Working with Objects

JavaScript objects are a collection of properties, each of which has a name and value. The simplest way to create an object is to use the literal syntax

```
let hat = {
    name: "Hat",
    price: 100
};

let boots = {
    name: "Boots",
    price: 100
}

console.log('Name: ${hat.name}, Price: ${hat.price}');
console.log('Name: ${boots.name}, Price: ${boots.price}');
```

O/p:

```
Name: Hat, Price: 100
Name: Boots, Price: 100
```

```
let hat = {
    name: "Hat",
    price: 100
};

let boots = {
    name: "Boots",
    price: 100
}

function printDetails(product : { name: string, price: number}) {
    console.log('Name: ${product.name}, Price: ${product.price}');
}

printDetails(hat);
printDetails(boots);
```

Defining Optional Properties in a Type Annotation

A question mark can be used to denote an optional property, as shown in Listing 4-22, allowing objects that don't define the property to still conform to the type.

```
let hat = {
    name: "Hat",
    price: 100
};

let boots = {
    name: "Boots",
    price: 100,
    category: "Snow Gear"
}

function printDetails(product : { name: string, price: number, category?:
string}) {
    if (product.category != undefined) {
        console.log('Name: ${product.name}, Price: ${product.price}, ' +
            'Category: ${product.category}');
    } else {
        console.log('Name: ${product.name}, Price: ${product.price}');
    }
}

printDetails(hat);
printDetails(boots);
```

The type annotation adds an optional `category` property, which is marked as optional. This means that the type of the property is `string | undefined`

O/p:

```
Name: Hat, Price: 100
Boots, Price: 100, Category: Snow Gear
```

Defining Classes

Classes are templates used to create objects, providing an alternative to the literal syntax. Support for classes is a recent addition to the JavaScript specification and is intended to make working with JavaScript more consistent with other mainstream programming languages

```
class Product {

    constructor(name: string, price: number, category?: string) {
        this.name = name;
        this.price = price;
        this.category = category;
    }

    name: string
    price: number
    category?: string
}

let hat = new Product("Hat", 100);
```

```
let boots = new Product("Boots", 100, "Snow Gear");

function printDetails(product : { name: string, price: number, category?:
string}) {
    if (product.category != undefined) {
        console.log('Name: ${product.name}, Price: ${product.price}, ' +
            'Category: ${product.category}');
    } else {
        console.log('Name: ${product.name}, Price: ${product.price}');
    }
}

printDetails(hat);
printDetails(boots);
```

The `constructor` function is invoked when a new object is created using the class, and it provides an opportunity to receive data values and do any initial setup that the class requires. In the example, the constructor defines `name`, `price`, and `category` parameters that are used to assign values to properties defined with the same names.

The `new` keyword is used to create an object from a class, like this:

```
let hat = new Product("Hat", 100);
```

This statement creates a new object using the `Product` class as its template. `Product` is used as a function in this situation, and the arguments passed to it will be received by the `constructor` function defined by the class. The result of this expression is a new object that is assigned to a variable called `hat`.

O?p:

```
Name: Hat, Price: 100
Name: Boots, Price: 100, Category: Snow Gear
```

Adding Methods to a Class

I can simplify the code in the example by moving the functionality defined by the `printDetails` function into a method defined by the `Product` class

```
class Product {

    constructor(name: string, price: number, category?: string) {
        this.name = name;
```

```
        this.price = price;
        this.category = category;
    }

    name: string
    price: number
    category?: string

    printDetails() {
        if (this.category != undefined) {
            console.log('Name: ${this.name}, Price: ${this.price}, ' +
                'Category: ${this.category}');
        } else {
            console.log('Name: ${this.name}, Price: ${this.price}');
        }
    }
}

let hat = new Product("Hat", 100);

let boots = new Product("Boots", 100, "Snow Gear");

// function printDetails(product : { name: string, price: number,
category?: string}) {
//     if (product.category != undefined) {
//         console.log('Name: ${product.name}, Price: ${product.price}, ' +
//             'Category: ${product.category}');
//     } else {
//         console.log('Name: ${product.name}, Price: ${product.price}');
//     }
// }

hat.printDetails();
boots.printDetails();
```

Methods are invoked through the object, like this:

```
...
hat.printDetails();
...
```

The method accesses the properties defined by the object through the `this` keyword:

```
...
console.log('Name: ${this.name}, Price: ${this.price}');
...
```

This example produces the following output in the browser's JavaScript console:

```
Name: Hat, Price: 100
Name: Boots, Price: 100, Category: Snow Gear
```

Access Controls and Simplified Constructors

TypeScript provides support for access controls using the `public`, `private`, and `protected` keywords. The `public` class gives unrestricted access to the properties and methods defined by a class, meaning they can be accessed by any other part of the application. The `private` keyword restricts access to features so they can be accessed only within the class that defines them. The `protected` keyword restricts access so that features can be accessed within the class or a subclass.

```
class Product {

    constructor(public name: string, public price: number, public
category?: string) {
        // this.name = name;
        // this.price = price;
        // this.category = category;
    }

    // name: string
    // price: number
    // category?: string

    printDetails() {
        if (this.category != undefined) {
            console.log('Name: ${this.name}, Price: ${this.price}, ' +
                'Category: ${this.category}');
        } else {
            console.log('Name: ${this.name}, Price: ${this.price}');
        }
    }
}

let hat = new Product("Hat", 100);

let boots = new Product("Boots", 100, "Snow Gear");

hat.printDetails();
boots.printDetails();
```

Adding one of the access control keywords to a constructor parameter has the effect of creating a property with the same name, type, and access level. So, adding the `public` keyword to the `price` parameter, for example, creates a `public` property named `price`, which can be assigned `number` values. The value received through the constructor is used to initialize the property. This is a useful feature that eliminates the need to copy parameter values to initialize properties, and it is a feature that I wish other languages would adopt.