CMSC335

Web Application Development with JavaScript

Object and Custom Types

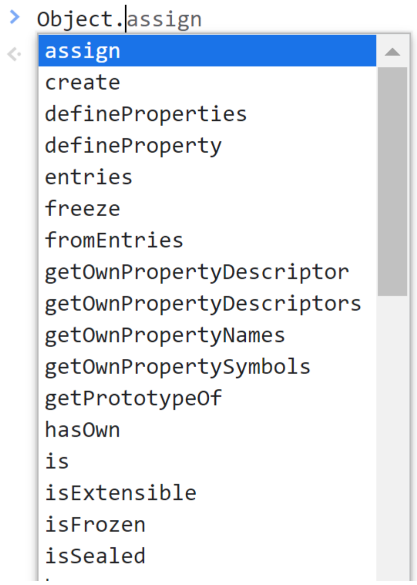
Department of Computer Science

University of MD, College Park

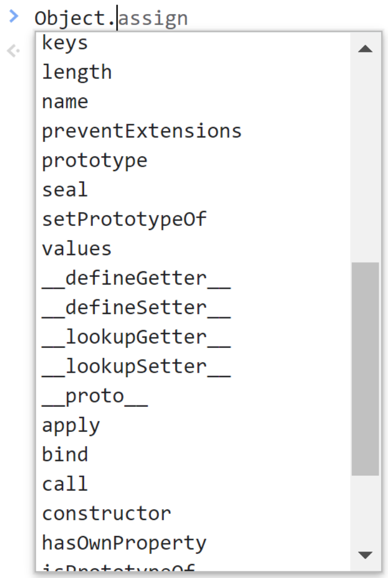
Slides material developed by Ilchul Yoon, Nelson Padua-Perez

Object Type

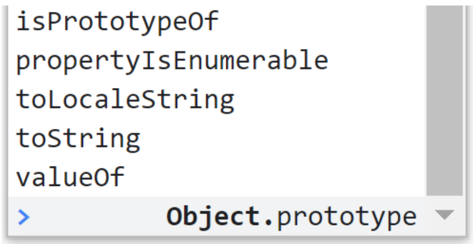
- In JavaScript, **functions are objects**
- There are several **constructor** functions (e.g., Object, Array, Function, Boolean)
  - Execute **typeof Array** on the console
- **Object - constructor function** (an object itself) that supports object creation. If you enter type **Object** in the console you will get 'function'; if you enter **Object** you will see:
  - ƒ Object() { [native code] }
- As an object, **Object** (even though it is a function) can have properties
- You can see several properties of **Object** by typing (in the console) **Object** followed by a period and waiting



```
> Object.assign
assign
create
defineProperties
defineProperty
entries
freeze
fromEntries
getOwnPropertyDescriptor
getOwnPropertyDescriptors
getOwnPropertyNames
getOwnPropertySymbols
getPrototypeOf
hasOwn
is
isExtensible
isFrozen
isSealed
```



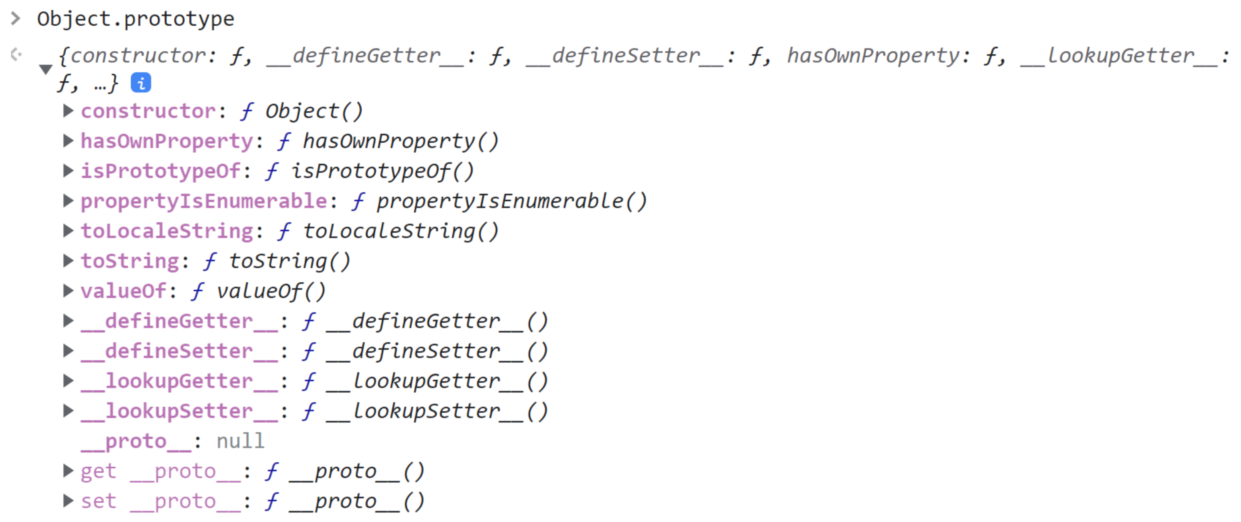
```
> Object.assign
keys
length
name
preventExtensions
prototype
seal
setPrototypeOf
values
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
__proto__
apply
bind
call
constructor
hasOwnProperty
```



```
isPrototypeOf
propertyIsEnumerable
toLocaleString
toString
valueOf
> Object.prototype
```

Object Type

- **Object** has a property called **prototype** that refers to an object
- Let’s type on the console **Object.prototype** and press enter. Expand the right triangle to see its contents. You will see something similar to

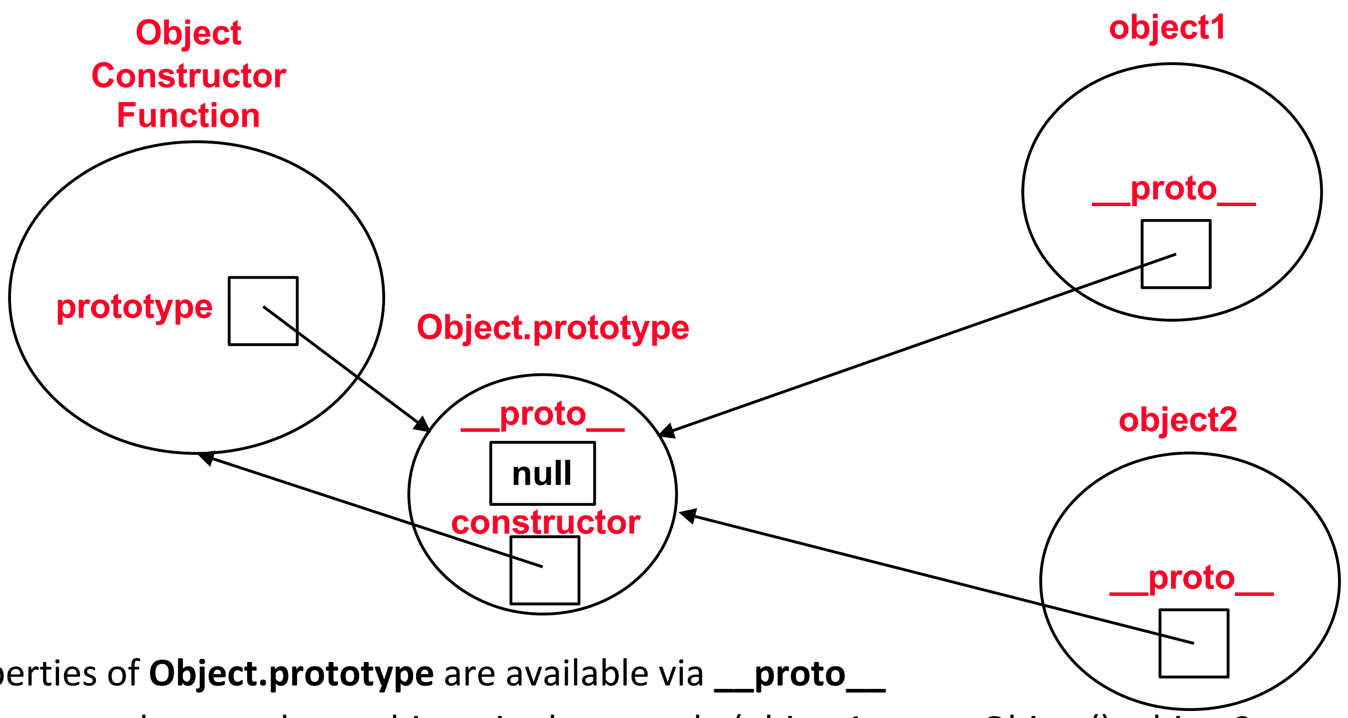


```
> Object.prototype
<· ▼{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __lookupGetter__:
    ƒ, …} i
    ▶constructor: ƒ Object()
    ▶hasOwnProperty: ƒ hasOwnProperty()
    ▶isPrototypeOf: ƒ isPrototypeOf()
    ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶toLocaleString: ƒ toLocaleString()
    ▶toString: ƒ toString()
    ▶valueOf: ƒ valueOf()
    ▶__defineGetter__: ƒ __defineGetter__()
    ▶__defineSetter__: ƒ __defineSetter__()
    ▶__lookupGetter__: ƒ __lookupGetter__()
    ▶__lookupSetter__: ƒ __lookupSetter__()
     __proto__: null
    ▶get __proto__: ƒ __proto__()
    ▶set __proto__: ƒ __proto__()
```

- Type **Object.prototype.constructor === Object**

Creating an object

- When we create an object (new Object(), { }) the following steps take place:
  - New object is created
  - The new object has a property called **__proto__** that now points to the object associated with the **Object.prototype** property

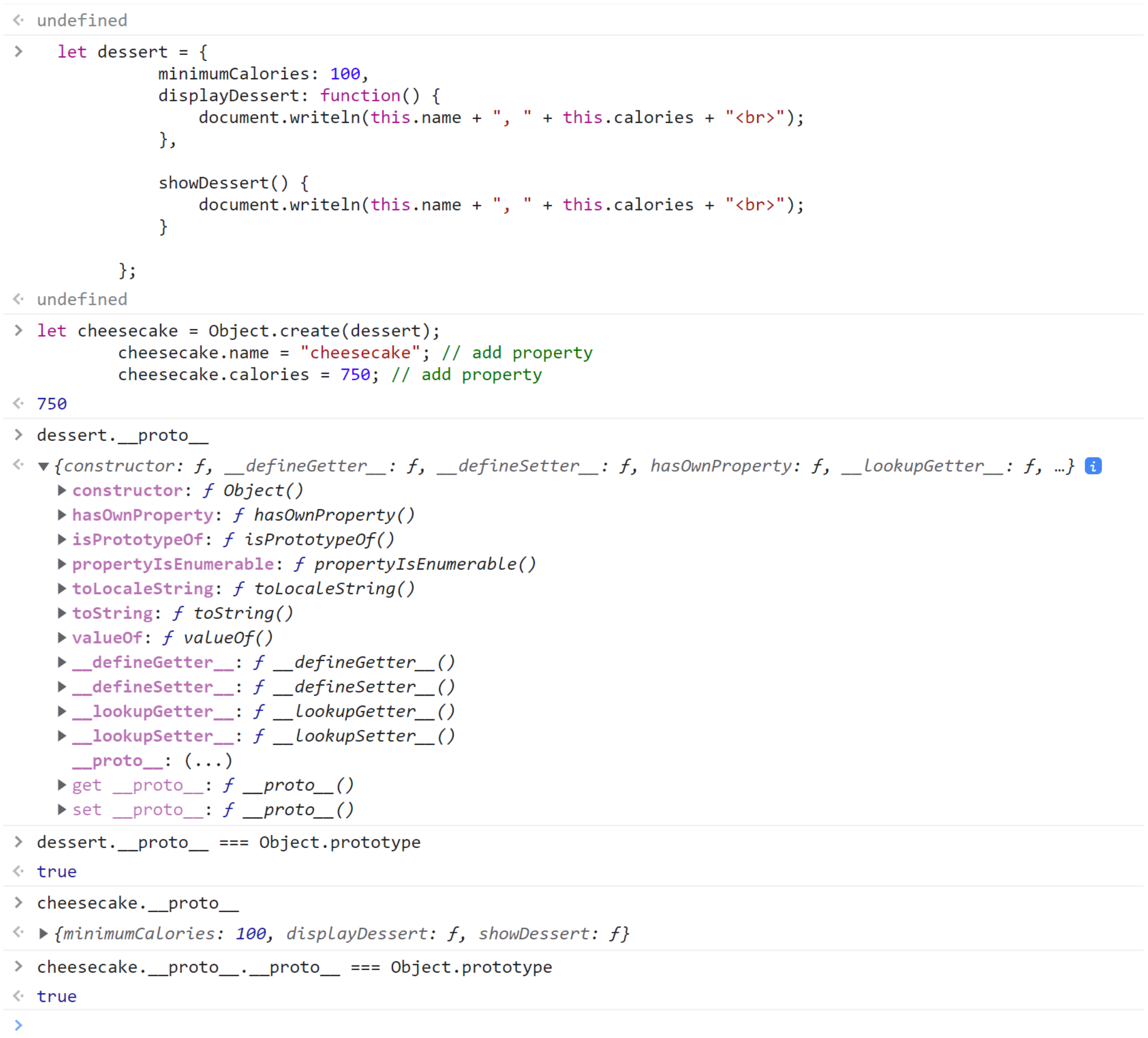


- Properties of **Object.prototype** are available via **__proto__**
- Let's create the two above objects in the console (object1 = new Object(), object2 = new Object())
- Note: We may have not set __proto__ to **null** for Object.prototype in previous exam solutions, but we now expect it
- **Example:** ObjectCreation1.html

Prototype chain

- **Prototype chain**
  - Set of objects defined by the **__proto__** property
  - The end of the chain is a prototype with the null value (Object.prototype.__proto__)
- The **Object.create()** method allow us to provide a prototype object to a newly created object
- **Example:** ObjectCreation2.html
  - Next slide shows relationship between objects

Prototype chain



```
<· undefined
>   let dessert = {
            minimumCalories: 100,
            displayDessert: function() {
                document.writeln(this.name + ", " + this.calories + "<br>");
            },

            showDessert() {
                document.writeln(this.name + ", " + this.calories + "<br>");
            }

        };
<· undefined
> let cheesecake = Object.create(dessert);
        cheesecake.name = "cheesecake"; // add property
        cheesecake.calories = 750; // add property
<· 750
> dessert.__proto__
<· ▼{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ, hasOwnProperty: ƒ, __lookupGetter__: ƒ, …}
    ▶constructor: ƒ Object()
    ▶hasOwnProperty: ƒ hasOwnProperty()
    ▶isPrototypeOf: ƒ isPrototypeOf()
    ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶toLocaleString: ƒ toLocaleString()
    ▶toString: ƒ toString()
    ▶valueOf: ƒ valueOf()
    ▶__defineGetter__: ƒ __defineGetter__()
    ▶__defineSetter__: ƒ __defineSetter__()
    ▶__lookupGetter__: ƒ __lookupGetter__()
    ▶__lookupSetter__: ƒ __lookupSetter__()
     __proto__: (...)
    ▶get __proto__: ƒ __proto__()
    ▶set __proto__: ƒ __proto__()
> dessert.__proto__ === Object.prototype
<· true
> cheesecake.__proto__
<· ▶{minimumCalories: 100, displayDessert: ƒ, showDessert: ƒ}
> cheesecake.__proto__.__proto__ === Object.prototype
<· true
>
```

Function Properties and Methods

- In JavaScript, **every function is a Function object**
- The Function constructor supports the creation of a new Function object
- **length** property
  - Number of parameters expected by a function
- Inside of a function, two objects exist
  - **arguments**
    - Has all the arguments passed into the function
    - **It is not an array**
  - **this**
    - Reference to the **context object** the function is operating on
    - Allows associating functions to an object at runtime
    - You can set **this** using apply(), call(), or bind()
- **Example:** FuncLength.html, FuncArguments.html,
- **Example:** FuncThis.html, FuncApplyCallBind.html

Creating custom objects

- **To create a custom object, you can:**
  - Create a function referred to as the **constructor function**
    - » Convention is to use an uppercase initial letter for the function's name
  - Instantiate and initialize an object using **new** and the constructor function
  - **Any function called with the new operator behaves as a constructor**; without it, the function behaves as a normal function
- **Example:** ConstructorFunction.html

  In the example, code object creation is not efficient as we duplicate the code for the functions (we will see a better alternative later on)

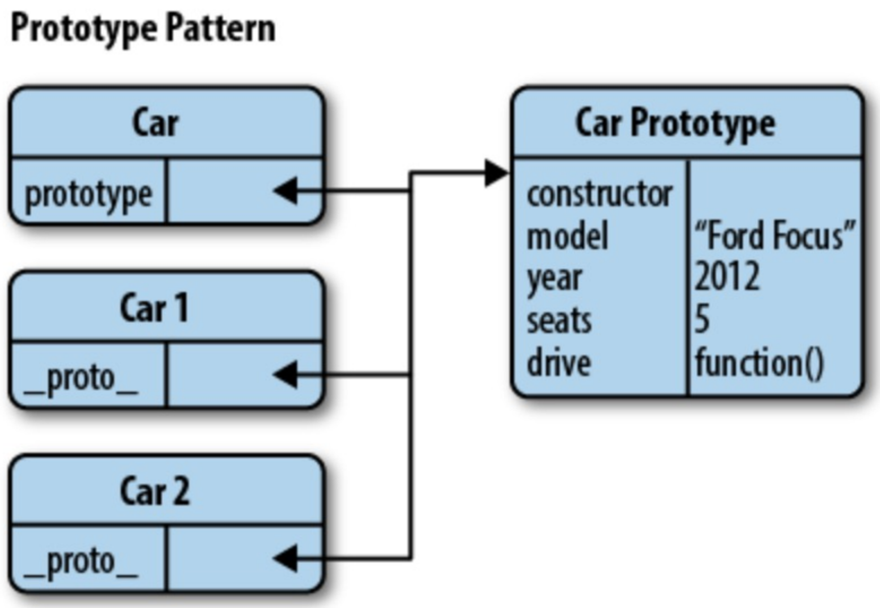
Custom Type Definition

- Before using “class” to implement an abstraction, different approaches were developed to address the creation of objects associated with a particular abstraction:
  - **Constructor Pattern**
  - **Prototype Pattern**
  - **Constructor/Prototype Pattern**
- **Constructor Pattern**
  - Using **constructor** functions
  - Disadvantage: duplication. For example, a function can be duplicated in each object
- **Example:** ConstructorPattern.html
  - Each object has its own copy of the info function

Sharing of prototype (reviewing)

- How **sharing of prototype** takes place when a **constructor** function is used to create an object using **new**:
- Steps

1. JavaScript creates a new empty object and calls the function with **this** referring to the new object
2. The **__proto__** property of the new object is initialized to point to the object referred to by the **prototype** property of the constructor function
3. The new object is returned



Source: Learning JavaScript Design Patterns

Prototype Pattern

- Remember, the **constructor function** has a property called prototype
- The **Constructor** pattern for custom type definition has some disadvantages
  - **Each instance has its own copy of the methods**
- The **Prototype** pattern addresses this problem
- **Example:** PrototypePattern.html
  - Sharing is a problem for certain properties using the Prototype Pattern

Default Pattern for Custom Types

- The default pattern for custom type definition (“class definition”) combines the constructor and prototype pattern
  - **Constructor pattern defines instance variables**
  - **Prototype pattern defines common methods and properties**
- **Example:** DefaultPattern.html
  - Even if instances for an object have been created, adding a property/method to the prototype will make it immediately available

Inheritance

- **Prototype chaining: a primary method for inheritance**
- We can assign a particular object to the prototype property
- **Example:** Inheritance.html

Example: inheritance.html