Front End
Software Development

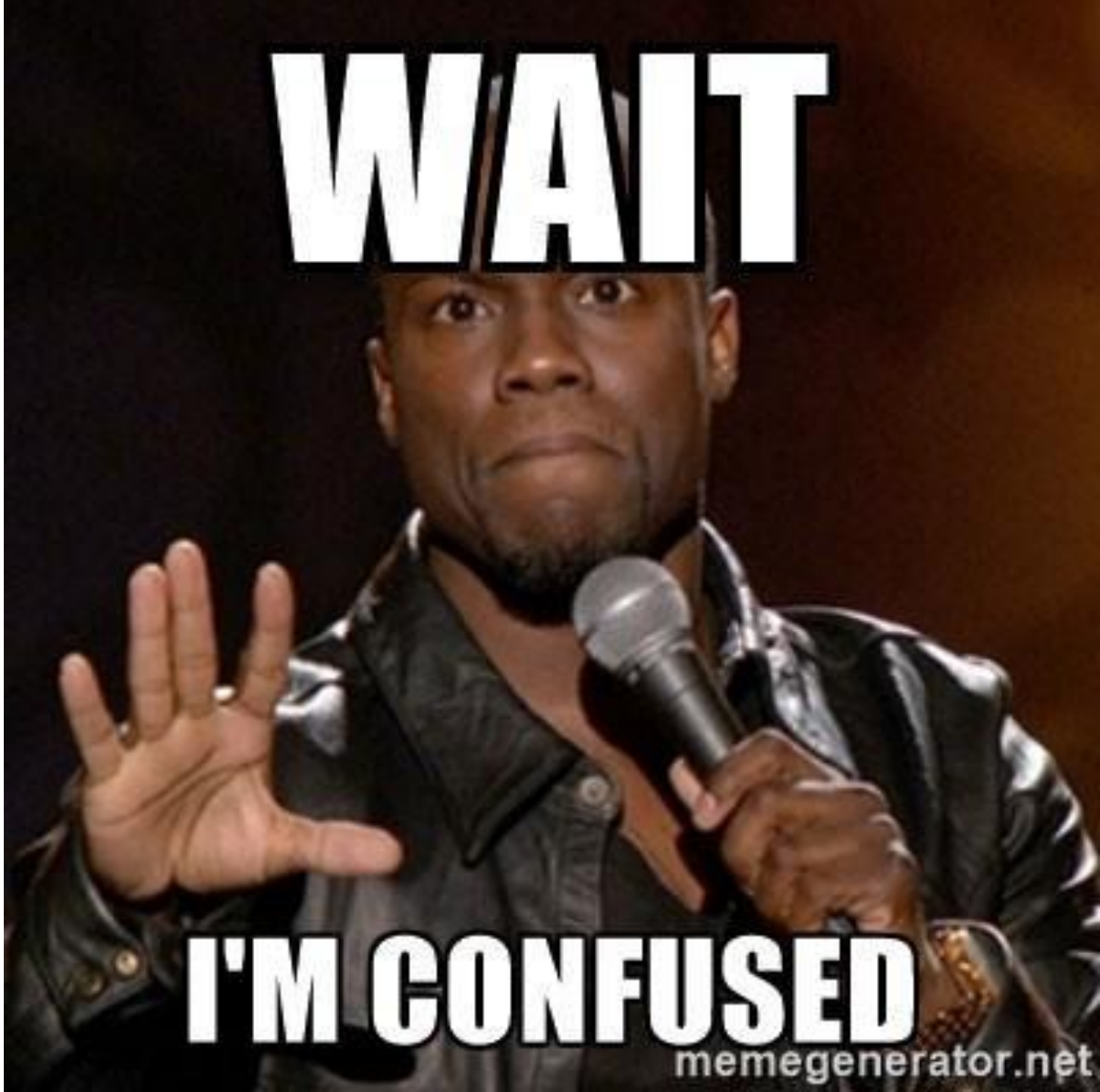
Introduction to JavaScript (weeks 1 - 6)

Week 05

Agenda

- Questions
- Object-Oriented Programming
- Classes
- Inheritance
- Handling Exceptions

Questions



Object **O**riented **P**rogramming

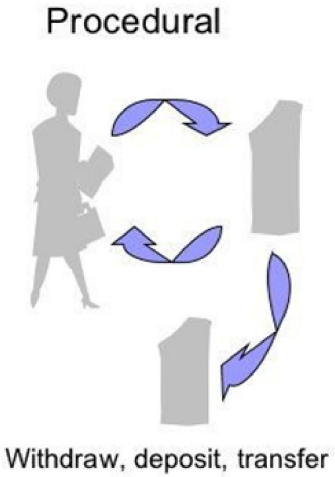
Procedural Programming

Focused on "actions", not "things"

Object Oriented Programming

Focused on "things" (i.e., objects)

- Anatomy of an **`Object`**
  - Properties *(Attributes)*
    - Height
    - Name
    - Color
  - Functionality *(Methods / Actions)*
    - Authorize
    - Stop
    - Go
    - Draw

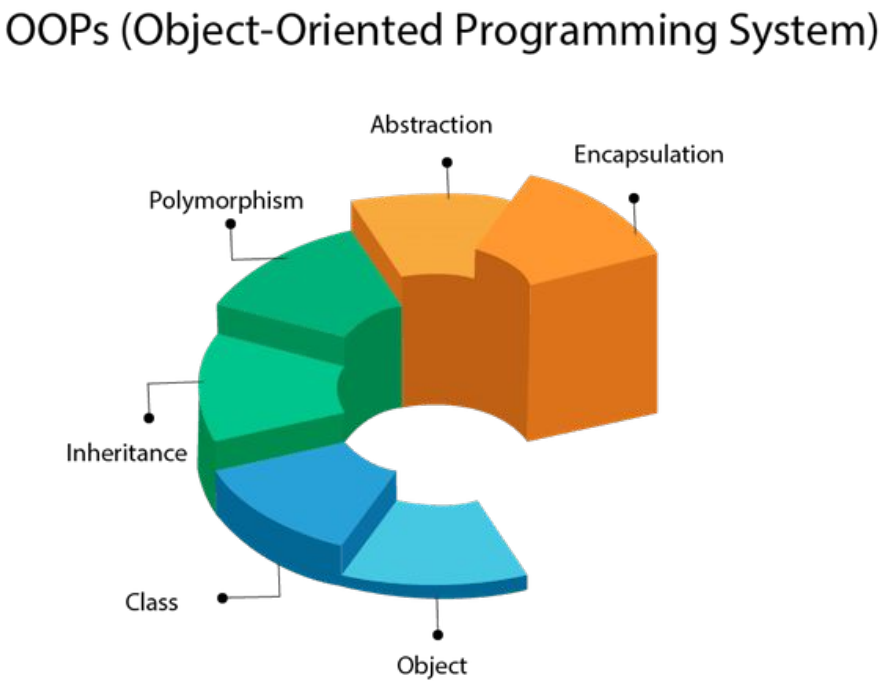


vs

Object Oriented Programming

(concepts)

- OOP Concepts
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
  - `Class`
  - `Object`

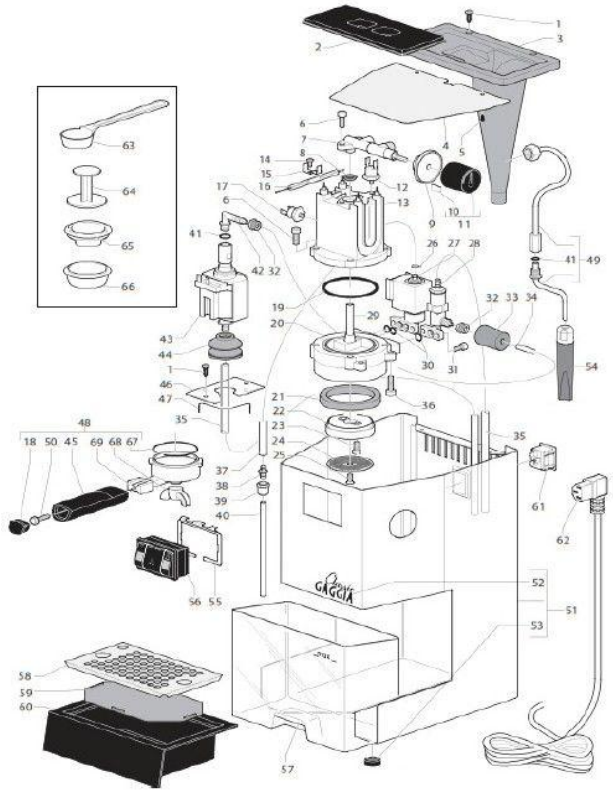


Object Oriented Programming

(abstraction)

- **Abstraction**
  Generalize the implementation details.

  Espresso vs Drip vs Keurig
  They all can `make()` coffee



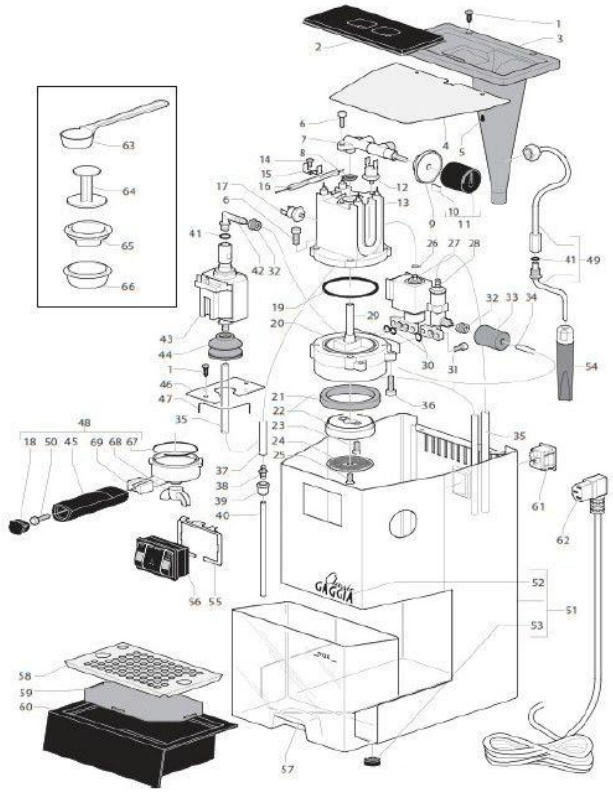
Expose only what you need.

Object Oriented Programming

(encapsulation)

- **Encapsulation**
  Bundle the data and methods that work on that data together.

"Abstraction is a concept which is allowed by encapsulation"



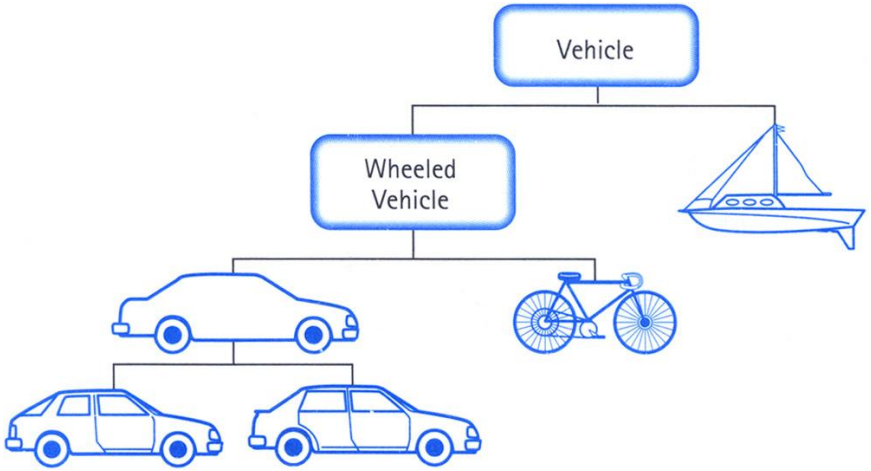
Hide (protect) information.

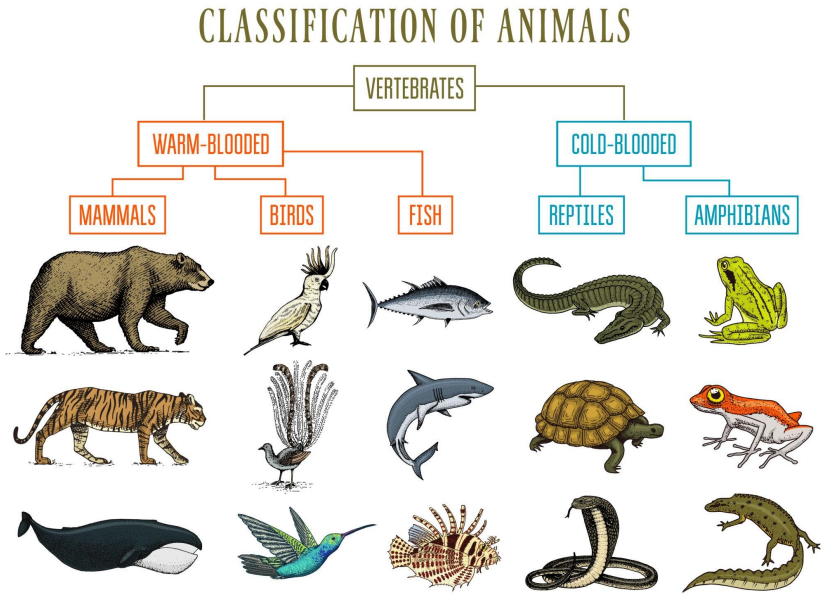
Object Oriented Programming

(inheritance)

- **Inheritance**
  Add features or modify functionality.

  Applied *Taxonomy*



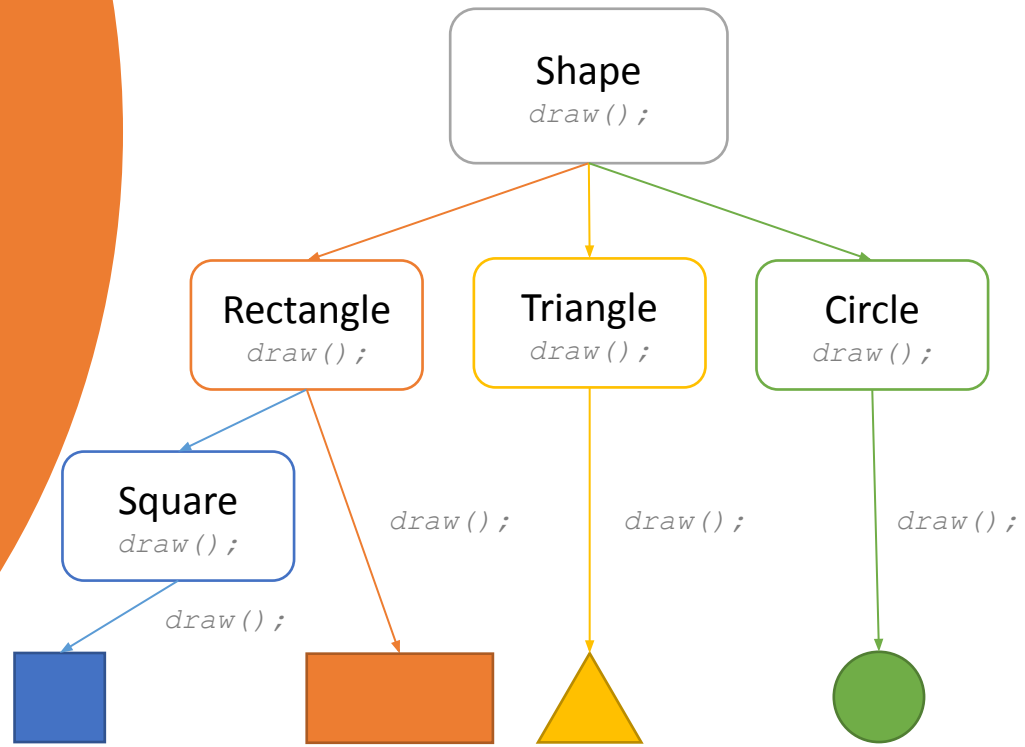


Reuse & share code.

Object Oriented Programming

(polymorphism)

- **Polymorphism**
  Allow different objects to respond or act in different ways to the same input or output.



Quacks like a duck, looks like a duck, then it's a duck...

Classes

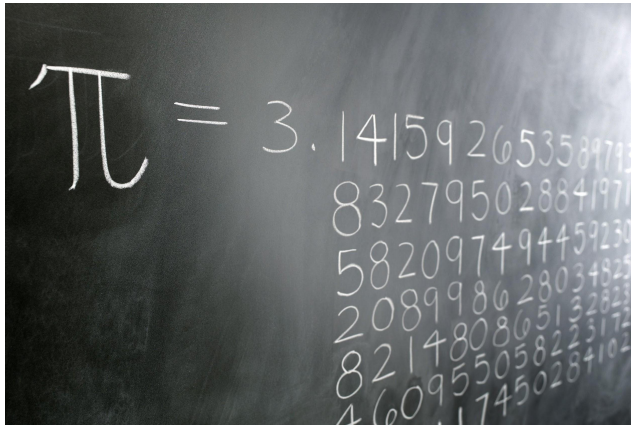
- Allows us to structure our code
  - `Class` is a "blueprint"
  - `object` is an instance or implementation of the blueprint.

The "blueprint" is used to create an `object`.

One "blueprint" can make an *infinite* number of objects.

Classes

(Our First Object)



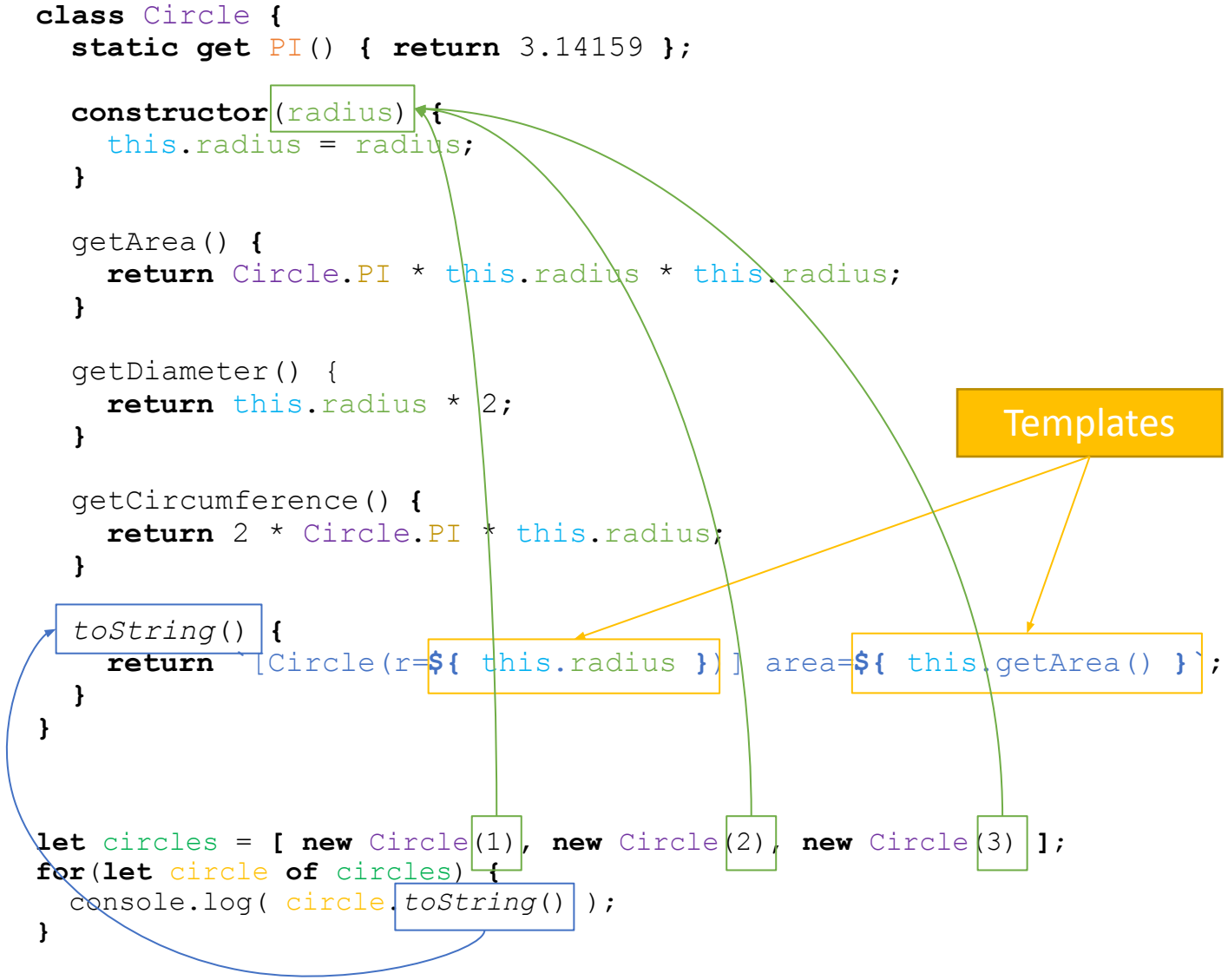
`Circle` Class

Write a `Circle` class that has the following:

- Fields *(Properties)*:
  - **`radius`**—A number that holds the radius of the circle.
  - **`PI`** - A number constant initialized with the value `3.14159`
    Note: We're going to define our own value for PI instead of using `Math.PI`
- Methods:
  - **`getArea`**. Returns the area of the circle, which is calculated as:
    `area = PI * radius * radius`
  - **`getDiameter`**. Returns the diameter of the circle, which is calculated as:
    `diameter = radius * 2`
  - **`getCircumference`**. Returns the circumference of the circle, which is calculated as:
    `circumference = 2 * PI * radius`

Classes

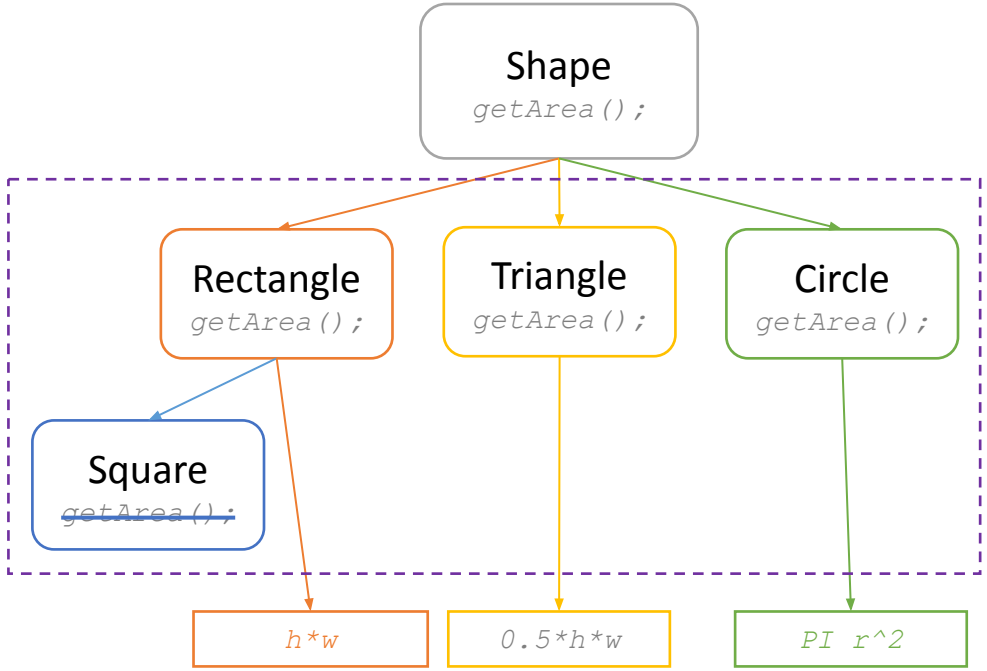
(Circle – A Solution)



Inheritance

Shape Class

Write a Shape class that has a `getArea()` method that each specific formulas for each class.



Child Classes

- `Circle` is a `Shape`
- `Triangle` is a `Shape`
- `Rectangle` is a `Shape`
- `Square` is a `Rectangle` (`Shape` Too!)

Inheritance

(Shapes / Area – A Solution)

```
class Shape {
  toString() {
    return `[${ this.getType() }]
            area=${ this.getArea() }`;
  }
  getArea() {
    return 0;
  }
  getType() {
    return "Shape";
  }
}
```

```
class Rectangle extends Shape {
  constructor(height, width) {
    super();
    this.height = height;
    this.width = width;
  }
  getType() {
    return "Rectangle";
  }
  getArea() {
    return this.height * this.width;
  }
}
```

```
class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }
  getType() {
    return "Circle";
  }
  getArea() {
    return Math.PI * this.radius *
           this.radius;
  }
}
```

```
class Square extends Rectangle {
  constructor(size) {
    super(size, size);
  }
  getType() {
    return "Square";
  }
}
```

```
class Triangle extends Shape {
  constructor(height, width) {
    super();
    this.height = height;
    this.width = width;
  }
  getType() {
    return "Triangle";
  }
  getArea() {
    return 0.5 * this.height *
           this.width;
  }
}
```

```
let shapes = [ new Circle(1), new Rectangle(2,2),
               new Triangle(3, 5), new Square(3) ];
for(let shape of shapes) {
  console.log( shape.toString() );
}
```

```
[Circle] area=3.141592653589793
[Rectangle] area=4
[Triangle] area=7.5
[Square] area=9
```

Handling Exceptions

- Exceptions / Errors
  - Network errors
  - File system errors
- Used for code we don't have control of. Don't use to wrap "bad" code.

Code that MIGHT fail?

```
let connected = false;
try {
  // code that might fail
  openDatabase(() => {
    console.log("Connected...");
    connected = true;
  });
  // any lines after exception are skipped
} catch (err) {
  // called when error
  console.log("An unhandled error occurred. " + err);
} finally {
  // code here runs ALWAYS (optional)
  if (connected) {
    closeDatabase();
  }
}
```

! i.e., Database Not Found

Runs on success OR failure.

Future Assignment

(looking ahead, next week)

WAR!

- Classes
  - `Card`
  - `Deck`
  - `Player`
  - *`Game`?*

- Requirements
  - Deal `26` `Card`s to two `Player`s from a `Deck`.
  - Iterate through the turns where each `Player` plays a `Card`
  - The `Player` who played the higher card is awarded a point
  - A tie result in zero points for either `Player`
  - After all cards have been played, display the score.