Day 0/1

Monday, September 7, 2020 2:42 PM

A package is a collection of files and configuration wraps in an easy to distribute wrapper.
Any package that your application relies on to work is referred to as a dependency.
Package managers are applications that accept your code bundled up with some important meta-data and provide services like versioning, change management and even tracking how many projects are using your code.
Most package managers consist of at least two parts, a command line interface, and a registry.
NPM stands for node package manager.

npm will show npm's help info, including some common commands and how to access more detailed guides.
npm init will set your current project directory up for npm.
This requires answering a few questions to generate a package.jsonfile, a critical part of npm's dependency management functionality.npm install will download and install a package into your project.
You can use the-g(or--global) flag to install a package for use everywhere on your system.
To have some fun, run npm install -g cowsay.
Once thedownload's completed, try runningcowsay Hello, world

Mini package managers including NPM have the ability to resolve correct dependency versions, this means the manager can compare all the packages used by an application and determine which versions are most compatible.
NPM accomplishes this dependency resolution process using both the package.json and package-lock.json files.
Package-lock.json is commonly known as a lock file in package manager parlance.
When the NPM CLI utility installs a package it adds to the node modules subdirectory in your project.
Each package will be placed in a directory named after itself and contain the role code for the package along with any associated package.Json as well as documentation.
The node_modules file is special for a few reasons:
It's a great way to keep dependencies separated for each project.
By keeping node modules for each project separate you can have as many different versions of each package as you like each project has a specific version it needs right on hand.

Because of the amount of space they take up it is good practice to keep node modules out of get repository's or other version control.

Semantic.version.number (Often abbreviated as semvar) : is a way of tracking version numbers that lets other developers know what to expect from each release of your package.

Semantic version numbers are made up of three parts each numbered sequentially in with no limit on how large they can be, the leftmost digit in a semvar is the most significant meaning that
1.0.0 > 0.8.99

Here's a high-level overview:

2.1.6

Major Minor Patch

Major changes should be considered breaking.
They will be incompatible with other major versions of the same package and may require significant changes in any app that depends on them. Creating a sequel to a hit video game would be a major change.

Minor changes generally represent new features. These shouldn't totally break anything, but might require a little tweak to keep dependent apps up-to-date. Adding a new level to a video game would be a minor change.

Patch-level changes are for fixing bugs or small issues. These shouldn't break any other functionality or force dependent apps to make any changes themselves. Fixing a typo in a video game's instructions would be a patch-level change

When adding a new dependency to your package.jsonfile, you can designate a range by adding some special characters to your version number:

- *indicates "whatever the latest version is" >
- 1.0.0indicates "any version above major version
- 1".^1.0.0indicates "any version in the 1.x.x range".
- ~1.0.0indicates "any patch version in the 1.0.x range"
- .1.0.0indicates "exactly version 1.0.0".

- `*` indicates "whatever the latest version is".
- `>1.0.0` indicates "any version above major version 1".
- `^1.0.0` indicates "any version in the 1.x.x range".
- `~1.0.0` indicates "any patch version in the 1.0.x range".
- `1.0.0` indicates "exactly version 1.0.0".

https://semver.npmjs.com/

You may also omit consecutive trailing zeroes, so:
^1.0.0 is the same as ^1.0 or just ^1.~1.0.2----------("any patch version greater than 1.0.2") would need to written out in its entirely, though.
You should consider the numbers in your semver to be a minimum value, ----------
so~2.1.3 would include2.1.4, but not2.1.2.

while npm helps manage your dependencies, it won't automatically keep them up to date!

version range you've set in your package.json.

Using npm to Perform Common Tasks -Part One:

NPM is a CLI and a registry (database of packages available, their locations, their versions and who maintains them)

Using npm to manage npm :
To confirm if you have the npm CLI installed, you can run the command npm --version
Or npm -v.

If you have the npm CLI installed, you'll see its version number displayed in the console.

If you don't have the npm CLI installed, you receive an error.

The npm CLI is available as an npm package, which allows you to use the npm CLI to update itself.

If you're using macOS or Linux, you can update the npm CLI to the latest version by running the following command :
npm install -g npm@latest

If you installed Node.js using the default installer, you might need to prefix the above command with sudo like this:

sudo npm install -g npm@latest

The sudo command allows you to run a command with the security privileges of another user, typically your computer's administrator or super user account.

When using the sudo command, you'll be prompted for your account's password.

Any node.js project that contains a package.json file is technically a NPM package the most of these projects will never be published at the NPM registry for consumption by the general development community.

When selecting a package, it's helpful to ask yourself the following questions:

- **Does the package do what I need?**Most packages in the npm registry will include some documentation on how to use the package. Usually you can review that documentation to determine if the package will suit your needs. Sometimes, you might need to review any additional documentation that's available in the package's code repository (i.e. GitHub, GitLab, or wherever the package's source code lives). Alternatively, you can install the package into a throwaway project to safely test it in a sandbox environment.
- **How popular is the package?**Popularity isn't always everything, but it can be an effective way to determine if a package is useful and reliable. It also increases the likelihood that other developers on your team might have experience with using a particular package.
- **Is the package being maintained?**If you're going to take a dependency on a package, you need to have confidence that the package is actively being maintained by its developer(s). To do that, review the package's associated code repository (i.e. GitHub, GitLab, etc.) Have there been recent commits and recent releases? Review the repository's issues to see if consumers of the package are getting their questions answered and bugs are being fixed.

Using a dependency in code:

Dependency type:
npm tracks two types of dependencies in the package.jsonfile:

Dependencies (dependencies) - These are the packages that your project needs in order to successfully run when in production (i.e. your application has been deployed or published to a server that can be accessed by your users).
Development dependencies (devDependencies) - These are the packages that are needed locally when doing development work on the project. Development dependencies often include one or more tools that are used to build and test your application.
There are actually three additional types dependencies that npm can track including peer dependencies (peerDependencies), bundled dependencies(bundledDependencies), and optional dependencies(optionalDependencies).

Installing a development dependency:
To install a development dependency, you simply add the--save-dev flag:
npm install mocha --save-dev

The--save-devflag causes npm to add the package to the devDependencies field in the package.jsonfile:
{"dependencies":{"colors":"^1.4.0"},

Installing a development dependency

To install a development dependency, you simply add the `--save-dev`flag:

```
npm install mocha --save-dev
```

The `--save-dev` flag causes npm will add the package to the `devDependencies`field in the `package.json`file:

```
{
  "dependencies": {
    "colors": "^1.4.0"
  },

  "devDependencies": {
```

VIDEO 1 - INTRO TO BUILT-IN NODE PACKAGES

Packages and objects of interest

- Console: the console object
- File System: reading and writing files
- Path: manipulating file paths
- Process: object representing the running program
- Readline: reading data from somewhere (like a file) line by line

Console Methods

console.log

- prints to terminal
- comes out on STDOUT steam
- can be directed elsewhere (node console.js > output.txt)

console.log

- prints to terminal
- comes out on STDERR stream

File System Methods

Requiring Packages

- const packageName = require("packageName");
- can name variable whatever you want

Methods: fs.appendFile(fileName, content, errorHandler)

- fileName: path to / name of file you want to create
- content: content of file
- errorHandler: callback that handles any errors

Methods: fs.readFile(pathToFile, contentType, errorHandler)

- pathToFile: path to file you want to read
- contentType: type of data in file
- callback: callback that is invoked after file is read
  - two args: error and data that was read

Methods: fs.readdir(dirName, callback)

- reads files in directory
- NOTE: THESE METHODS ARE ASYNC
  - callbacks invoked in order of completion

Method: fs.createReadStream(pathToFile, dataType);

PATH

__dirname

- global var that contains directory name of current files path
- "dunder dirname"

PROCESS

process.argv

- returns all args typed into command line
- node process.js
- node process.js arg3 arg4

process.cwd()

- returns directory from where you ran executable

process.exit()

- exits program
- can provide exit code, default is 0 (success)

READLINE

- SEE CODE

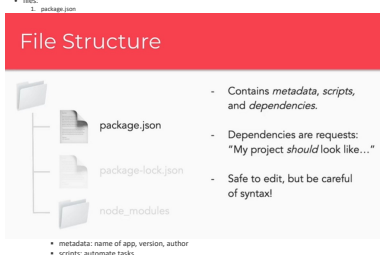
VIDEO 2 - INTRO TO NPM

Overview

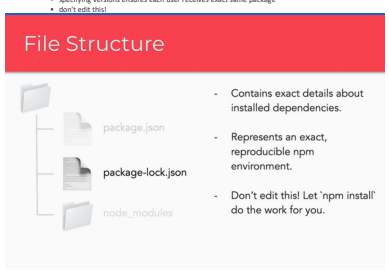
- NPM: default package manager for JavaScript runtime environment, Node.js
- consists of:
  1. CLI tool (ex: npm init, npm install)
  2. online database of public & paid-for packages

File Structure

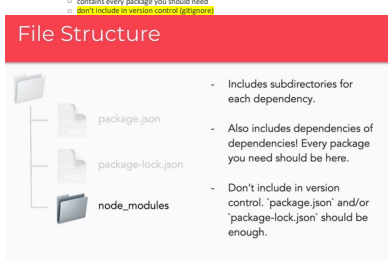
- npm requirements must be at root level of application
- files:
  1. package.json



  - metadata: name of app, version, author
  - scripts: automate tasks
  - dependencies: packages app relies on
  - safe to edit!
2. package-lock.json
  - contains exact details about installed dependencies (versions, etc.)
  - represents an exact, reproducible npm environment
  - specifying versions ensures each user receives exact same package
  - don't edit this!

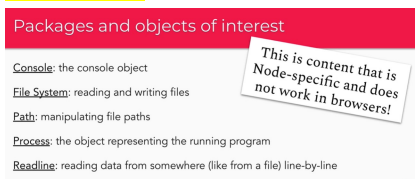


3. node_modules
  - contains subdir for each dependency
    - includes dependencies of dependencies
    - contains every package you should need
    - don't include in version control (gitignore)



VIDEO 3 - IGNORING THINGS IN GIT

Intro to Built In Node Packages:



INTRO TO NODE PACKAGE MANAGER (NPM)

Overview
- NPM: default package manager for JavaScript runtime environment, Node.js
- consists of:
  1. command line client (npm)
  2. online database of public & paid-for packages

Package Management
- package is collection of files & configurations wrapped up
- can rely on work of other developers to move our projects along
- can create our own packages and share with the world
- dependencies: packages our app depends on

Package Managers
- apps that accept code and provide services like versioning, change management, etc
- const of at least two parts:
  1. command line interface (CLI)
     - enables us to download, install/uninstall packages
  2. registry
     - database of package information

JavaScript Package Management
- before npm: embedded script tags used to share code
- Node.js released in 2010 with npm
  - most widely used package manager

Getting Started with NPM
- don't have to install since its part of Node.js
- npm init: generates package.json file
- npm install [-g] <package-names>: installs package into your project
  - optional -g (--global) flag installs package globally (everywhere in system)

NPM AND DEPENDENCY MANAGEMENT

```
// console.log
console.log('this is a log statement');


// console.error
console.error('this is an error');


// console.table
const dogs = [
    { name: 'Bodhi', age: 5 },
    { name: 'Lucy', age: 10 }
]
console.table(dogs);


// console.group || console.groupEnd
console.log('DOGS');
console.group('dog group')
console.log('bodhi');
console.log('lucy');
console.groupEnd();
console.log('CATS');
console.group('cat group');
console.log('pancake');
console.log('bob');
console.groupEnd()



const fs = require('fs');

// // fs.appendFile: create or append
// fs.appendFile('./new_text.txt', 'here is some text', err => {
//  if (err) {
//      console.error(err);
//  }
// })


// // fs.readFile: reads aall
// fs.readFile('./poem.txt', 'utf-8', (err, data) => {
//  if (err) {
//      console.error(err);
//  }
//  console.log(data);
// })


// // fs.readdir: reads a directory
// fs.readdir('./my-dir', (err, files) => {
//  if (err) {
//      console.error(err);
//      return
//  }
//  files.forEach(file => {
//      console.log(file);
//  })
// })


// fs.createReadStream: read file piece by piece
const stream = fs.createReadStream('./poem.txt', 'utf-8');
stream.on('data', chunk => {
    console.log(chunk);
})
stream.on('close', () => {
    console.log('done reading stream')
})



const path = require("path");

// __dirname global - "dunder dirname"
console.log(__dirname);


// path.join
const newPath = path.join('/Users', 'Alissa', 'notes.text');
console.log(newPath)


// path.dirname: returns directory name of path
console.log(path.dirname(newPath))


// path.extname: returns just extension
console.log(path.extname(newPath))


// path.parse: returns obj containing diff pieces
console.log(path.parse(newPath))



const process = require('process');

// process.argv: all args on command line that were typed
console.log(process.argv)


// process.cwd(): shows current working directory
console.log(process.cwd())

// process.exit(0) : way to exit program
process.exit()


const fs = require('fs');
const readline = require('readline')


// create readable stream
const stream = fs.createReadStream('./poem.txt')
// create instance of readline interface
// - accepts readable stream as argument
const rl = readline.createInterface(stream)

// read stream line by line
rl.on('line', line => {
    console.log(line, "...");
})
```

Object-Oriented Programming Explained

Classes vs objects

In strictest terms, an **object class** is the definition of an object, and an **object instance** is a use of that object. Quite often developers loosely use the generic term "object" to refer to either or both.

Or sometimes "object" means a data structure with key-value storage represented by curly-braces (a.k.a. POJO - Plain Old JavaScript Object).

You will quickly learn to tell the difference based on the context. For example, "object" means "object class" whenever discussing how data is stored and changed within the object (that is, the properties & methods);

and "object" means the "object instance" (or "instance of an object") whenever speaking of a specific, individual use of the object class.

Encapsulation: enclose (something) in or as if in a capsule

When you write a class, : you put behavior (knows as methods) and the data it works on (known as **properties** or **instance variables**, or **fields** or **members**) together.

With classes, you can deal with code that declares data structures in the same place as code that modifies hem.

Without classes, a programmer might have to deal with code that declares data structures in one file and use them in multiple other files all over the code base. Understanding where data got changed becomes exponentially difficult as the size of software grows - UNLESS classes are used to "black box" the changes to a specific structure all in one central place.

Knowing where data changes is one of the most important aspects of software.

Abstraction

Some programming literature will talk about a fourth pillar of Object-Oriented Programming: abstraction.

Abstraction is taking a higher-level view of the problem. Backing up from it to look at the bigger, more general picture. For example, if you were going to build multiple classes that behave like lists, then you would likely want to give them the same properties and methods. Not only does that follow the pillar of **abstraction**, it will help you remember how to use them.

Abstraction focuses on what and object does, encapsulation focuses on how it does it. Both center around reducing complexity by hiding details. This is why they may sometimes be considered one pillar.

Private, public and protected

It is important to know that some languages allow developers to mark the properties and methods of a class as "private". This allows the class they are in to use them, but blocks access from other classes or code within the system. The opposite of private is "public", meaning the property or method is accessible. A special case is "protected" where a class and all its children (see inheritance below) have access, but separate classes or code do not.

Using private and protected methods is one way encapsulation shows up in the code. The other way is hiding complexity inside methods.

JavaScript does NOT fully support private, protected or public flags on its classes, so you do not need to worry about this aspect of OOP just yet.

Thought experiment: the vending machine

Imagine, if you will, that you want to buy something from a vending machine. You tap your phone against the payment reader (or insert bills or coins into a slot, or swipe a credit card). Once the payment is authorized in any of those forms, you can make a selection and receive your tasty treat. Thinking about that as a series of steps, you could write them like this.

1. Authorize payment.
2. Make selection.
3. Retrieve tasty treat.

With respect to the concepts of object-oriented programming, the vending machine would be an object. All of the internal workings of the machine, how it communicates wirelessly with a payment vendor or counts cash, how its internal mechanics thrash about to deliver a beverage or snack to you, all of that is hidden behind a plastic advertisement. All of that behavior is *encapsulated* (and *abstracted*) inside the machine so that you don't have to worry about the details. Imagine a less-encapsulated world where you had to perform the following steps to get your tasty treat.

4. Call your payment provider.
5. Specify that you want to spend no more than $2 on your next purchase.
6. Write down a confirmation number for an authorization of up to a $2 payment.
7. Call the vending machine company.
8. Give them your payment authorization confirmation number.
9. Key in a 16-digit authorization number that they tell you.
10. Make selection.
11. Tell the vending company the transaction number and the total amount.
12. Retrieve tasty treat.

Now, instead of hiding all the details about how payments work, this imaginary vending machine is forcing you to have to participate in the payment process. A system such as this relies on people acting as good agents when they report the total amount with the transaction number. A system such as this is inconvenient for the person making the purchase.

However, since the actual vending machine *encapsulates the complexity of its internal mechanisms behind easy-to-use interactions*, users return to it again and again. The same goes with encapsulation with object-oriented programming.

Thought experiment: gym memberships

Say that you wanted to write software that helped gyms manage their business. Something that can happen at the gym would be that you want to **register a new person as a member of the gym**. In order to take this action a few bits of information are required, such as

- First name
- Last name
- Email
- Credit card info

A function to handle the registration might look something like `register(firstName, lastName, email, creditCardInfo)`.

The code that calls this doesn't need to know or care how it works. It only needs to know how to get the function to work. Therefore, this is a perfect candidate for **encapsulation**.

```
class Member {
    constructor(firstName, lastName, email) {
      // store first & last name and email
    }
    register(creditCardInfo) {
      // process credit card info (securely)
      // if successful, enable membership
    }
}

// Lots of other code
```

Additionally, a `Gym` object seems like a great place to handle adding a new `Member` to the gym and storing the list of previously registered members.

```
class Gym {
    constructor() {
      this.members = []
    }
    registerNewMember(firstName, lastName, email, creditCardInfo) {
      const member = new Member(lastName, firstName, email);
      member.register(creditCardInfo);
      this.members.push(member);
    }
}

// Lots of other code
```

When other code uses this, all it knows about how the `Gym` object works is that there's a method named `registerNewMember`, and if it's given some information, then a person becomes a member of the gym. The encapsulation of the behavior (adding a new member to the gym) is hidden inside the `Gym` object.

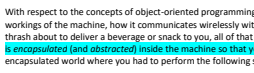
Likewise, the `Gym` object doesn't care how the credit card processing works. All of that is encapsulated in the `Member` object.

In the future, if the developers wanted to add another class for the `CreditCard` in order to encapsulate the secure processing, then the `Gym` object would not need to change at all. That is the power of encapsulation!

Encapsulation puts the behavior and data together behind methods that hide the specific implementation so that code that uses it doesn't need to worry about the details of it.

Inheritance: derived from one's ancestors

In the same way that biology passes traits of a parent organism to its descendants, so does object-oriented programming through its support of **inheritance**.



Conceptually, a class can be split into two parts:

- Interface: how other code "sees" the class
- Implementation: how the class actually does what it does

In other words, an interface is just a "signature with no body". A class is a **subtype** of an interface when it implements that interface. The language used is often "is a". Imagine an interface for `Car`. Then you could say, a `Porsche` class "is a" `Car`.

Javascript does not use interfaces, therefore it does not have "interface inheritance" or "subtyping". Instead, **Javscript strictly uses implementation inheritance.**

Implementation inheritance means that the properties and methods defined on a parent class are available on objects created from classes that inherit from those parent classes. This also means a child class has access to all the implementation (the doing) that was written in its parent class.

Exploration: inheritance in JavaScript

When you declare a class with no explicit parent class, then JavaScript will make it a child of `Object`.

```
class MyClass {}

// is the same as
class MyClass extends Object {}
```

When declaring a child class that will inherit from a specific parent class, use the `extends` keyword.

```
// child class
class MyChildClass extends MyClass {}
```

Polymorphism: having several shapes or forms

In object-oriented programming, polymorphism refers to processing various data types and classes through a single, uniform interface.

The two most common types of polymorphism are

"overloading" and

"overriding".

"Function overloading" refers to sending a different number or type of parameters to a particular function.

Consider `sum(number1, number2)` and `sum(listOfNumbers)`.

The former accepts two numeric parameters;

the second, an array of numbers.

The returned value from both would be a number (which is the total calculated by adding up all the numbers provided).

Each version of the function needs a different implementation (`number1 + number2` vs. a for-loop). Therefore,

this is an example of polymorphism.

"**Function overriding**" is when a child class gives its own - or a variation of - the implementation of a function from one of its ancestor classes (usually the parent).

Basics of OOP Design

The problem

Pretend you need to design a system to track students and teachers accessing an online school. Don't worry about the classrooms right now; rather, consider how the users will get through the virtual "door" (a.k.a. log in).

How can you break down this complex task using OOP design?

Step 1 - Identify potential classes

A good place to start is thinking about what "boxes" or "capsules" you could make to break down the problem into smaller pieces. In order words, where some **encapsulation** might be helpful.

Now, make a list of the pieces with a short description of each one's responsibility. A hint that you're on the right track is when a short phrase (5-10 words is sufficient). This should be like brainstorming - there's no right way.

Here are some possibilities for the online portal.

- Authentication - do they have the right username and password?
- Communication - how can the system help them recover a lost password
- Attendance - tracking when students should be there and when they actually show up
- Teacher schedule - when should a teacher be allowed to enter

You may come up with some alternate or additional candidates for objects and classes. There's more than one right answer and possibly even several very good answers. Sometimes a "best" solution is easy to find; other times you'll only discover it during implementation.

Step 2 - Specify properties and methods

The next step is to start listing the properties and methods of the identified classes.

Don't worry yet about how exactly these classes will inherit (extend) or connect with each other. And don't worry about what language they will be written in. You just need lists and notes about the classes you are proposing for the solution. This makes it possible to have meaningful discussions about data, structures and algorithms without getting stuck on parentheses, brackets, semicolons and other details which are not relevant until you start coding.

A) Class Account

Responsibility: Check credentials

- Property: username
- Property: password
- Method: login
- Method: logout
- Method: resetPassword

B) Class Person

Responsibility: Contact the user

- Property: emailAddress
- Property: mailingAddress
- Property: phoneNumber
- Method: verifyEmail
- Method: forgotPassword (e.g. emails a link to reset password)
- Method: sendTextMessage
- Method: sendLetter

C) Class Student

Responsibility: Track attendance compared to class schedule

- Property: schedule
- Property: attendanceRecord
- Method: enroll
- Method: dropout
- Method: attend

D) Class Teacher

Responsibility: Track instruction

- Method: teach

As you do this exercise, you may identify other pieces of complexity that you want to encapsulate. For example, a mailing address has many parts and printing a mailing label probably doesn't need to be a concern for any of the other classes.

E) Class MailingAddress

Responsibility: Physical address

- Property: street
- Property: city
- Property: state
- Property: zipCode
- Method: printMailingLabel

Step 3 - Define relationships

Finally, it's time to decide how the classes should relate to each other. There are two main philosophies which programmers love to debate endlessly.

- Inheritance
- Composition

You already learned that **inheritance** is when a class extends another class by extending its features then adding on and/or overriding.

Composition you've experienced before, even if you didn't know it. **Composition** is simply when one class is stored as the property of another class.

The easiest way to understand the difference is to apply it to a scenario like the virtual classroom access system.

Inheritance approach

First, you could strictly use inheritance so every class extends another class (except the one designated root or base class).

- Base class: Account
- Class Person extends Account
- Class Student extends Person
- Class Teacher extends Person
- Class MailingAddress extends ?????

However, this starts to break down when you get to MailingAddress. Where in the chain would that fit? Could Person extend both Account and MailingAddress. Perhaps. However, in the real would you would NOT say something like "a person is their mailing address"! So that means the answer is Mailing Address does NOT fit in this inheritance chain unless it's forced in.

The key question for inheritance goes something like this, "Is Y a type of X?"

So give that a try now....

- Is a Student a type of Person? Yes - it's a good case for inheritance.
- Is a Teacher a type of Person? Yes - it's a good case for inheritance.
- Is a Person a type of Account? Maybe - or maybe there's a better way.
- Is a MailingAddress a type of any of the other classes, or are any of these classes a type of MailingAddress? No - then you'll need another option.

Composition approach

Another way to connect objects is composition. **Composition** refers to objects "containing" other objects. In other words, the properties of one class are one or more other classes.

To understand the differences between inheritance and composition, consider the same example with a pure composition implementation (no inheritance). In addition to the properties listed above, the following could be included.

- *Account* properties
- No additions
- *Person* properties
- mailingAddress as an instance of MailingAddress
- *Student* properties
- account as instance of Account
- person as instance of Person
- *Teacher* properties
- account as instance of Account
- person as instance of Person

Alternatively, a person could be a property of an account...

- *Account* properties
- person as instance of Person
- *Person* properties
- mailingAddress as an instance of MailingAddress
- *Student* properties
- account as instance of Account
- *Teacher* properties
- account as instance of Account

Or, an account could be a property of a person...

- *Account* properties
- No additions
- *Person* properties
- mailingAddress as an instance of MailingAddress
- account as instance of Account
- *Student* properties
- person as instance of Person
- *Teacher* properties
- person as instance of Person

As you can see, there are many possibilities when it comes to OOP design. That is one of the reasons it is so important to take the time BEFORE you begin writing code to explore and discuss the options and make your best guess as to the best approach. Then, as you work, stay open to the possibility that you may need to revisit the design if you discover unexpected complexities while you have enough time remaining before your deadline.

Blended approach

Most of the time, the best OOP designs contain a blend of **composition** and **inheritance**.

Those objects which benefit from sharing a common base because they have many of the same properties and methods are ideal for inheritance. They often are the objects where an "is a" statement describes their relationship.

Object relationships which are better describes using a "contains" statement are ideal candidates for composition.

Some relationships between are not totally clear when you start the design. You just have to make your best guess and learn from your mistakes. It is helpful to study and implement different designs to learn what works well, and what doesn't work so well, for you. Therefore, throughout this course, you will be exposed to a variety of approaches and solutions.

Now, take one more look at classes and relationships for the virtual classroom access manager. This time with a blended approach. Notice the use of "is a" and "contains" to denote **inheritance** and **composition**, respectively.

- Class Account

- Class Person
- "Contains" account as an instance of Account
- "Contains" mailingAddress as an instance of MailingAddress
- Class Student "is a" Person
- Class Teacher "is a" Person
- Class MailingAddress

As you read this specification, you may get the sense that it "feels good". It will take time and experience - not just in the classroom, but also in the real world - to hone your skills of detecting excellent solutions. For now, it is okay and expected for you to ask your instructors for their guidance!

Step 4 - Document your decisions

The final step is crucial to mastering OOP. Write down in one place the final decisions you made in all the steps of your OOP design process.

A) Class Account

Responsibility: Check credentials

Intro to OOP Thinking

The main concept behind OOP is the idea that you can group data and related actions or behaviors together in order to treat them as a single entity within a larger system.

An item containing data and behaviors is called an **object**.

The data parts are called **properties** or **fields** of the object.

The action parts are called **methods** of the object.

The specification or definition of an object with properties and methods is called a **class**.

The **class** specifies the framework of the **properties** and **methods** (that is, data and actions).

(This is similar to how a function definition is just the framework of the function; whereas nothing happens until the function has been called. Likewise, a class is just a framework until it is **instantiated**.)

A specific object made from a class is called an **instance**

imagine a class Computer, then the device you are using to read this article is a specific **instance** of Computer. When a class is instantiated, there might be certain actions to do or data values to set right away.

A special method called a **constructor** handles this setup

a class may need to take some actions when it is destroyed or deconstructed; sometimes called "destructed" for short

A **destructor** method may be included in a class for this purpose.

"DRY stands for Don't Repeat Yourself and the principle is that there should only ever be one copy of any important piece of information. The reason for this principle is that one copy is much easier to maintain than multiple copies; if the information needs to be changed, there is only one place to change it." -*Paul Murrell, Introduction to Data Technologies*

First, consider an ordinary pen. In order to determine the properties of a Pen class, think about what questions you could ask someone about a pen to pick it out from a pile of various different pens.

- What color is the body?
- What color is the ink?
- How full is the ink level?
- What style is the writing tip?
- Can it be refilled?

Then translate these questions into property names, for example you might choose some properties like these.

- bodyColor
- inkColor
- inkLevel
- type (e.g. gel, fountain, rollerball, etc.)

Next, consider what a pen can do and what can be done to it.

- A pen can write
- A person can refill a pen with more ink

These lead you to some **methods** for the Pen class.

- write

Perhaps you are thinking about a pen which has replacement cartridges. While you may choose to make another method for that, you could also decide that it's another way to refill the ink, so there might be a way to **reuse** refillInk.

When thinking about objects you can start from the most specific object (e.g. a Sharpie). Or you can start from the most generic as in this thought experiment.

Here's one possbility for the **properties** of the Pencil class.

- type (mechanical, #2, drawing, ...)
- material (wood, plastic, metal)
- color (e.g. yellow, black, ...)
- graphiteThickness
- label (and more!)

Some OOP **methods** of the Pencil class may include

- write
- erase
- sharpen

Pillars of OOP

An object is most useful when it can hide complex functionality from other parts of the code. When each part has its own purpose, then it is easier to test and build a complex system.

Example 1: An ear does not need to know or care how an eye works. Likewise, an eye cares nothing about the ear. In fact, the location of each doesn't matter to their functionality so that can be handled by yet another "class" (the face). And the sensory portion of the brain cares about the signals coming from the eyes and ears, but not about their location or how the signals are generated.

In other words, you could assign a role to each of these "objects".

- Eyes: Convert light to sensory signals (e.g. user input)
- Ears: Convert sound to sensory signals (e.g. user input)
- Face: Place the eyes and ears where they are useful (e.g. layout)
- Cortex: Do brainy things with the sensory signals (e.g. processing)

This division of responsibility is called encapsulation.

- **Abstraction**

You probably noticed that the Pen class and the Pencil class have similar **properties (a.k.a. fields)** and **methods (a.k.a. functions)**. That's because they are both forms of writing instruments.

This **consideration of how objects are similar is called "abstraction"**. In some online materials, articles or books, you may find abstracted treated as a fourth pillar of OOP.

In this course, you will see abstraction discussed primarily with encapsulation (since they both are **concerned with hiding implementation details within classes**

Inheritance

Just like human parents can pass on traits and behaviors to their children, classes can be written which pass along their properties and methods to other classes through the process of **inheritance**.

So consider a WritingInstrument class as a parent of both the Pencil and Pen child classes. The benefit here is that the action of "writing" is (mostly) the same whether you are using a pen or pencil. Since writing is a complex task, it would be nice to only have to learn (or code) it one time.

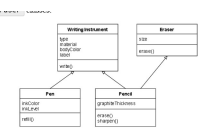
There also happen to be a few similar properties, such as material, bodyColor, and label.

Multiple inheritance

Some programming languages allow developers to specify more than one parent for a class. This is called **multiple inheritance**.

Consider the eraser. There are stand-alone erasers. There are also erasers on the ends of pencils. (Ignore those pens with erasers for now, just to keep this thought experiment a little easier.)

So then an Eraser class could have a property such as size and a method such as erase. There are times an instance of Eraser is useful. Additionally, Pencil can inherit from both the Writing Instrument and Eraser classes.



JAVASCRIPT DOES NOT SUPPORT MULTIPLE INHERITANCE!!!!!

Polymorphism:

If you break it down you'll see *poly*, which means "many", and *morph*, which refers to "change". So **polymorphism** in OOP is the idea that one method can have a different implementation even though it has the same name because the result is equivalent.

Now, consider those pens with erasers. The way they erase is similar to a stand-alone eraser or a pencil. The end result of erasing is to undo a mistake.

Some pens do not have erasers. If you want to erase a mistake you need something like white-out.

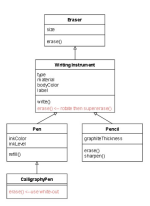
So, the WritingInstrument class could inherit from Eraser, and extend the implementation of erase() by rotating the writing instrument with the eraser near the paper before using the same erasing action.

Calling a function with the same name in the parent class requires you to reference the "super" or base class.

For example, super.erase().

Now, both Pen and Pencil may inherit from Writing Instrument and they will be able to erase. Cool!

Another class, perhaps CalligraphyPen, could then inherit from Pen and replace the erase() method with an implementation using white-out. This is what **polymorphism** is all about! Accomplishing the same result (undesired mark no longer visible a.k.a. erase()) in another fashion by changing or "morphing" the implementation appropriately for each class (Eraser, WritingInstrument, and CalligraphyPen).



```
  "dependencies": {
    "colors": "^1.4.0"
  },
```

```
  "devDependencies": {
    "mocha": "^7.0.1"
  }
}
```

Separating the development dependencies from the application's "regular" dependencies keeps the package installation process as lean as possible, by allowing npm to install just the packages that are actually needed for the package or application to successfully run.

Using npm to Perform Common Tasks -Part Two:

When getting started with an existing project that already contains package.json and package-lock.json files, you will need to use NPM to install dependencies if you don't install them you will receive errors when attempting to run the application.

to install an existing projects dependencies simply run the NPM install command, this causes NPM to install the dependencies listed in the package lock file.

Uninstalling a dependency.

Imagine that you install the `lodash`package by running the following command:

```
npm install lodash
```

Here's the output displayed by npm after the package is installed:

```
+ lodash@4.17.15
added 1 package from 2 contributors and audited 1 package in 0.609s
found 0 vulnerabilities
```

At this point, your `node_modules`folder will contain a folder named `lodash`that contains the code for the `lodash`package. Your `package.json`file will also list `lodash`as a dependency:

```
{
  "dependencies": {
    "lodash": "^4.17.15"
  }
}
```

To remove or uninstall the load dash package you can run the following command

npm uninstall load ash

When NPM uninstall the package it will remove the packaging all of its dependencies from the nose modules folder it will also update the package that Jason file in the package lock file so the package is no longer listed as a dependency.

Updating a dependency

Imagine that you added the `lodash`package as a dependency to a project awhile ago; back when the latest version of `lodash`was `3.0.0`. You can simulate this situation by installing a specific version of `lodash`:

```
npm install lodash@3.0.0
```

Which results in the following output:

```
+ lodash@3.0.0
added 1 package from 5 contributors and audited 1 package in 0.606s
found 3 vulnerabilities (1 low, 2 high)
  run `npm audit fix` to fix them, or `npm audit` for details
```

For now, ignore the security vulnerabilities that npm found. You'll learn in a bit how to use `npm audit`to resolve those issues.

Here's what the dependency in the `package.json`file looks like at this point:

```
{
  "dependencies": {
    "lodash": "^3.0.0"
  }
}
```

Here's what the dependency in the `package.json`file looks like at this point:

```
{
  "dependencies": {
    "lodash": "^3.0.0"
  }
}
```

Remember that server ^3.0.0 Means that you'll except any minor and patch versions For loadash major version three

If you want to update lodash, so you can use some features that were added in version3.1.0.
You can update the lodash packageby running the following command:
npm update lodash

And here's what your dependencies look like in the `package.json`file:

```
{
  "dependencies": {
    "lodash": "^3.0.0"
  }
}
```

Updating all project dependencies:

The returned value from both would be a number (which is the total calculated by adding up all the numbers provided).

Each version of the function needs a different implementation (number1 + number2 vs. a for-loop). Therefore,

this is an example of polymorphism.

"**Function overriding**" is when a child class gives its own - or a variation of - the implementation of a function from one of its ancestor classes (usually the parent).

A built-in example of polymorphism

All objects in JavaScript share a common parent class, the Object parent class.

The Object class has a method named toString() on it.

Since all objects in JavaScript are child classes (or grandchild classes or great-grandchild classes or great-great-...), that means that every object in JavaScript has a toString() method!

If a class doesn't create its own, then it will fall back to its parent class' implementation of toString().

If the parent class doesn't have an implementation, and the parent's parent class doesn't have an implementation, it will keep going until it gets to the Object class and use that one. (That's some recursion in there. Did you sense that?)

If you like to experience this for yourself, open a terminal, start node, and type the following.

```
> [1, 2, 3].toString();
'1,2,3'
> "some text".toString();
'some text'
> new Date().toString();
'the current date and time='
> new Object().toString();
'[object Object]'
```

You'll notice the following:

- The toString() method of an array takes the values in the array and turns them into a comma-delimited string; that is, it puts commas between each of the items.
- The toString() method of a string does nothing and just returns the string object (you might remember that strings are primitive types, but strings are special, and you can also call methods on them like they are objects)
- The toString() method of a Date object returns a long textual representation of the date and time which the Date object represents.
- The toString() method of Object returns "[object Object]" because that's all it knows about itself.

If you feel like [object Object] this is less-than-useful, then it can be your motivation to overload the toString() method in your classes!

A deeper exploration of polymorphism and inheritance

Imagine that you have created the following code in a JavaScript file and loaded it (into a browser through a <script> tag, or run it with the node command).

The **extends** keyword is used in class declarations or class expressions to create a class that is a child of another class.

```
class Charity {}
class Business {
    toString() {
        return 'Give us your money.';
    }
}
class Restaurant extends Business {
    toString() {
        return 'Eat at Joe\'s!';
    }
}
class AutoRepairShop extends Business {}
class Retail extends Business {
    toString() {
        return 'Buy some stuff!';
    }
}
class ClothingStore extends Retail {}
class PhoneStore extends Retail {
    toString() {
        return 'Upgrade your perfectly good phone, now!';
    }
}
console.log( new PhoneStore().toString() ); //Upgrade your perfectly good phone, now!
console.log( new ClothingStore().toString() ); //Buy some stuff!
console.log( new Restaurant().toString() ); //Eat at Joe's!
console.log( new AutoRepairShop().toString() ); //Give us your money.
console.log( new Charity().toString() ); //[object Object]

/*
The two class PhoneStore and Restaurant use polymorphism to overloaded the toString()
function, so their specific messages are printed.

The three classes AutoRepairShop, Charity, and ClothingStore do not have their own toString() methods.

That means that an object of one of those three types can't immediately respond to that method invocation. The JavaScript runti
me at that point starts inspecting parent objects (following the "inheritance"* chain) to find a toString() method.


For AutoRepairShop, it finds a toString() method on its parent class Business, and prints "Give us your money.".
For ClothingStore, it finds a toString() method on its parent class Retail, and prints "Buy some stuff!".


For Charity, it finds a toString() method on its implicit parent class Object, and prints "object Object".
*/
```

In this reading, you learned about the **three pillars of object-oriented programming** (encapsulation, inheritance, and polymorphism) and how they relate to JavaScript.

You learned that encapsulation is putting all of the details behind an object's methods.

You learned that inheritance is the ability to gain behavior and data from parent classes.

You learned that polymorphism is just a way of extending the functionality of a given method in a child class to do something more or different than its parent.

Step 4 - Document your decisions

The final step is crucial to mastering OOP. Write down in one place the final decisions you made in all the steps of your OOP design process.

A) Class `Account`

Responsibility: Check credentials

- Property: `username`
- Property: `password`
- Method: `login`
- Method: `logout`
- Method: `resetPassword`

B) Class `Person`

Responsibility: Contact the user

- Property: `account` as an instance of `Account`
- Property: `emailAddress`
- Property: `mailingAddress` as an instance of `MailingAddress`
- Property: `phoneNumber`
- Method: `verifyEmail`
- Method: `forgotPassword` (e.g. emails a link to reset password)
- Method: `sendTextMessage`
- Method: `sendLetter`

C) Class `Student` is a `Person`

Responsibility: Track attendance compared to class schedule

- Property: `schedule`
- Property: `attendanceRecord`
- Method: `enroll`
- Method: `dropout`
- Method: `attend`

D) Class `Teacher` is a `Person`

Responsibility: Track instruction

- Method: `teach`

E) Class `MailingAddress`

Responsibility: Physical address

- Property: `street`
- Property: `city`
- Property: `state`
- Property: `zipCode`
- Method: `printMailingLabel`

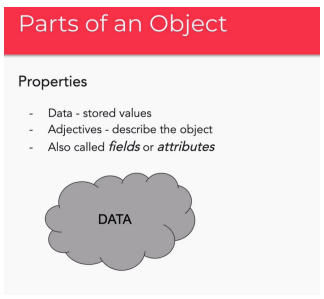
What you've learned

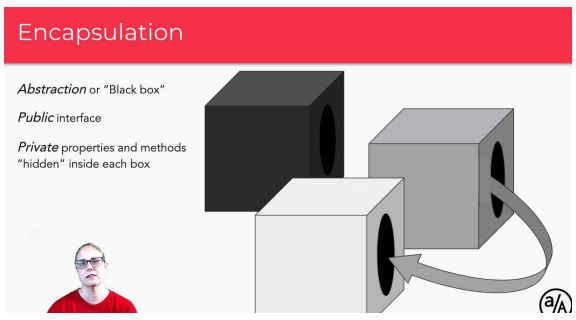
In this article, you walked through the steps to approach design in **Object-Oriented Programming** (OOP).

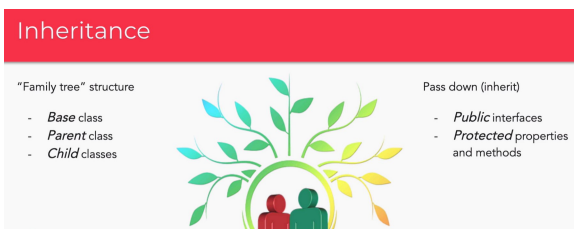
1. Identify potential classes
2. Specify properties and methods
3. Define relationships
4. Document your decisions

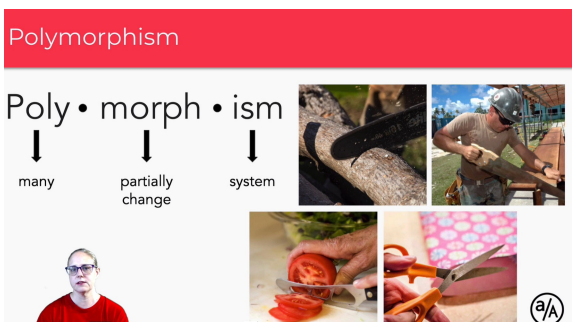
Along the way, you compared and contrasted several approaches to object relationships, namely **inheritance** (with an "is a" statement) verses **composition** (with a "contains" statement).

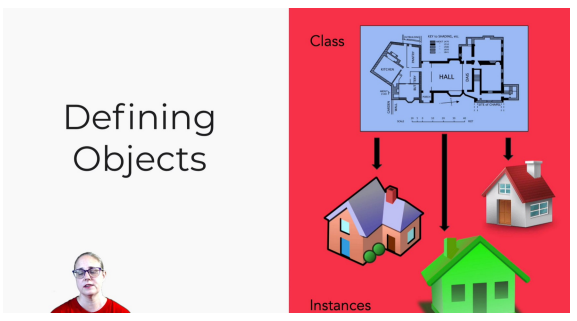
Remember, design is both an iterative process, and a skill you'll continuously improve throughout your career as a software programmer.

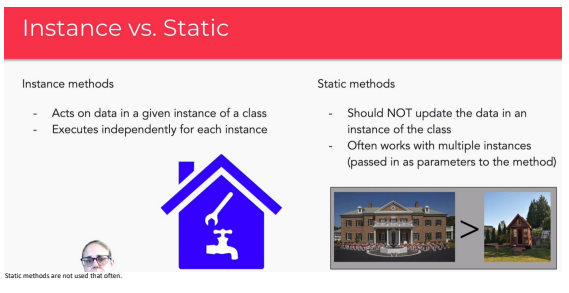










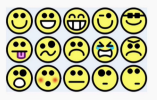


Private, Protected, and Public

Private - only for me

Protected - Shared with my children

Public - How others connect with me



OOP Summary

Object-Oriented Programming

Pillars

- Encapsulation
- Inheritance
- Polymorphism

Defines

- Classes, Instances
- Properties & Methods
  - Constructor, Destructor
  - Instance, Static
  - Private, Protected, Public