PER SCHOLAS

GLAB 308A.2.1:

An Object-Oriented Adventure

Version 1.0, 10/13/23

Click here to open in a separate window.

Introduction

This guided activity walks you through the process of creating a simple adventuring game using object-oriented programming principles. We will begin with basic concepts and the structure of the game, and then refine that structure with classes and other programming patterns.

Objectives

- Use nested arrays and objects.
- Combine objects, arrays, and functions.
- Create a class to define the blueprint for creating objects.
- Add methods to a class.
- Set properties on an instance of a class.
- Make an instance of each class customizable.
- Create methods to alter the properties of an instance.
- Develop a class that inherits attributes from a "parent" class.
- Create static properties for a class.
- Create a "factory."

Submission

Submit your completed lab using the **Start Assignment** button on the assignment page in Canvas.

Your submission should include:

- A GitHub link to your completed project repository.

Instructions

Initialize a new git repository in a local project folder and create a JavaScript file to contain your code. Within your code editor of choice, follow along with the steps below to create the adventure game.

Feel free to take creative liberty with your own game!

Commit frequently! Every time something works, you should commit it. Remember, you can always go back to a previous commit if something breaks.

Part 1: Humble Beginnings

Let's model a simple adventurer with basic properties such as health and an inventory.

We will call the adventurer "Robin."

```
const adventurer = {
   name: "Robin",
   health: 10,
   inventory: ["sword", "potion", "artifact"]
}
```

From the adventurer object, we can access Robin's inventory using a combination of dot notation and square bracket syntax. For example, we could find a sword at adventurer.inventory[0].

As an additional practice exercise, create a loop that logs each item in Robin's inventory.

Nested arrays are useful, but so are nested objects. Let's give Robin a companion to travel with – a furry friend they call "Leo."

```
const adventurer = {
   name: "Robin",
   health: 10,
   inventory: ["sword", "potion", "artifact"],
   companion: {
       name: "Leo",
       type: "Cat"
   }
}
```

This is an extremely common data pattern in programming. Nested arrays and objects allow programmers to store data in organized ways. Accessing the data should be both convenient and easily understood, particularly when using several objects of the same data structure, such as those derived from a Class (more on that later).

Next, give Robin's feline friend a friend of his own:

- Add a "companion" sub-object to "Leo" with the following properties:
  - The companion's name is "Frank."
  - The companion's type is "Flea."
  - The companion has its own belongings, which includes a small hat and sunglasses.

Now we have created an array within an object within an object within an object, but that is not all we can do! Objects can also have their own functions, called methods, which define specific actions that object can take.

Give Robin the following method:

```
const adventurer = {
   name: "Robin",
   health: 10,
   inventory: ["sword", "potion", "artifact"],
   companion: ...
   roll (mod = 0) {
      const result = Math.floor(Math.random() * 20) + 1 + mod;
      console.log(`${this.name} rolled a ${result}.`)
   }
}
```

Now we have a method for "dice rolls," a common way to handle outcomes in adventuring games. Test this method by calling adventurer.roll() a few times.

What if we had many adventurers? As you can imagine, creating several of these objects manually would be time consuming, inefficient, and prone to errors.

Next, we will level up our approach using Classes.

Part 2: Class Fantasy

Let's take a moment to analyze the data above. What are the common features of each object?

When creating classes, begin by searching for the simplest form your data takes. Remember, you can add specificity later by *extending* the classes.

Start with a Character class, which will define generic character entities. Robin and their companions all have a name, so the Character class should definitely include name as one of its properties. At this stage, we will also make the decision that every character should have health (which we will standardize to a maximum of 100, and an inventory (even if the inventory is empty).

Here is what the basic Character class looks like so far, including a **constructor** function that allows us to create new characters with whatever name we would like:

```
class Character {
  constructor (name) {
    this.name = name;
    this.health = 100;
    this.inventory = [];
  }
```

```
}
```

Every character should also be able to make rolls. Add the roll method to the Character class.

Now, we can re-create Robin using the Character class!

```
const robin = new Character("Robin");
robin.inventory = ["sword", "potion", "artifact"];
robin.companion = new Character("Leo");
robin.companion.type = "Cat";
robin.companion.companion = new Character("Frank");
robin.companion.companion.type = "Flea";
robin.companion.companion.inventory = ["small hat", "sunglasses"];
```

Not only does this allow us to create standardized objects for each character, it also ensures that they all have common properties and methods such as roll(). Even the companions can roll now; give it a try! This saves us a significant amount of typing since we do not need to manually add this method to each nested object.

While progress has been made, this is still not the most efficient way to create these objects. In order to effectively create companions, we need to extend the Character class for added specificity.

Part 3: Class Features

When extending a class, the “child” class inherits all properties of its parents. This means that we do not need to account for the name, health, inventory, or roll method of Character children classes.

Let’s begin by creating an Adventurer class. What attributes might be specific to an adventure, but that not all characters have? Take a look at our example below, and expand upon it with your own properties and methods.

```
class Adventurer extends Character {
  constructor (name, role) {
    super(name);
    // Adventurers have specialized roles.
    this.role = role;
    // Every adventurer starts with a bed and 50 gold coins.
    this.inventory.push("bedroll", "50 gold coins");
  }
  // Adventurers have the ability to scout ahead of them.
  scout () {
    console.log(`${this.name} is scouting ahead...`);
    super.roll();
  }
}
```

What else should an adventurer be able to do? What other properties should they have?

Next, create a Companion class with properties and methods specific to the companions.

Finally, change the declaration of Robin and the companions to use the new Adventurer and Companion classes.

Part 4: Class Uniforms

Using static properties and methods, you can create uniform attributes for the class itself rather than instances of the class. Static properties are typically constant values that can be used elsewhere for reference, or utility methods that do not rely on the values of a specific class instance.

Using the static keyword:

- Add a static MAX_HEALTH property to the Character class, equal to 100.
- Add a static ROLES array to the Adventurer class, with the values "Fighter," "Healer," and "Wizard." Feel free to add other roles, if you desire!
  - Add a check to the constructor of the Adventurer class that ensures the given role matches one of these values.

Are there other static properties or methods that make sense to add to these classes?

Part 5: Gather your Party

Sometimes, you need many objects of a class that have one or more shared property values. A common approach for creating many similar objects of a single class, and keeping track of them is creating a "factory."

Factories are classes that generate objects according to the *factory's* instance properties.

As an example, let's look at how we might create many "healer" role adventurers using a factory:

```
class AdventurerFactory {
  constructor (role) {
    this.role = role;
    this.adventurers = [];
  }
  generate (name) {
    const newAdventurer = new Adventurer(name, this.role);
    this.adventurers.push(newAdventurer);
  }
  findByIndex (index) {
    return this.adventurers[index];
  }
  findByName (name) {
    return this.adventurers.find((a) => a.name === name);
```

```
    }
}

const healers = new AdventurerFactory("Healer");
const robin = healers.generate("Robin");
```

Now, we can create many “healers” using the healer factory, and conveniently find them using the factory’s methods. We can also add additional convenience methods to the factory as the requirements of the program expand.

An alternative approach to this would be to extend the Adventurer class to create a Healer class. This would be the practical approach if healers had additional properties and methods, but if healers are just adventurers with a specific role, creating an entire class for them is inefficient.

In the next part, it may be prudent to create classes for each adventuring role, depending on the additional properties and methods you would like to add.

Part 6: Developing Skills

Many of the core features of these characters are now implemented, but the adventurers cannot really do much yet. The only action (method) they have is scout().

Create an additional method, duel(), for the Adventurer class with the following functionality:

- Accept an Adventurer as a parameter.
- Use the roll() functionality to create opposing rolls for each adventurer.
- Subtract 1 from the adventurer with the lower roll.
- Log the results of this “round” of the duel, including the rolls and current health values.
- Repeat this process until one of the two adventurers reaches 50 health.
- Log the winner of the duel: the adventurer still above 50 health.

What other properties and methods could these classes have? Should fighters, healers, and wizards have their own methods? Should companions have specific methods?

Feel free to experiment with your own ideas, be they silly or practical. The goal of this exercise is to develop new skills for the characters and yourself! Express your creativity.

Part 7: Adventure Forth

Now that you have convenient ways to create a variety of characters with a variety of methods, experiment! Generate a whole host of adventurers and their companions, and have them interact using the instance methods you have developed.

If time allows, create other classes that can interact with your characters; perhaps more characters, but in a different direction from adventurers or companions – dragons, orcs, elves, vampires...

You can also create classes for the inventory itself, and include inventory methods such as adding, removing, searching, selling, trading. Even individual items could be their own classes, and have properties and methods specific to the type of item.

While this activity is intended to be light-hearted and silly, every tool you have utilized thus far is extremely common and relevant in every variety of development environment, from game development to data processing to complex enterprise applications.

Object-oriented programming is a feature of almost every modern programming language, and being comfortable with its core concepts will enable you to be a more capable developer!