Object-Oriented Programming

Lecture 17

Data Buddies Survey

Undergraduate Survey
http://bit.ly/CSundergraduate

Graduate Survey
http://bit.ly/CSgraduate

What is it?

- Anonymous survey provided by CRA open now through Oct. 31st

Why is it important?

- Your feedback gives department real-time data on curriculum, pedagogy, student support and cultural climate from student POV

What's in it for you?

- Completion of survey means raffle entry and chance to win Amazon gift card *(dept to raffle more than $1K in gift cards)*

Check your email for more details

Announcements

- PS04 – Linked List Functions
  - Due Friday at 11:59pm
  - Only five functions… but they're conceptually very challenging. Start today.

- Next WS will go out soon!
  - More practice with recursion

PollEv: Given the class and functions below, what is the output when the code right runs?

```
class Point {
    x: number = 0;
    y: number = 0;
}

function shiftX(p: Point, amount: number): void {
    p.x = p.x + amount;
}

function toString(p: Point): string {
    return p.x + ", " + p.y;
}
```

```
let a: Point = new Point();
a.x = 110;
a.y = 110;
shiftX(a, 10);
print(toString(a));

let b: Point = new Point();
b.x = 401;
b.y = 401;
shiftX(b, -10);
print(toString(b));
```

Hands-on #1) Refresher practice working with objects

- In lec-17 / 00-simple-class-app.ts:

1. **Initialize** the variable **a** to be a new Point object.
2. **Assign** different values to **a**'s **x** and **y** properties.
3. **Print a**'s **x** and **y** properties in the format of: **"<x>, <y>"**

- Check-in on PollEv.com/comp110 once complete.

```
a = new Point();
a.x = 110;
a.y = 110;
print(a.x + ", " + a.y);
```

Review of **Classes** and **Objects**

- A class defines a new **Data Type**
  - The class definition specifies properties
- Instances of a class are called **objects**
  - To create an object you must use the **new** keyword: **`new <Classname>()`**
- *Every object of a class* has the **same properties** but has **its own values**
- Objects are reference-types
  - variables do not hold objects, but rather ***references to objects***

Object-oriented Programming

- So far we've used objects as compound data types
  - i.e. to model a row of data in a spreadsheet
- We've written functions, separate from classes, that operate on objects
- The only thing we've been able to *do* with an "object" is access and assign values to its properties
- Object-oriented programming allows us to give objects *capabilities*
  - We'll do this with two special kinds of functions: methods and constructors

Functions vs. **Methods**

1. Let's define a *silly* **function.**

```
function sayHello(): void {
    print("Hello, world");
}
```

2. Once defined, we can then call it.

```
sayHello();
```

3. Now, let's define that same function as a **method** *of the* `Point` *class*.

```
class Point {
    // ... properties elided...

    sayHello(): void {
        print("Hello, world");
    }
}
```

4. Once defined, we can call this method on any `Point` object:

```
let a: Point = new Point();
a.sayHello();
```

Introducing: **Methods**

- A **method** is a function defined in a class.
  - Everything you know about a function's parameters, return types, and evaluation rules are the same with methods.
  - *Syntactically*, the only difference is you do not write the keyword "`function`".
- Once defined, you can call a method on any object of that class using the dot operator.
  - Just like how properties were accessed except followed by parenthesis and any necessary arguments
- Methods have one special feature beyond plain functions...

```
class Point {

    // Properties Elided

    <name>(<parameters>): <returnType> {
        <method body>;
    }

}
```

```
let a: Point = new Point();
print(a.methodName());
```

Follow-along: Simple Method App

- Let's implement and call the sayHello method example from previous slides in 01-simple-method-app.ts

```
class Point {
    // ... properties elided...

    sayHello(): void {
        print("Hello, world");
    }
}
```

```
let a: Point = new Point();
a.sayHello();
```

Method's Special Feature:

Methods can refer to the object the method was called on.

Consider this plain **function.** Notice that its parameter **p** is *a reference* to a Point object.

```
function toString(p: Point): string {
   return p.x + ", " + p.y;
}
```

To call it, we would pass a reference to a Point object as an argument.

```
let a: Point = new Point();
print(toString(a));
```

It turns out we *can* write a method that does the same thing and it can be called like the example to the right.

```
let a: Point = new Point();
print(a.toString());
```

How can this magic work???

Method's Special Feature:

Methods can refer to the object the method was called on.

When a method is called, inside of the function, a special "variable" is initialized named **`this`**

The **`this`** keyword ***refers to*** the object the method was called upon.

```
class Point {

    // ... Properties Elided ...

    toString(): string {
        return this.x + ", " + this.y;
    }

}
```

```
let a: Point = new Point();
a.x = 110;
a.y = 110;
print(a.toString());
```

When the above code jumps to *toString*, **this** will refer to the same Point object **a** refers to.

```
let b: Point = new Point();
b.x = 401;
b.y = 401;
print(b.toString());
```

When the above code jumps to *toString*, **this** will refer to the same Point object **b** refers to.

Follow-along: Practice with the **this** keyword

- In 02-this-keyword-app.ts...

1. At TODO#1, define the **`toString` method** to the right.

2. In the `main` function, at TODO's #2 and #3, call the **`toString`** method on **`Points a`** and **`b`** respectively.

```
class Point {

    // ... Properties Elided ...

    toString(): string {
        return this.x + ", " + this.y;
    }

}
```

Hands-on: Practice with the **this** keyword

- In 02-this-keyword-app.ts, let's make it easy to shift a Point's **x** property.

1. At TODO#4, define a method named **`shiftX,`** that has a single number parameter named **amount** and a **`void`** return type.

2. Increment the **`x`** property of the object **shiftX** is called on by **amount**.

3. Call **`shiftX`** on **`Points a`** and **`b`** in the `main` function.

4. Once you've tested that it works, check-in on PollEv.com/comp110

```
shiftX(amount: number): void {
    this.x = this.x + amount;
}
```

```
b.shiftX(10);
```

Follow-Along: Distance Method

- Let's add a method to compute the distance between two points.
- We'll specify the 2nd point as a parameter named *other*.
- We'll also make use of some special Math functions:
  - Math.pow(x, n) raises **x** to the **n**th
  - Math.sqrt(x) computes square root

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
class Point {
    // … elided …
    distance(other: Point): number {
        let xDelta2: number = Math.pow(other.x - this.x, 2);
        let yDelta2: number = Math.pow(other.y - this.y, 2);
        return Math.sqrt(xDelta2 + yDelta2);
    }
}
```

```
// Calling the distance method
print(a.distance(b));
```

Why have both functions and methods?

- Different schools of thought in *functional programming-style (FP)* versus *object-oriented programming-style (OOP).*
  - *Both are equaly **capable**, but some problems are better suited for one style vs. other.*
- FP tends to shine with *data processing* problems
  - Data analysis programs like *weather stats* and *cpu hat heist* are natural fits
- OOP is great for *graphics, long-running programs / simulations, systems*
- Methods allow us to build and package functionality *into* objects.
  - You don't need to import extra functions to work with an object, they are bundled.
  - As programs grow in size, methods and OOP have some extra capabilities to help teams of programmers avoid accidental errors. You'll see this in 401!

Constructors

- An object's properties must be initialized before the object is usable
- A constructor allows us to both
  1. Specify unique initial values of properties upon construction
  2. Require certain properties are initialized
- A constructor is just a special function
  - Does *not* use the keyword "function"
  - Name is **`constructor`**
  - Special, self-referencing variable named **`this`**
  - No return type
- A class' constructor is called each time the **`new <Classname>`** expression is evaluated.

Before

```
let a: Point = new Point();
a.x = 10;
a.y = 0
```

Defining a constructor

```
class Point {

    x: number;
    y: number;

    constructor(x: number, y: number) {
        this.x = x;
        this.y = y;
    }

}
```

After

```
let a: Point = new Point(10, 0);
```

Follow-along: Constructors

- Let's open 03-constructor-app.ts
- We'll add the constructor from the previous slide.
- Then we'll need to update where we call the constructor from.

Addition Assignment Operator

- Changing a number variable's value by some amount in this way:

```
x = x + amount;
```

- TypeScript and many other languages, have a built in "addition assignment" operator:

```
x += amount;
```

- The two statements above achieve the same outcome.

- There are subtraction assignment **-=**, multiplication assignment ***=**, concatenation assignment **+=**, operators that work just the same way, as well.

`toString` is a "Magic Method"

- We deliberately defined the **`toString`** method earlier using a convention.
- Any class that has a method named **`toString`**, with no parameters, and a return type of `string`, has a superpower…
- We can print an object directly:

```
let a: Point = new Point(5, 10);
print(a); // toString is "automagically called"
let s: string = "a: " + a; // toString is "automagically" called
print(s);
```