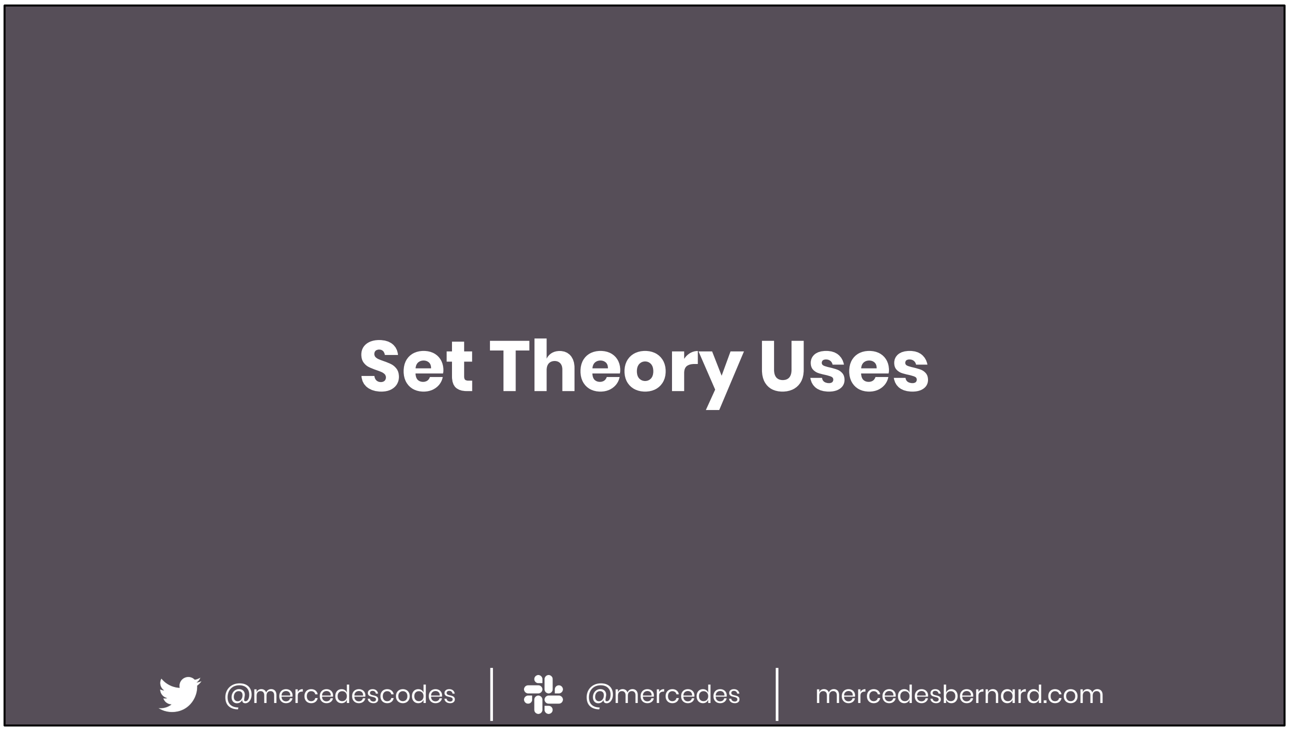
Symmetric Difference: X △ Y

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

@mercedescodes | @mercedes | mercedesbernard.com

Set theory is the foundation of relational databases
Website filters

Agenda

1. **Concepts**
    - ~~Big O notation~~
    - ~~Set theory~~
    - Recursion
2. **Principles of OO programming**
    - Encapsulation
    - Abstraction
    - Inheritance
    - Polymorphism

Recursion

@mercedescodes | @mercedes | mercedesbernard.com

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

@mercedescodes | @mercedes | mercedesbernard.com

Navigation paths on a website

Agenda

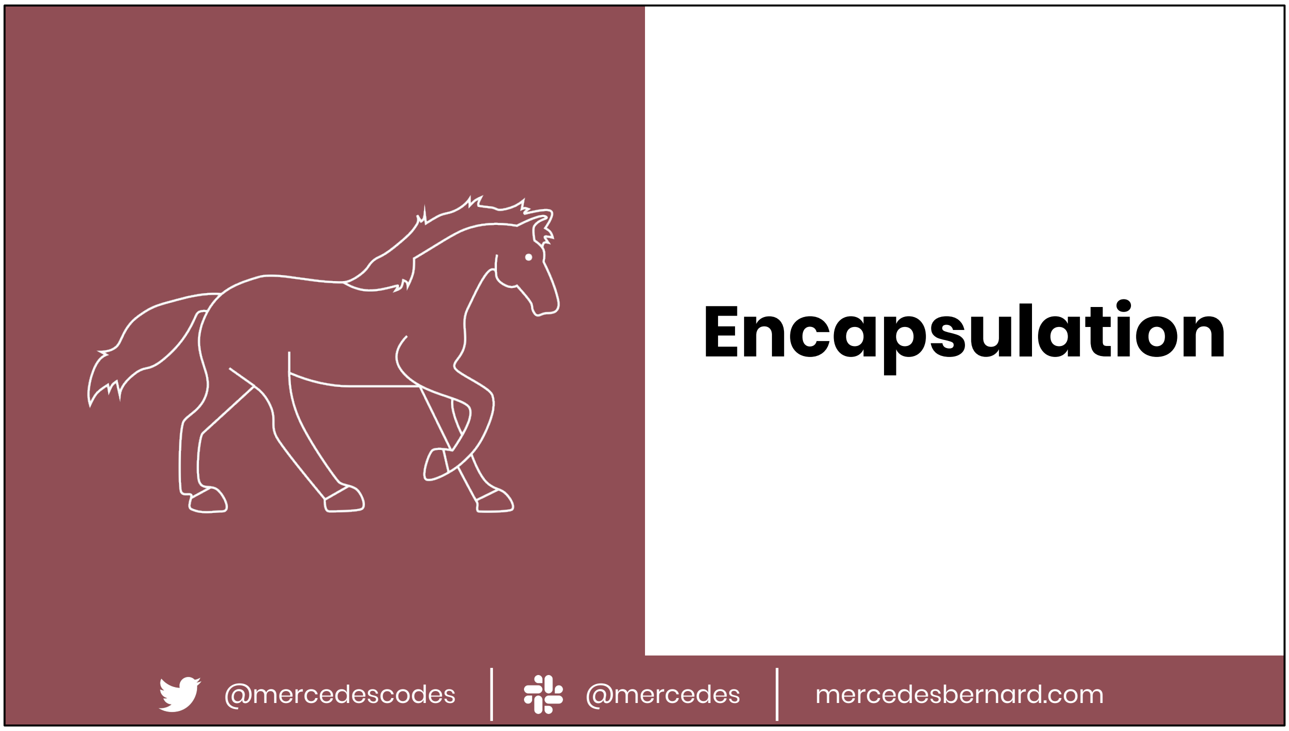
1. Concepts

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

2. Principles of OO programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Principles



Encapsulation

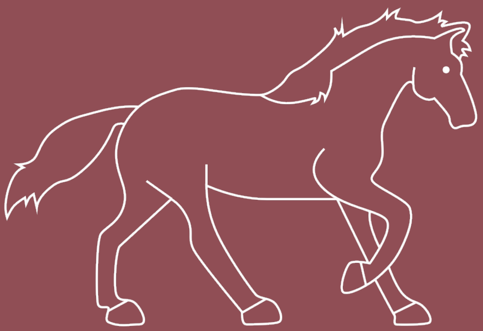
@mercedescodes | @mercedes | mercedesbernard.com

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.

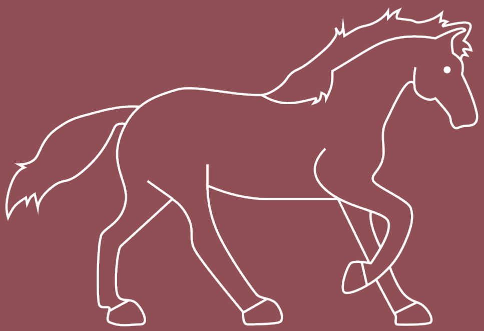
Walking: 4 beat pace, left hind leg, left front leg, right hind leg, right front leg
Trot: 2 beat pace, left hind leg + right front leg, right hind leg + left front leg
Canter: 3 beat pace, left hind leg, right hind leg + left front leg, right front leg
Gallop: 4 beat pace, left hind leg, right hind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

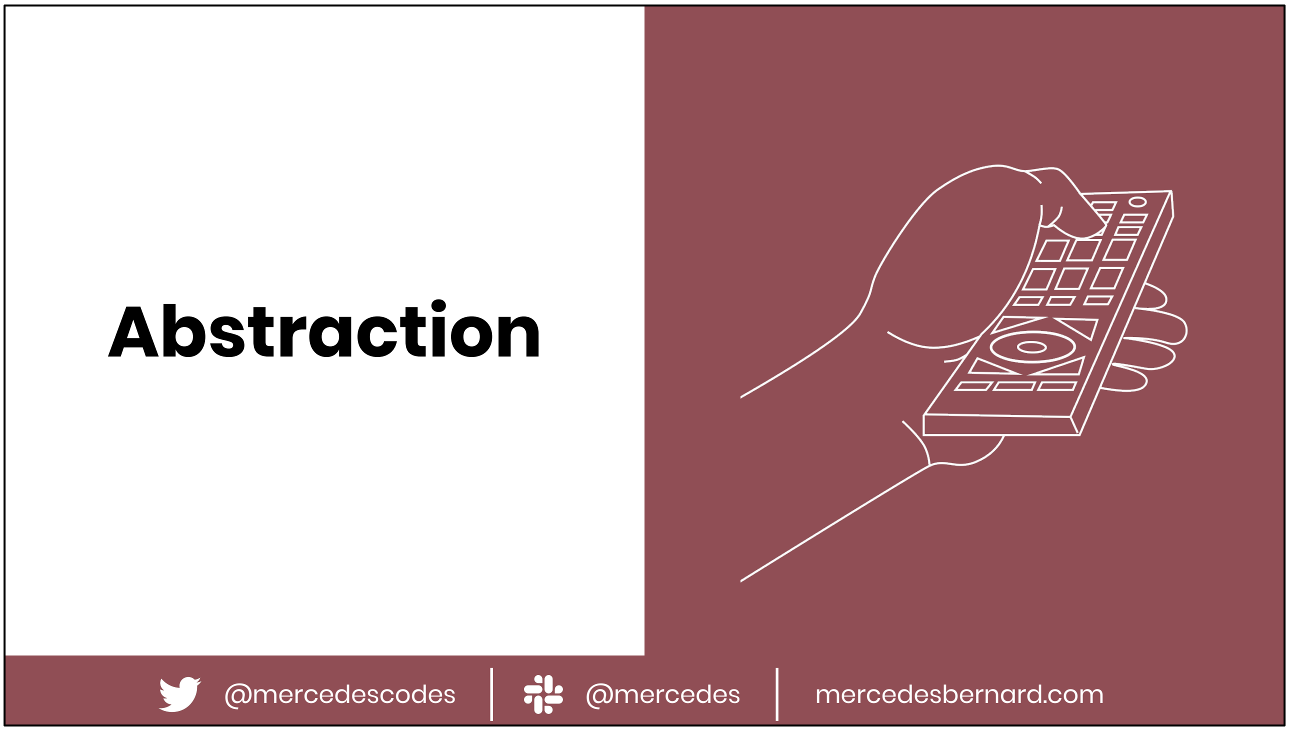
Agenda

1. Concepts

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

2. Principles of OO programming

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



Abstraction

@mercedescodes | @mercedes | mercedesbernard.com

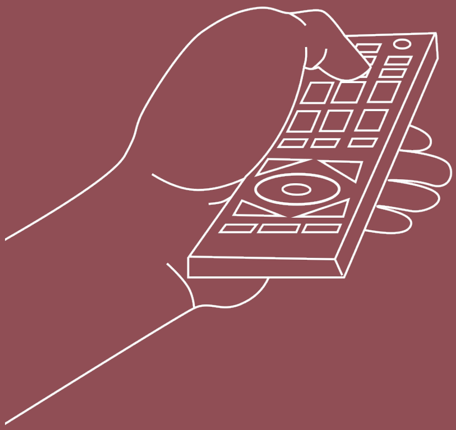
When you push a button on a remote control, something happens on your TV. The volume goes up, the channel changes, etc. There are few, finite things you can do with a remote. And you don't need to know how it works in order to make your change happen. For example, you don't need to know how infrared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

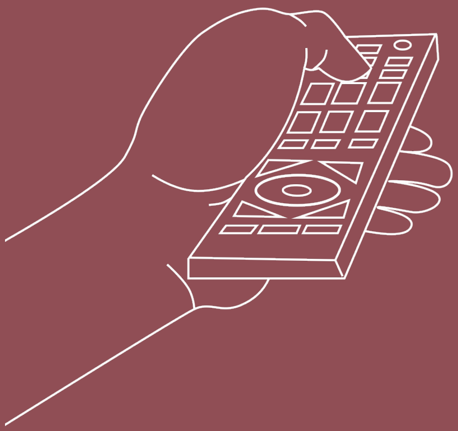
Abstraction is an extension of encapsulation. Abstraction refers to hiding all the internal implementation details of a class and providing very few, clear mechanisms for other objects in the code to interact with each other.

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. Principles of OO programming

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

Inheritance

@mercedescodes | @mercedes | mercedesbernard.com

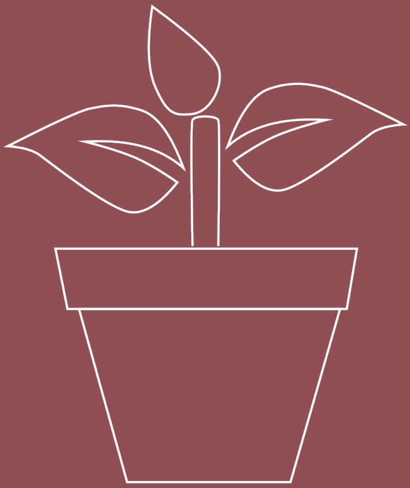
All garden plants need sun, soil, and water to survive. And if you want to learn how to garden, you'll be reading about the plant's care instructions that describe how to help your plant stay healthy and thrive.

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.

```
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());
```

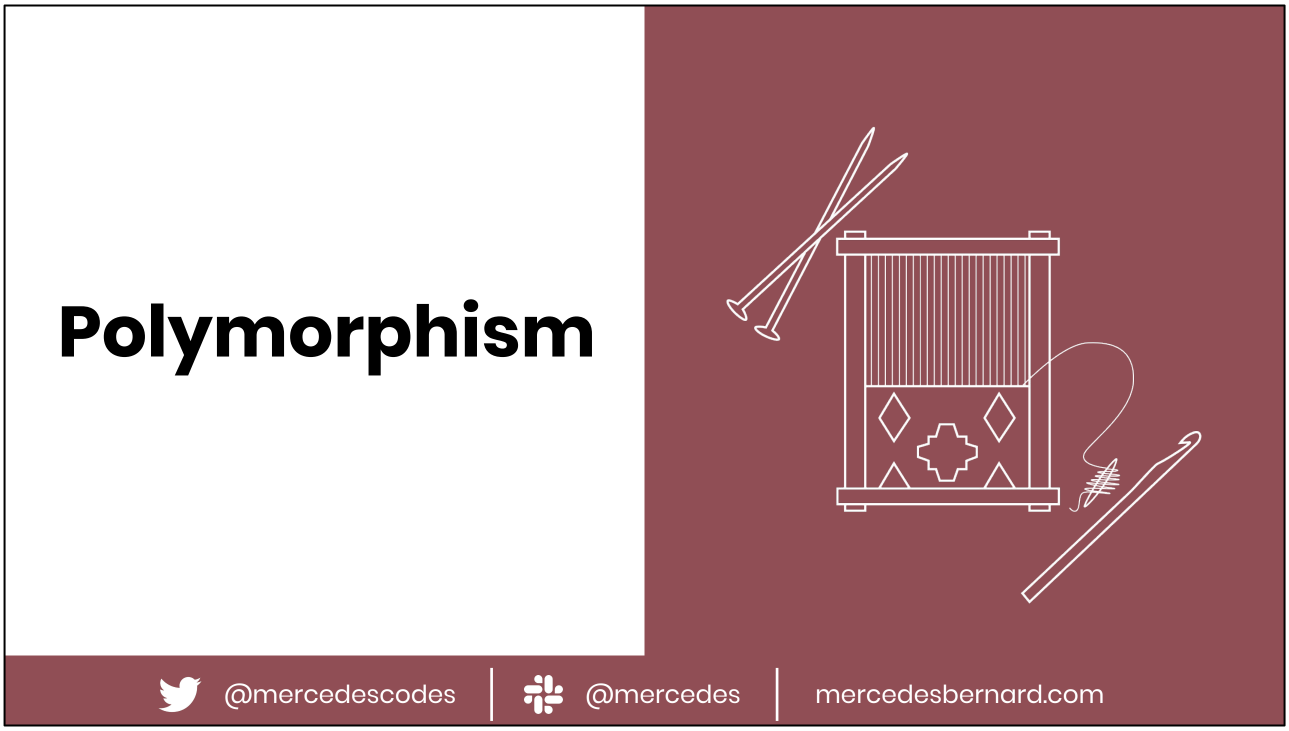
Agenda

1. Concepts

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

2. Principles of OO programming

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



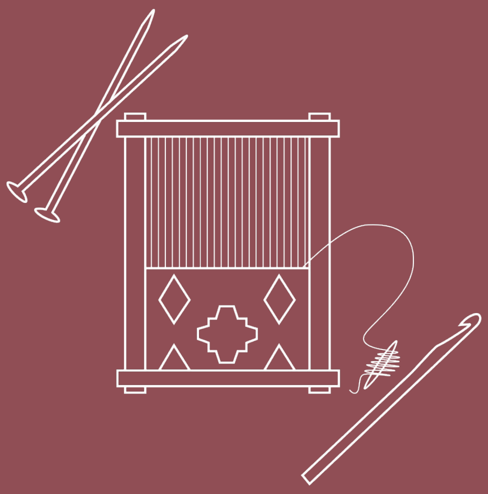
Polymorphism

@mercedescodes | @mercedes | mercedesbernard.com

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).

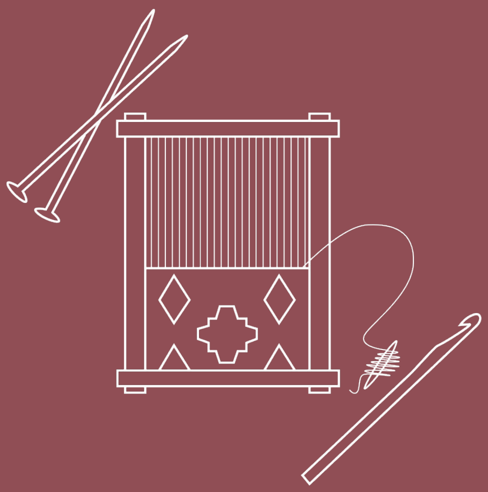
```
createFabric(numberOfRows) {
  let fabric = [];
  for (let row = 0; row < numberOfRows; row++) {
    if (row % 2 === 0) {
      fabric.push(this.stitchRow(Knit));
    }
    if (row % 2 > 0) {
      fabric.push(this.stitchRow(Purl));
    }
  }
  return fabric;
}

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;
}
```



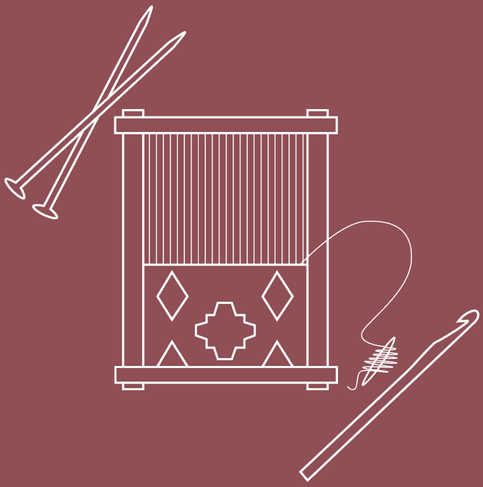
@mercedescodes | @mercedes | mercedesbernard.com

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.

```
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;
}
```

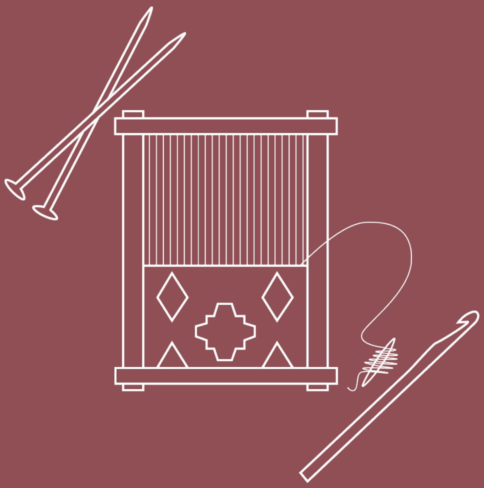


```
createFabric(numberOfRows) {
  let fabric = [];
  for (let row = 0; row < numberOfRows; row++) {
    fabric.push(this.stitchRow());
  }
  return fabric;
}

stitchRow() {
  let row = "";
  for (let stitch = 0; stitch < this.rowLength; stitch++) {
    row += this.singleCrochet();
  }
  row += "Turn.\n";
  return row;
}
```



@mercedescodes | @mercedes | mercedesbernard.com

```
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. Principles of OO programming

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

mercedesbernard.com/speaking

@mercedescodes | @mercedes | mercedesbernard.com

Thank you

@mercedescodes | @mercedes | mercedesbernard.com