

Lectures 3 - 4: Introduction to Classes and Object Oriented Programming, JSDoc, Pseudorandom Number Generators (PRNG)

Dr. Jason Schanker

Objects/Classes/Methods

- A class describes a group of related objects and provides the “blueprint” for what these objects can do (methods) and what attributes they have.
 - ❑ Example: String class: Method such as `toUpperCase`
 - ❑ Example: Car class: Methods such as `drive`; properties such as `fuelAmount`
- In object-oriented programming languages such as Java (not JavaScript) where functions are organized around objects, an object is often referred to as an instance of a class. Each object of a class can have the same or different values for their attributes.
- In JavaScript, can create a new instance of a class or object using the class name, `new` and one or more arguments:
 - ❑ Example:

```
let s = new String("abc");           // don't do this in general
    console.log(s.length);              // logs length property: 8
    console.log(s.toUpperCase());       // logs "ABC"
    console.log(s);                     // logs "abc"
```

Objects/Classes/Methods (cont.)

- Different objects of the same class may respond differently to method calls.

❑ Example:

```
let s = new String("abc");  
let t = new String("stressed");  
console.log(s.toUpperCase()); // logs "ABC"  
console.log(t.toUpperCase()); // logs "STRESSED"
```

- **Calling a method on an object is like calling a function which takes the object as an implicit argument:**

```
let toUpperCase = word => word.toUpperCase();  
console.log(toUpperCase(s));  
console.log(toUpperCase(t));
```

Objects/Classes/Methods (cont.)

- Calling a method on an object belonging to a class that doesn't support it results in an error.

- ❑ Example:

- ```
let p = 3.1415;
console.log(p.toUpperCase()); // error (3.1415 is a number,
 // doesn't support toUpperCase)
```

- However, you can “monkey patch” the class so it does (this refers to the object it's called on; monkey patching classes you didn't define is generally considered a bad idea.

- ```
Number.prototype.toUpperCase = () => "uppercase" + this.toString();  
let p = 3.1415;  
console.log(p.toUpperCase()); // logs "uppercase3.1415"
```

Car Class

➤ You can also create your own new classes such as a Car and create new cars, each with their own fuel and mpg!

❑ Example: <https://repl.it/LBCA>

```
class Car {
  constructor(fuel, mpg) {
    this.fuel = fuel;
    this.mpg = mpg;
  }

  drive(miles) {
    console.log("VRRRRRRROOOOOOMMMMMMM.....");
    let fuelRequired = miles/this.mpg;

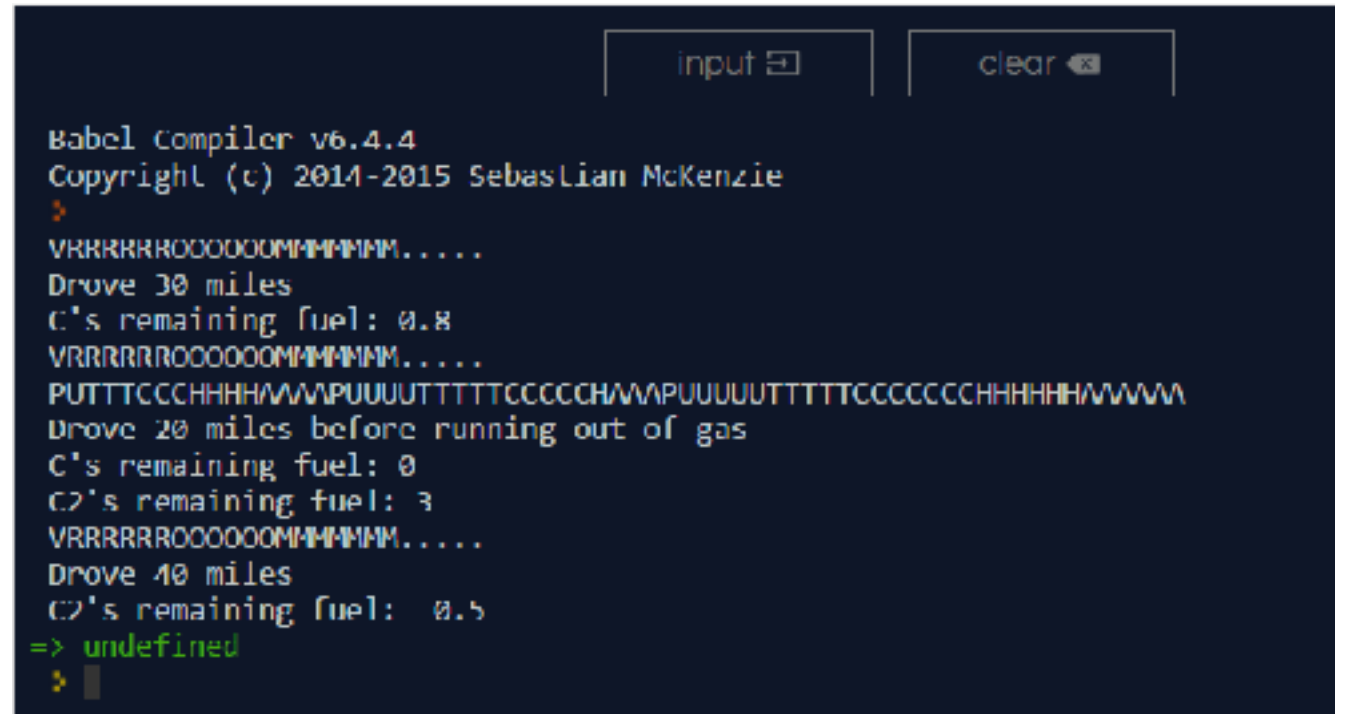
    if(fuelRequired <= this.fuel) {
      this.fuel = this.fuel - fuelRequired;
      console.log("Drove", miles, "miles");
    }
    else {
      console.log("PUTTTCCCHHHHAAAAPUUUUUTTTTCCCCCHAAAPUUUUUTTTTCCCCCCHHHHHHAAAAAA");
      console.log("Drove", (this.fuel*this.mpg), "miles before running out of gas");
      this.fuel = 0;
    }
  }
}
```

Car Objects (Instances of Car class)

- You can also create your own new classes such as a Car and create new cars, each with their own fuel and mpg!

❏ Example: <https://repl.it/LBCA>

```
let c = new Car(2, 25);  
let c2 = new Car(3, 16)  
c.drive(30);  
console.log("C's remaining fuel:", c.fuel);  
c.drive(30);  
console.log("C's remaining fuel:", c.fuel);  
console.log("C2's remaining fuel:", c2.fuel);  
c2.drive(40);  
console.log("C2's remaining fuel: ", c2.fuel);
```



```
Babel Compiler v6.4.4  
Copyright (c) 2014-2015 Sebastian McKenzie  
>  
VRRRRRRROOOOOOMMMMMM.....  
Drove 30 miles  
C's remaining fuel: 0.8  
VRRRRRRROOOOOOMMMMMM.....  
PUTTTCCCHHHHWWWWPUUUUTTTTCCCCCHWWWWPUUUUTTTTCCCCCHHHHHWWWW  
Drove 20 miles before running out of gas  
C's remaining fuel: 0  
C2's remaining fuel: 3  
VRRRRRRROOOOOOMMMMMM.....  
Drove 40 miles  
C2's remaining fuel: 0.5  
=> undefined  
>
```

Exercise: Create Point Class

- Create a Point class that has a constructor and distance method. Test it by creating new points and checking the distances between them.

Note you can surround multiline comments between `/*` and `*/`; these will be ignored by the computer but if you follow the format of JSDoc, you can generate nice Documentation (like a user manual) page. See: <http://usejsdoc.org/howto-es2015-classes.html>

```
/** Class representing a point with 2 coordinates. */
class Point {
  /**
   * Create a point.
   * @param {number} x - The x value.
   * @param {number} y - The y value.
   */
  constructor(x, y) {
    // Fill in code to set attributes of point
  }

  /**
   * Returns the distance between itself and another point
   * @param {Point} p the point to find the distance between
   * @return {number} distance between itself and p
   */
  distance(p) {
    return formula for distance;
  }
}
```

<https://repl.it/LHpr/1>

Encapsulation

- From Java Concepts: “Encapsulation is the process of providing a public interface while hiding the implementation.”
 - ❑ Real-world Example: When you drive a car, you can use the steering wheel and pedals, but you don’t directly interact with the engine. This allows a safer experience and frees you from worrying. It also allows for your engine to be replaced, and your car still works the same. If you somehow depended on the engine being a certain way, this wouldn’t be the case.
 - ❑ Two different students might have implemented the Point class differently, but so long as they both do their job, you can swap one for the other without knowing the difference.
 - ❑ Should a person be able to modify the available fuel to be greater than the capacity? Set to a negative number? Should anyone be able to access another person’s social security number, or when they ask the person, should they be turned down?
- Why encapsulation is useful?
 - ❑ Simplifying tasks -- You can focus on the big picture of your application without worrying about lower level details (e.g., drawing rectangles).
 - ❑ If the implementation of a class needs to change, classes depending on the *use* of that changing class’s methods can still work (e.g., Point)
 - ❑ Security -- Avoid accidental/malicious modification: e.g., setting fuel to negative number.

Getters/setters

- Rather than accessing instance variables directly (attributes such as a car's fuel) from outside a class, it's good practice to provide methods, called getters and setters, for controlling access.
- By convention, you may see variables that are meant to be private have an underscore as its first or last character (e.g., instead of `fuel`, you'd see `fuel_`). For clarity, we will instead use a prefix of `my`. **Note:** In JavaScript, the getter/setter methods must have a different name from the instance variable. Getters/setters can be accessed without `()` like attributes.

❑ Car Class Example Revisited: In the constructor: `this.myFuel = fuel;`

```
get fuel() {  
    return this.myFuel;  
}  
  
set fuel() {  
    throw new Error("Car myFuel property should not be modified directly.");  
}  
  
// Outside of class definition  
let c = new Car(2, 25);  
console.log(c.fuel); // logs 2  
c.fuel = 4; // Error: Car myFuel property should not be modified directly.
```

Updated Car Class with getters/setters

➤ <https://repl.it/LBCA/1>

```
class Car {
  constructor(amount, mpg) {
    this.myFuel = amount;
    this.myMpg = mpg;
  }

  drive(miles) {
    console.log("VRRRRRRROOOOOOMMMMMMM.....");
    let fuelRequired = miles/this.myMpg;

    if(fuelRequired <= this.fuel) {
      this.myFuel = this.myFuel - fuelRequired;
      console.log("Drove", miles, "miles");
    }
    else {
      console.log("PUTTTCCCHHHHAAAAPUUUUTTTTCCCCCHAAAPUUUUUTTTTCCCCCCHHHHHHAAAAAA");
      console.log("Drove", (this.fuel*this.myMpg), "miles before running out of gas");
      this.myFuel = 0;
    }
  }

  get fuel() {
    return this.myFuel;
  }

  set fuel(amount) {
    throw new Error("Car myFuel property should not be modified directly.");
  }
}
```

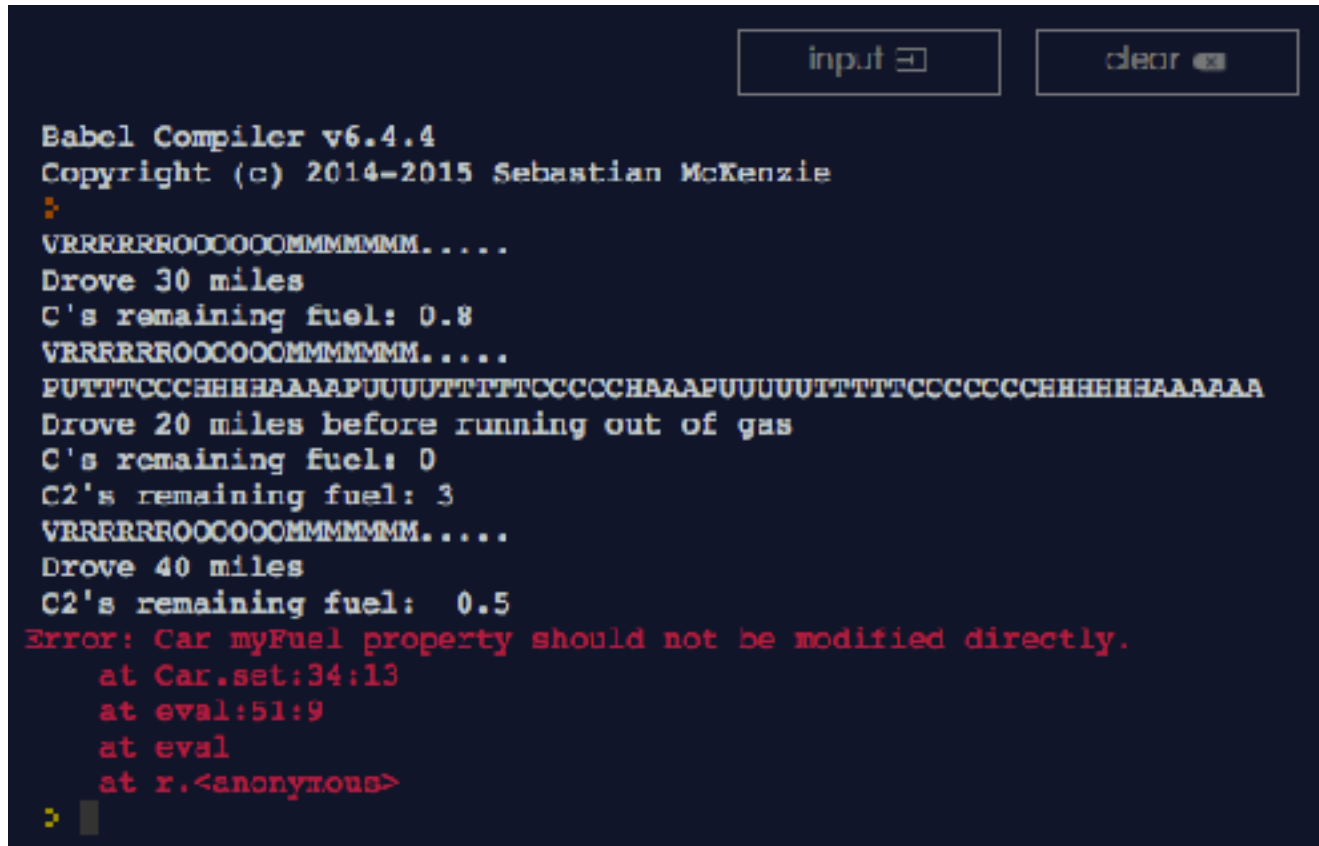
Car Objects (Instances of Car class)

➤ Note that the client code (code using the Class) works as before. From the previous code, it would be unaware that any change was made.

❏ Example: <https://repl.it/LBCA/1>

```
let c = new Car(2, 25);
let c2 = new Car(3, 16)
c.drive(30);
console.log("C's remaining fuel:", c.fuel);
c.drive(30);
console.log("C's remaining fuel:", c.fuel);
console.log("C2's remaining fuel:", c2.fuel);
c2.drive(40);
console.log("C2's remaining fuel: ", c2.fuel);

c2.fuel = -5; // results in error
```



```
Babel Compiler v6.4.4
Copyright (c) 2014-2015 Sebastian McKenzie
>
VRRRRRRROOOOOOMMMMMMM.....
Drove 30 miles
C's remaining fuel: 0.8
VRRRRRRROOOOOOMMMMMMM.....
PUTTTCCCHHHHAAAAPUUUUUTTTTCCCCCHAAAPUUUUUTTTTCCCCCCHHHHHHAAAAAA
Drove 20 miles before running out of gas
C's remaining fuel: 0
C2's remaining fuel: 3
VRRRRRRROOOOOOMMMMMMM.....
Drove 40 miles
C2's remaining fuel: 0.5
Error: Car myFuel property should not be modified directly.
    at Car.set:34:13
    at eval:51:9
    at eval
    at r.<anonymous>
>
```

Exercise: Modify Point Class w/getters and setters

➤ Modify to have instance variables `myX` and `myY`. Add getters to get `x` and `y`, but make setters throw error: Point is immutable: i.e., Point's `x` and `y` can't be changed.

Note you can surround multiline comments between `/*` and `*/`; these will be ignored by the computer but if you follow the format of JSDoc, you can generate nice Documentation (like a user manual) page. See: <http://usejsdoc.org/howto-es2015-classes.html>

```
/** Class representing a point with 2 coordinates. */
class Point {
  /**
   * Create a point.
   * @param {number} x - The x value.
   * @param {number} y - The y value.
   */
  constructor(x, y) {
    // Fill in code to set attributes of point
  }

  /**
   * Returns the distance between itself and another point
   * @param {Point} p the point to find the distance between
   * @return {number} distance between itself and p
   */
  distance(p) {
    return formula for distance;
  }
}
```

<https://repl.it/LHpr/1>

Another example with getters/setters (Person class)

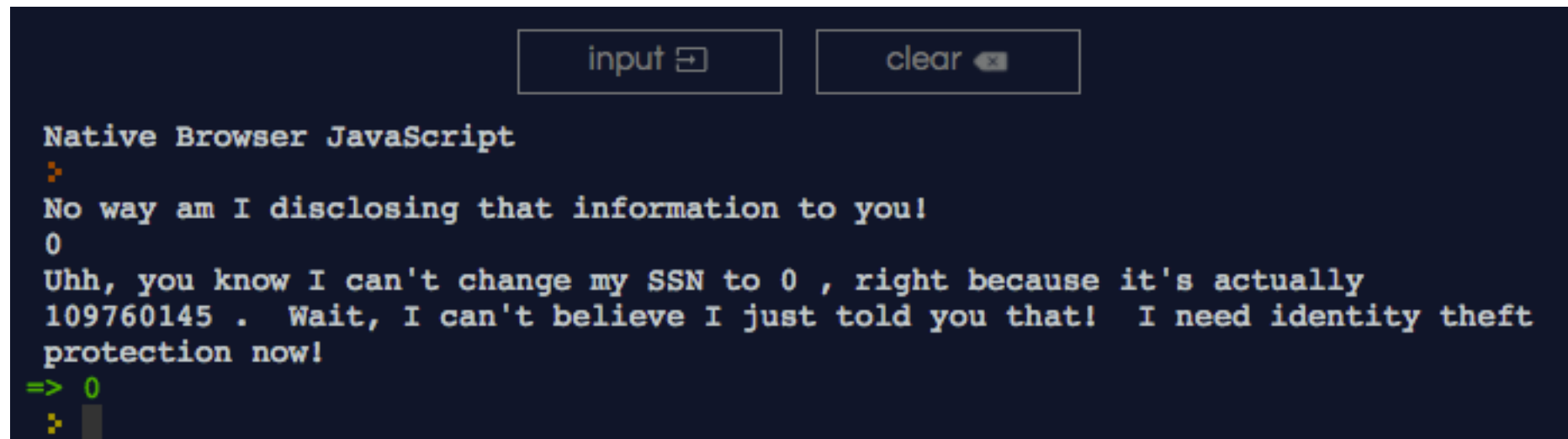
➤ <https://repl.it/B25r/1>

```
class Person {
  constructor() {
    this.mySSN = Math.floor(Math.random()*1000000000) + 1000000000;
  }

  get SSN() {
    console.log("No way am I disclosing that information to you!");
    return 0;
  }

  set SSN(value) {
    console.log("Uhh, you know I can't change my SSN to", value, ", right because it's actually",
this.mySSN, ". Wait, I can't believe I just told you that! I need identity theft protection now!");
  }
}

var p = new Person();
console.log(p.SSN);
p.SSN = 0;
```



The screenshot shows a web browser interface with a dark background. At the top, there are two buttons: "input" with a text input field and "clear" with a trash icon. Below the buttons, the browser's developer console is open, displaying the output of the JavaScript code. The console title is "Native Browser JavaScript". The output shows the constructor message, the SSN value (0), and the setter message. The input field is empty, and the clear button is visible.

```
Native Browser JavaScript
>
No way am I disclosing that information to you!
0
Uhh, you know I can't change my SSN to 0 , right because it's actually
109760145 . Wait, I can't believe I just told you that! I need identity theft
protection now!
=> 0
>
```

Car Class

➤ You can also create your own new classes such as a Car and create new cars, each with their own fuel and mpg!

❑ Example: <https://repl.it/LBCA>

```
class Car {
  constructor(fuel, mpg) {
    this.fuel = fuel;
    this.mpg = mpg;
  }

  drive(miles) {
    console.log("VRRRRRRROOOOOOMMMMMMM.....");
    let fuelRequired = miles/this.mpg;

    if(fuelRequired <= this.fuel) {
      this.fuel = this.fuel - fuelRequired;
      console.log("Drove", miles, "miles");
    }
    else {
      console.log("PUTTTCCCHHHHAAAAPUUUUUTTTTCCCCCHAAAPUUUUUTTTTCCCCCCHHHHHHAAAAAA");
      console.log("Drove", (this.fuel*this.mpg), "miles before running out of gas");
      this.fuel = 0;
    }
  }
}
```

Another Example: BankAccount Class Skeleton adapted from Java Concepts):

<https://repl.it/LHuE/1>

```
/**
 * A bank account has a balance that can be changed by deposits and withdrawals.
 */
class BankAccount
{

    /**
     * Creates a bank account with the given balance.
     * @param {number} amount the starting amount.
     */

    constructor(amount)
    {
        // TODO: implement constructor
    }

    /**
     * @type {number}
     */

    get balance()
    {
        // TODO: implement getter for balance
    }

    /**
     * Deposits money into the bank account.
     * @param {number} amount the amount to deposit
     */

    deposit(amount)
    {
        // TODO: implement deposit method
    }

    /**
     * Withdraws money from the bank account.
     * @param amount the amount to withdraw
     */
    withdraw(amount)
    {
        // TODO: implement withdraw method
    }
}
```

Pseudorandom number generator (PRNG)

- You can generate pseudorandom numbers between 0 and 1 in JavaScript using `Math.random()` ;

```
console.log(Math.random()) ;
```

- The numbers generated, while seeming random, are actually completely determined by an initial value, called a seed, and the number of times that a value was asked for.
 - ❑ Imagine book of seemingly random numbers (seed is like page #).

Pseudorandom number generator (PRNG)

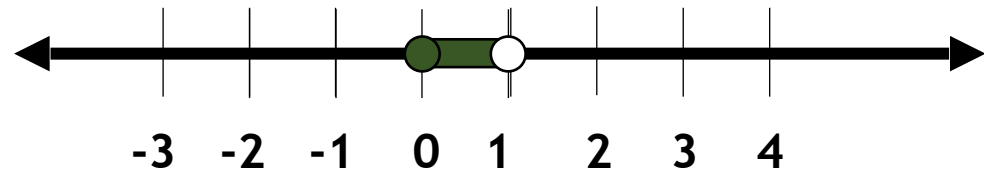
- Reproducible - given same seed, determined by e.g., current time, you'll get the same sequence every time; has advantage of reproducing event; disadvantage of predictability

```
# Python code below (https://repl.it/LHvt)
import random
random.seed(123)          # "page 123"
print(random.random())    # 0.052363598850944326
print(random.random())    # 0.08718667752263232
print(random.random())    # 0.4072417636703983
```

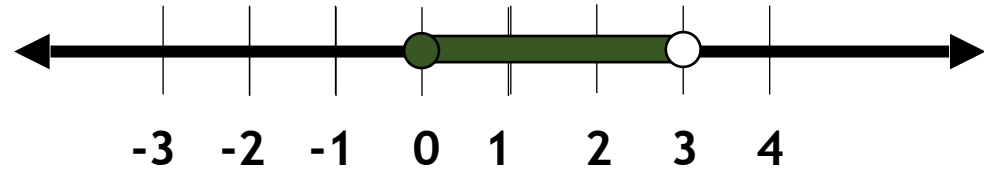
Digits of π could work for PRNG (seed determines start position, number of times run determines offset from that position), but too slow to generate.

From $[0, 1)$ to other interval

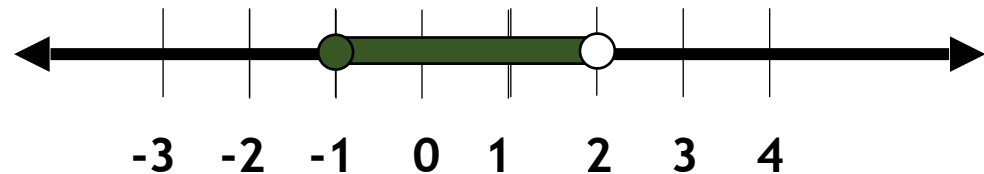
- What if you want a (pseudo)random number in $[-1, 2)$ instead?
- What if you want an integer from -1 to 1? (use `Math.floor`)



`Math.random()`



`3*Math.random()`



`3*Math.random() - 1`

- Interval length: 3 ($2 - (-1)$): Stretch x 3 Smallest number: -1 (Shift left 1)

```

/**
 * API adapted from A Computer Science Tapestry
 * Class for simulating a die (object "rolled" to generate a random number)
 */
class Die
{

    /**
     * Creates a die with the given number of sides
     * @param {number} sides the number of "sides" on the die
     */

    constructor(sides)
    {
        // TODO: implement constructor
    }

    /**
     * Rolls the die
     * @returns {number} the random "roll" of the die, a uniformly distributed
     *                    random number between 1 and # sides
     */

    roll()
    {
        // TODO: implement deposit method
    }

    /**
     * @type {number}
     */

    get rolls()
    {
        // TODO: implement getter for number of times roll called for an instance of the class
    }

    /**
     * @type {number}
     */

    get sides()
    {
        // TODO: implement getter for number of sides
    }
}

```

Example: Implement
Dice class using API
from A Computer
Science Tapestry -
<https://repl.it/LM0h>

Implementing Methods

- Keep method bodies small: If it's more than a few lines of code, consider implementing a new method.
- Minimize side effects (pg. 384, Java Concepts): Side effects are (unexpected?) changes to data that are noticed outside of the method.
 - ❑ Example: Printing to the Console is a side effect.
 - ❑ Example: Changing data passed to a method (e.g., For two dice d1 and d2, if calling `d1.isSameNumberAfterRoll(d2)` resulted in the number of rolls for d1 and d2 being increased by 1). Alternative: Make copies of d1 and d2 before rolling.
 - ❑ Nonexample: Changing a variable declared in a method (can't access it from outside).

```

/**
 * A Bird class maintains the position of a bird over time
 */
class Bird {

    /**
     * Create a new Bird.
     * @param {Point} startPosition - The 2D starting position of the Bird (x, y)
     * @param {number} startXSpeed - The starting horizontal speed of the bird (pixels/second)
     * @param {number} gravity - The change in the y velocity due to gravity (pixels/second)
     * @param {number} flapUpSpeed - The y velocity (opposite direction of gravity) caused by a flap
     */

    constructor(startPosition, startXSpeed, gravity, flapUpSpeed) {

    }

    /**
     * Updates the position of the bird (both x and y coordinates)
     * @param {number} secondsElapsed - the number of seconds that passed since the last move
     */

    move(secondsElapsed) {

    }

    /**
     * Updates the bird's y velocity caused by a flap given by flapUpSpeed
     */

    flap() {

    }

    /**
     * @type {Point}
     */

    get position() {
        // getter for current position of Bird
    }
}

```

Homework:
Implement Bird
Model Class:
<https://repl.it/LMRV>