# Code Organization

## Where were we?

Concept Roadmap:

1. Bottom-up, i.e., building blocks of languages.
2. **Top-down**, i.e., using building blocks.
    - TODAY: OOP vs. functional
3. Meta-theory.

## Goals

- Compare object-oriented and procedural/functional ways of organizing code.
- Learn the trade-offs of each.
- Learn about the **expression problem**.

## Programs are made up of two things:

- Data
- Code

There are two ways to think of the relationship between the two:

- Package code and data **together** (object-oriented programming).
- Package code and data **separately** (functional and procedural programming).

In [1]:

```
// Familiar OOP
class User {
    private readonly firstName: string;  // Data
    private readonly lastName: string;   // Data

    constructor(firstName: string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public fullName(): string {  // Code/method
        return this.firstName + " " + this.lastName;
    }
}
```

In [2]:

```
const user = new User("John", "Smith");
user.fullName();
```

```
John Smith
```

In [3]:

```
// Strange OOP
class User {
    public readonly firstName: string;  // Data
    public readonly lastName: string;   // Data

    constructor(firstName: string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}

function fullName(user: User): string {  // Code
    return user.firstName + " " + user.lastName;
}
```

In [4]:

```
const user = new User("John", "Smith");
fullName(user);
```

```
John Smith
```

In [5]:

```
// Familiar ADT
// ---------------------------------------------------
// File 1
type User = {
    firstName: string,  // Data
    lastName: string    // Data
};

function newUser(firstName: string, lastName: string): User {
    // Replaces constructor
    return { firstName: firstName, lastName: lastName };
}
// ---------------------------------------------------


// ---------------------------------------------------
// File 2
function fullName(user: User): string {  // Code
    return user.firstName + " " + user.lastName;
}
// ---------------------------------------------------
```

In [6]:

```
const user = newUser("John", "Smith");
fullName(user);
```

```
John Smith
```

## A more detailed comparison

Let's compare and contrast implementing a library of shapes with OOP and functional styles.

In [7]:

```
interface Shape {
    /**
     * Return the area of the shape, in whatever units it was defined in.
     */
    area(): number;
    /**
     * Return a string that represents the shape type and its parameter.
     */
    print(): string;
}
```

### Shape 1: Box

In [8]:

```
class Box implements Shape {
    public readonly width: number;
    public readonly height: number;

    constructor(width: number, height: number) {
        this.width = width;
        this.height = height;
    }

    area(): number {
        return this.width*this.height;
    }

    print(): string {
        return "Box(" + this.width + "," + this.height + ")";
    }
}
```

In [9]:

```
// interface Shape {
//     /**
//      * Return the area of the shape, in whatever units it was defined in.
//      */
//     area(): number;
//     /**
//      * Return a string that represents the shape type and its parameter.
//      */
//     print(): string;
// }

// Analog of interface Shape
type Shape = {
    area: () => number,
    print: () => string
};
```

In [10]:

```
// Simulating constructor, new Box(width, height)
// newBox(width, height)
function newBox(width: number, height: number): Shape {
    type _Box = { width: number, height: number };

    // fakeThis analogous to this
    // constructor(width: number, height: number) {
    //     this.width = width;
    //     this.height = height;
    // }
    const fakeThis = { width: width, height: height };

    // area(): number {
    //     return this.width*this.height;
    // }
    function area(fakeThis: _Box): number {
        return fakeThis.width * fakeThis.height;
    }

    //  print(): string {
    //     return "Box(" + this.width + "," + this.height + ")";
    //  }
    function print(fakeThis: _Box): string {
        return "Box(" + fakeThis.width + "," + fakeThis.height + ")";
    }

    return {
        area: () => area(fakeThis),
        print: () => print(fakeThis)
    };
}
```

In [11]:

```
const b = newBox(2, 3);
[b.area(), b.print()]
```

```
[ 6, 'Box(2,3)' ]
```

In [12]:

```
class Box implements Shape {
    public readonly width: number;
    public readonly height: number;

    constructor(width: number, height: number) {
        this.width = width;
        this.height = height;
    }

    area(): number {
        return this.width*this.height;
    }

    print(): string {
        return "Box(" + this.width + "," + this.height + ")";
    }
}
```

In [13]:

```
const box = new Box(2, 3);
[box.area(), box.print()]
```

```
[ 6, 'Box(2,3)' ]
```

### Shape 2: Circle

In [14]:
```
class Circle implements Shape {
    private readonly radius: number;

    constructor(radius: number) {
        this.radius = radius;
    }

    // Method
    area(): number {
        // pi r^2
        return Math.PI*this.radius*this.radius;
    }

    // Method
    print(): string {
        return "Circle(" + this.radius + ")";
    }
}
```

In [15]:
```
const circle = new Circle(4);
[circle.area(), circle.print()]
```

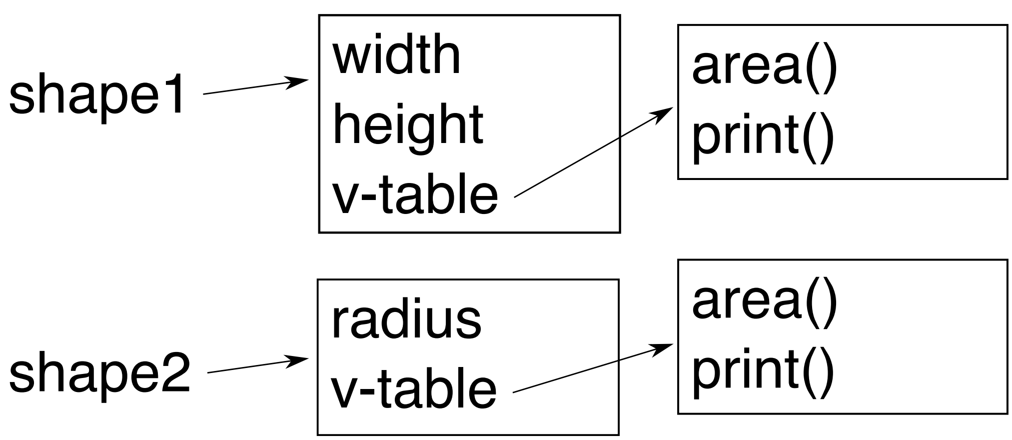
```
[ 50.26548245743669, 'Circle(4)' ]
```

### Mixing Shapes

In [16]:
```
const shapes: Shape[] = [
    new Box(2, 3),
    new Circle(4),
];

for (const shape of shapes) {
    console.log(shape.area());
    console.log(shape.print());
}
```

```
6
Box(2,3)
50.26548245743669
Circle(4)
```

This is called ***dynamic dispatch***. When we call `shape.area()` , the system figures out which one we want (the box or circle one) depending on the type of the object. The word ***dispatch*** means to send something (in this case a message to the function), and ***dynamic*** means that it's not until the code runs that we know which one to call.




**More on dynamic dispatch**

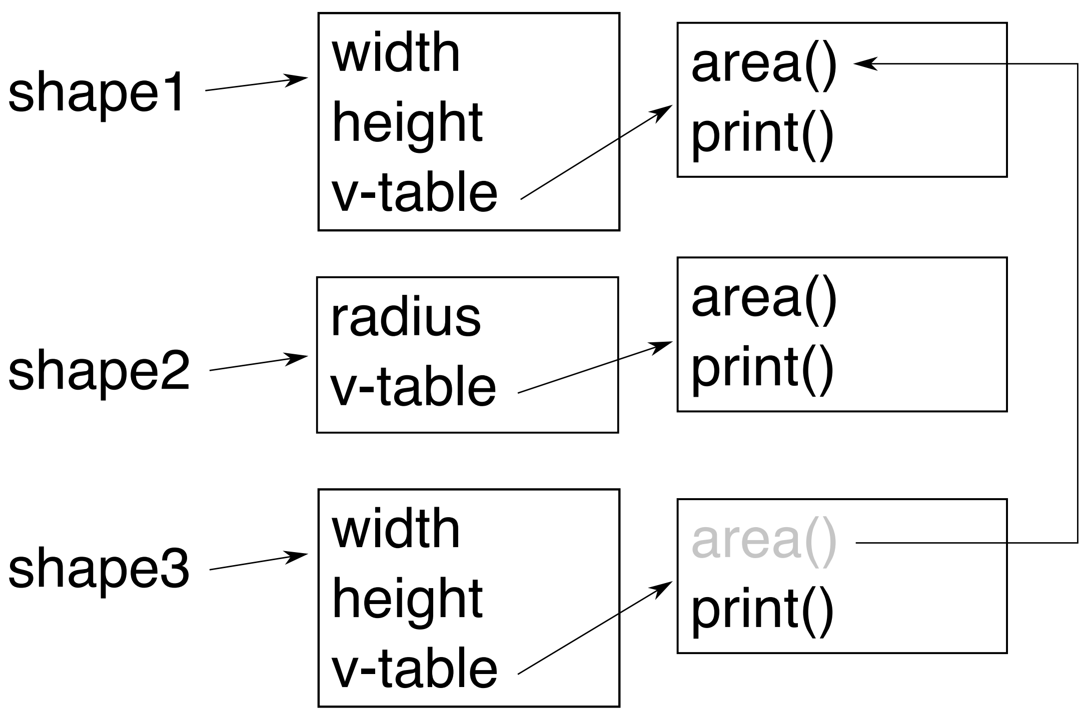
In [17]:
```
class Square extends Box {
    constructor(size: number) {
        super(size, size);
    }

    print(): string {
        // 1: Override
        return "Square(" + this.width + ")";
    }

    // 2: inherit area
}
```

In [18]:
```
const shape3: Shape = new Square(5);
[shape3.area(), shape3.print()]
```

```
[ 25, 'Square(5)' ]
```

The `Square` class overrides the constructor to force the width and height to be the same, and overrides the `print()` method to make it clear that it's a square. The `area()` method can be left alone since it already computes the right thing. Objects of the type `Square` will just use the superclass `Box` 's `area()` method.

Our heap now looks like this:




- Why does the new square have both a width and height?
- Why is the square's `area()` method gray and why does it point to the box's v-table?

**Remember we can encode classes with closures**

The vtable in code ...

In [19]:
```
type Shape = {
    tag: string,                  // tells the objects at runtime I'm box, circle, ...
    [id: string]: any,            // contains fields of object
    vtable: { [id: string]: any }  // contains area, print
};

function newBox(width: number, height: number): Shape {
    return {
        tag: "BOX",
        width: width,    // field 1
        height: height,  // field 2
        vtable: {
            area: () => width*height,
            print: () => "Box(" + width + "," + height + ")"
        }
    };
}

function newCircle(radius: number): Shape {
    return {
        tag: "CIRCLE",
        radius: radius,  // field 1
        vtable: {
            area: () => Math.PI*radius*radius,
            print: () => "Circle(" + radius + ")"
        }
    };
}
```

In [20]:
```
// If you didn't have inheritance, you might write code that looks like this
function newSquare(size: number): Shape {
    const box = newBox(size, size);
    return {
        tag: "SQUARE",
        box: box,
        vtable: {
            area: () => box.vtable.area(),        // inheritance
            print: () => "Square(" + size + ")"   // override print
        }
    };
}
```

### What's the functional approach look like?

In [21]:
```
type Box = {
    tag: "BOX",
    width: number,
    height: number
};

function newBox(width: number, height: number): Box {
    return { tag: "BOX", width: width, height: height };
}
```

In [22]:
```
type Circle = {
    tag: "CIRCLE",
    radius: number
};

function newCircle(radius: number): Circle {
    return { tag: "CIRCLE", radius: radius };
}
```

In [23]:
```
type Shape = Box | Circle;  // Algebraic data-type
```

In [24]:
```
function area(shape: Shape): number {
    switch (shape.tag) {  // Dynamic dispatch in code
        case "BOX": {
            return shape.width*shape.height;
        }
        case "CIRCLE": {
            return Math.PI*shape.radius*shape.radius;
        }
    }
}
```

In [25]:
```
function print(shape: Shape): string {
    switch (shape.tag) {  // Dynamic dispatch in code
        case "BOX": {
            return "Box(" + shape.width + "," + shape.height + ")";
        }
        case "CIRCLE": {
            return "Circle(" + shape.radius + ")";
        }
    }
}
```

## Tradeoffs between how we package code and data

### Challenge 1: Creating new Shapes

In [26]:
```
interface Shape {
    /**
     * Return the area of the shape, in whatever units it was defined in.
     */
    area(): number;
    /**
     * Return a string that represents the shape type and its parameter.
     */
    print(): string;
}
```

In [27]:
```
class Triangle implements Shape {
    public readonly base: number;
    public readonly height: number;

    constructor(base: number, height: number) {
        this.base = base;
        this.height = height;
    }

    area(): number {
        return 0.5*this.base*this.height;
    }

    print(): string {
        return "Triangle(" + this.base + ", " + this.height + ")";
    }
}
```

In [28]:
```
const shape4: Shape = new Triangle(6, 7);
[shape4.area(), shape4.print()]
```

```
[ 21, 'Triangle(6, 7)' ]
```

#### That was easy in OO style, what about Functional?

In [29]:
```
type Triangle = {
    tag: "TRIANGLE",
    base: number,
    height: number
};

function Triangle(base: number, height: number): Triangle {
    return { tag: "TRIANGLE", base: base, height: height };
}
```

In [30]:
```
type Shape = Box | Circle | Triangle;  // Algebraic data-type, have to modify this

function area(shape: Shape): number {
    switch (shape.tag) {  // Dynamic dispatch in code
        case "BOX": {
            return shape.width*shape.height;
        }
        case "CIRCLE": {
            return Math.PI*shape.radius*shape.radius;
        }
        // Add this
        case "TRIANGLE": {  // Have to modify this
            return 0.5*shape.base*shape.height;
        }
    }
}

function print(shape: Shape): string {
    switch (shape.tag) {  // Dynamic dispatch in code
        case "BOX": {
            return "Box(" + shape.width + "," + shape.height + ")";
        }
        case "CIRCLE": {
            return "Circle(" + shape.radius + ")";
        }
        // Add this
        case "TRIANGLE": {  // Have to modify this
            return "Triangle(" + shape.base + ", " + shape.height + ")";
        }
    }
}
```

**Summary**

- When adding a new shape, you got more things "for free" in OO
- In particular, you did not need to modify the area and print function.

### Challenge 2: Adding new functionality on shape

In [31]:
```
interface Shape {
    /**
     * Return the area of the shape, in whatever units it was defined in.
     */
    area(): number;
    /**
     * Return a string that represents the shape type and its parameter.
     */
    print(): string;
    /**
     * Return the perimeter of the shape, in whatever units it was defined in.
     */
    perimeter(): number;
}
```

In [32]:
```
class Box implements Shape {
    public readonly width: number;
    public readonly height: number;

    constructor(width: number, height: number) {
        this.width = width;
        this.height = height;
    }

    area(): number {
        return this.width*this.height;
    }

    print(): string {
        return "Box(" + this.width + "," + this.height + ")";
    }

    perimeter(): number {  // have to modify this
        return 2*this.width + 2*this.height;
    }
}
```

In [33]:
```
class Circle implements Shape {
    public readonly radius: number;

    constructor(radius: number) {
        this.radius = radius;
    }

    area(): number {
        return Math.PI*this.radius*this.radius;
    }

    print(): string {
        return "Circle(" + this.radius + ")";
    }

    perimeter(): number {  // have to modify this
        return Math.PI*this.radius*2.0;
    }
}
```

**Ok, what about the functional case?**

In [34]:
```
type Box = {
    tag: "BOX",
    width: number,
    height: number
};

type Circle = {
    tag: "CIRCLE",
    radius: number
};

function Box(width: number, height: number): Box {
    return { tag: "BOX", width: width, height: height };
}

function Circle(radius: number): Circle {
    return { tag: "CIRCLE", radius: radius };
}
```

In [35]:
```
type Shape = Box | Circle;  // Algebraic data-type

function area(shape: Shape): number {  // Unmodified
    switch (shape.tag) {  // Dynamic dispatch in code
        case "BOX": {
            return shape.width*shape.height;
        }
        case "CIRCLE": {
            return Math.PI*shape.radius*shape.radius;
        }
    }
}

function print(shape: Shape): string {  // Unmodified
    switch (shape.tag) {  // Dynamic dispatch in code
        case "BOX": {
            return "Box(" + shape.width + "," + shape.height + ")";
        }
        case "CIRCLE": {
            return "Circle(" + shape.radius + ")";
        }
    }
}
```

In [36]:
```
function perimeter(shape: Shape): number {  // Just add new function
    switch (shape.tag) {  // Dynamic dispatch in code
        case "BOX": {
            return 2*shape.height + 2*shape.width;
        }
        case "CIRCLE": {
            return 2*shape.radius*Math.PI;
        }
    }
}
```

## Summary

When adding shapes

- OO: could create a new subclass independently of existing shapes.
- Functional: had to modify every function.

When adding functionality

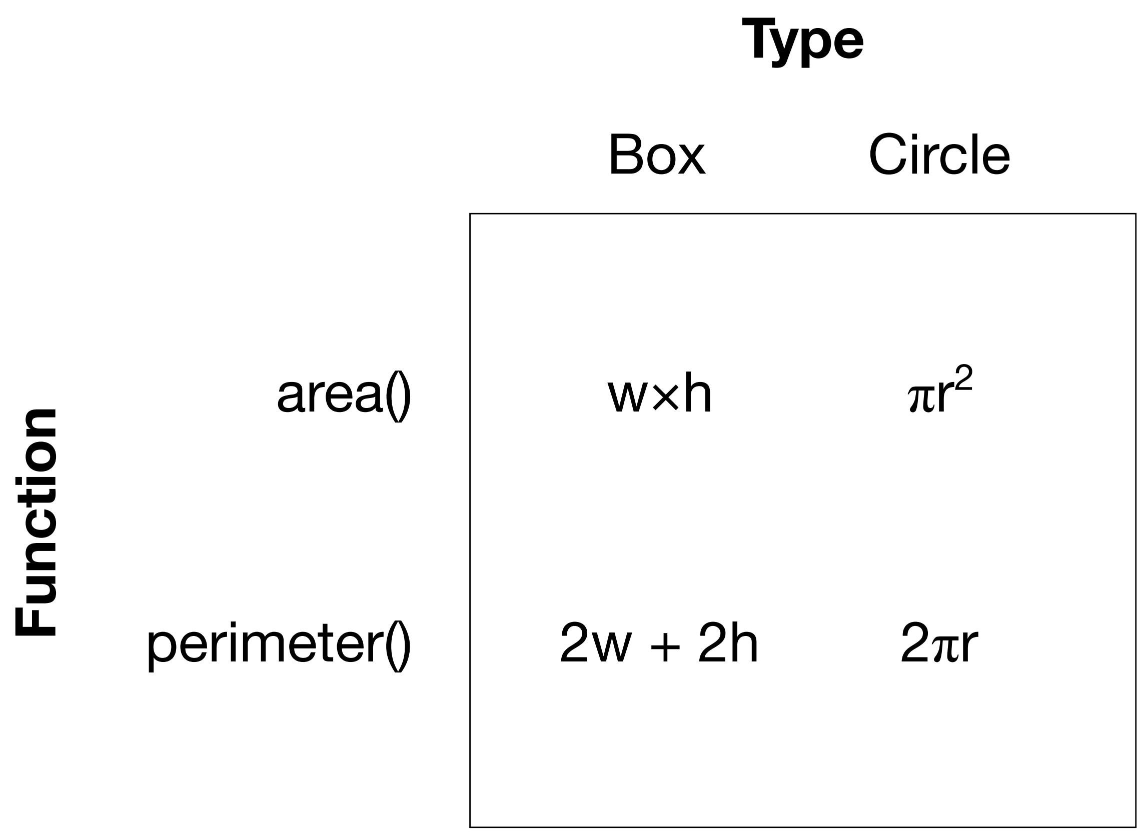
- OO: had to modify every class.
- Functional: could create a new function independently of existing functions.

## OOP vs. Functional / Procedural

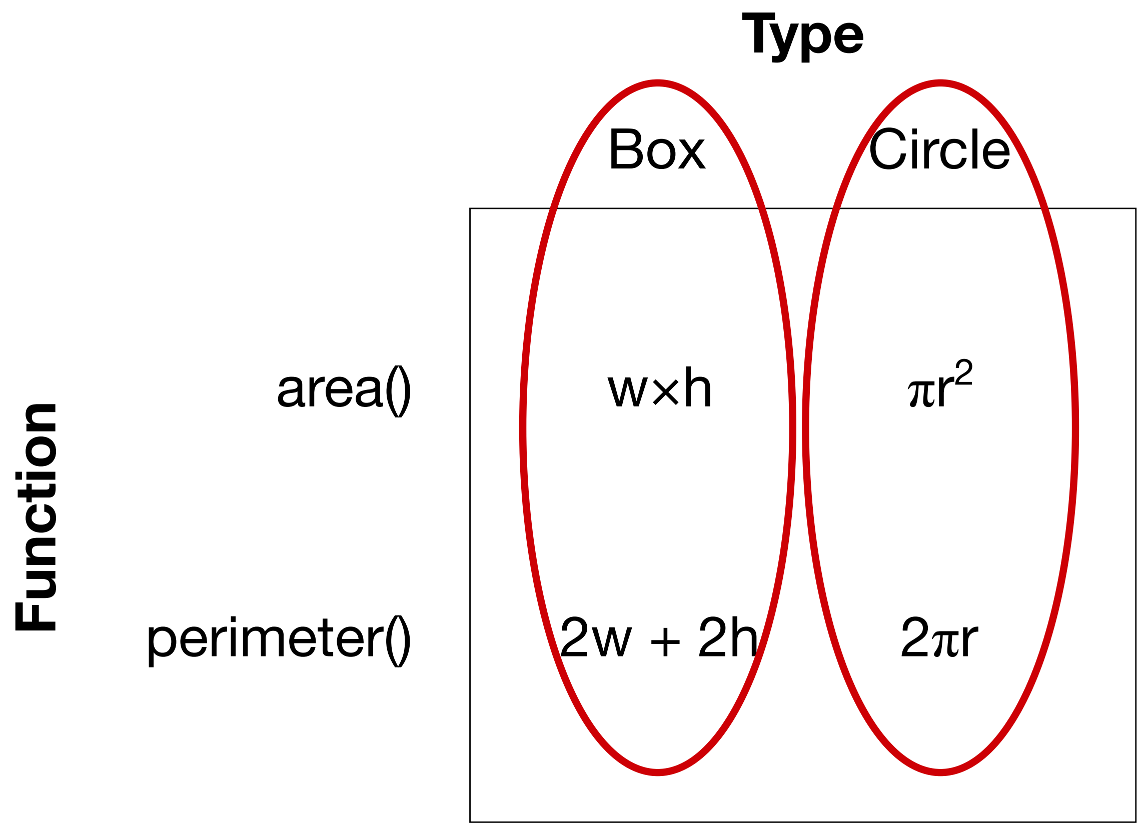
When should we use object-oriented programming and when should we use procedural?

- In OOP, adding types of objects (classes) is easy, but adding new types of functionality is awkward.
- In procedural or functional, it's the reverse.

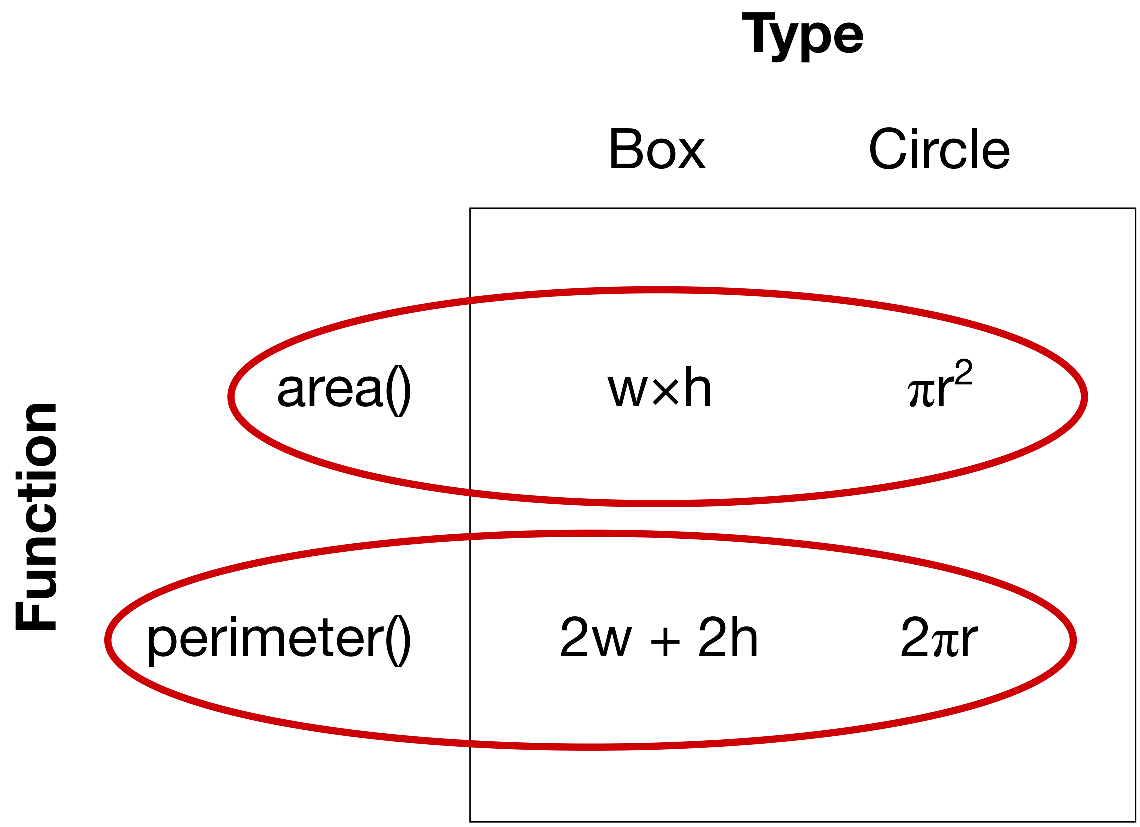
The dilemma is called the ***expression problem***.




With object-oriented programming, we group the functions by type:




This makes it easy to add types, but awkward to add functions. With procedural or functional programming, we group functions by function:




This makes it easy to add functions, but awkward to add types.

### How do I choose?

- If you're in an OO langauge, use OO ...
- If you're in a functional language, use functional ...

```
In [ ]:
```