Object.create(proto)

- Creates a new object with the specified prototype object and properties
- proto parameter must be null or an object
  - Throws TypeError otherwise

Object.create Argument

- Can add/specify initialization parameters directly in Object.create as an (optional) 2nd argument

```
var e = Object.create( Person, { title: {value: "Manager" }} )
```

Prototype Inheritance with *Object.create*: Example

```
var Person = {
    firstName:  "John";
    lastName: "Smith";
    gender: "Male";
    print : function() {
        console.log( "Person : " + this.firstName
                     + this.lastName + this.gender;
    }
};
var e = Object.create( Person );
e.title = "Manager";
```

Design Tips

- Object.create might be cleaner in some situations, rather than using new and .prototype (no need for artificial objects)
- With new, you need to remember to use this and also NOT return an object in the constructor
  - Otherwise, bad things can happen
- Object.create allows you to create objects without running their constructor functions
  - Need to run your constructor manually if you want
  - i.e., Person.call(p2, “Bob”)

Class Activity

- Construct a class hierarchy with the following properties:
- Add an area method and a toString prototype function to all the objects.

Point { x, y } ⇒ Circle { x, y ,r } ⇒ Ellipse { x, y, r, r2 }

Class Activity

Start with:

```
// Base Point constructor function
function Point(x, y) {
  this.x = x;
  this.y = y;
}

// Adding toString method to Point prototype
Point.prototype.toString = function() {
  return `Point at (${this.x}, ${this.y})`;
};
```

Solution: Circle

```
// Circle constructor inheriting from Point
function Circle(x, y, r) {
  Point.call(this, x, y); // Call the parent constructor
  this.r = r;
}

// Inheriting from Point prototype
Circle.prototype = Object.create(Point.prototype);
Circle.prototype.constructor = Circle;

// Adding area method to Circle prototype
Circle.prototype.area = function() {
  return Math.PI * this.r * this.r;
};

// Adding toString method to Circle prototype
Circle.prototype.toString = function() {
  return `Circle at (${this.x}, ${this.y}) with radius ${
      this.r}`;
};
```

Solution: Eclipse

```
// Ellipse constructor inheriting from Circle
function Ellipse(x, y, r, r2) {
  Circle.call(this, x, y, r); // Call the parent constructor
  this.r2 = r2;
}

// Inheriting from Circle prototype
Ellipse.prototype = Object.create(Circle.prototype);
Ellipse.prototype.constructor = Ellipse;

// Adding area method to Ellipse prototype
Ellipse.prototype.area = function() {
  return Math.PI * this.r * this.r2;
};

// Adding toString method to Ellipse prototype
Ellipse.prototype.toString = function() {
  return `Ellipse at (${this.x}, ${this.y}) with radii ${
      this.r} and ${this.r2}`;
};
```

Solution: Usage

```
// Usage
var p = new Point(1, 2);
var c = new Circle(1, 2, 3);
var e = new Ellipse(1, 2, 3, 4);

console.log(p.toString()); // Point at (1, 2)
console.log(c.toString()); // Circle at (1, 2) with radius 3
console.log(e.toString()); // Ellipse at (1, 2) with radii 3
     and 4

console.log(c.area()); // 28.274333882308138
console.log(e.area()); // 37.69911184307752
```

Type-Checking and Reflection

Reflection and Type-Checking

- In JS, you can query an object for its type, prototype, and properties at runtime
  - To get the Prototype: getPrototypeOf()
  - To get the type of: typeof
    - "undefined", "boolean", "number", "string", "symbol", "object", "function"
  - To check if it's of certain instance: instanceof
  - To check if it has a certain property: in
  - To check if it has a property, and the property was not inherited through the prototype chain: hasOwnProperty()

typeof

- Can be used for both primitive types and objects

```
typeof( Person.firstName ) => String
typeof( Person.lastName ) => String
typeof( Person.age ) => Number
typeof(Person.constructor) => function (prototype)
typeof(Person.toString) => function (from Object)
typeof(Person.middleName) => undefined
typeof(Person) => object
typeof(null) => object (bug in js!!!)
```

instanceof

- Checks if an object has in its prototype chain the prototype property of the constructor

```
object instanceof constructor => Boolean

// Example:
var p = new Person( /* ... */ );
var e = new Employee( /* ... */ );

p instanceof Person;     // True
p instanceof Employee;     // False
e instanceof Person;   // True
e instanceof Employee;     // True
p instanceof Object;     // True
e instanceof Object;     // True
```

When to use which?

- Use **typeof** when you need to know the **type of a primitive**.
- Use **instanceof** when you need to confirm the **prototype-based inheritance.**

getPrototypeOf

- Gets an object's prototype (From the prototype field) – Object.getPrototypeOf(Obj)
    - Equivalent of 'super' in languages like Java

in operator

- Tests if an object o has property p
  - Checks both object and its prototype chain

```
var p = new Person( /* ... */ );
var e = new Employee( /* ... */ );

"firstName" in p;  // True
"lastName" in e;   // True
"Title" in p;   // False
"Title" in e;   // True
```

hasOwnProperty

- Only checks the object's properties itself
  - Does not follow the prototype chain
  - Useful to know if an object has overridden a property or introduced a new one

```
var p = new Employee( /* ... */ );
p.hasOwnProperty("Title")  // True
p.hasOwnProperty("FirstName")  // False
```

Iterating over an Object's fields

- Go over the fields of an object and perform some action(s) on them (e.g., print them)
  - Can use hasOwnProperty as a filter if needed

```
var name;
for (name in obj) {
    if ( typeof( obj[name] ) != "function") {
        document.writeln(name + " : " + obj[name]);
    }
}
```

Removing an Object's Property

- To remove a property from an object if it has one (not removed from its prototype), use:

```
delete object.property-name
```

- Properties inherited from the prototype cannot be deleted unless the object had overriden them.

```
var e = new Employee( /* ... */ );
delete e.Title;     // Title is removed from e
```

Object Property Types

- Properties of an object can be configured to have the following attributes (or not):
  - Enumerable: Show up during enumeration(for.. in)
  - Configurable: Can be removed using delete, and the attributes can be changed after creation
  - Writeable: Can be modified after creation
- By default, all properties of an object are enumerable, configurable and writeable

Specifying Object Property types

- Can be done during Object creation with Object.create

```
var Person = { ... };

var jane = Object.create(Person, {
  title: {
    value: "Manager",
    enumerable: true,
    configurable: true,
    writable: false
  }
});
```

- Can be done after creation using Object.defineProperty

Design Guidelines

- Use for...in loops to iterate over object's properties to make the code extensible
  - Avoid hardcoding property names if possible
  - Use instanceof rather than getPrototypeOf
- Try to fix the attributes of a property at object creation time. With very few exceptions, there is no need to change a property's attribute.

Class Activity

- Write a function to iterate over the properties of a given object, and identify those properties that it **inherited** from its prototype AND **overrode** it with its own values
    - Do not consider functions

Solution

```
function findOverriddenProperties(obj) {
  var overridden = [];
  var currentProto = Object.getPrototypeOf(obj);

  while(currentProto && currentProto !== Object.prototype) {
    for(var prop in currentProto) {
      if (!currentProto.hasOwnProperty(prop)) continue;
      // Check if it's not a function, the property exists
         on obj,
      // and its value is different from that on the
         prototype
      if(typeof currentProto[prop] !== 'function' &&
          obj.hasOwnProperty(prop) && obj[prop] !==
            currentProto[prop]) {
        overridden.push(prop);
      }
    }
    currentProto = Object.getPrototypeOf(currentProto);
  }

  return overridden;
}
```

ES6 Classes

With the introduction of ES6 classes, the syntax for creating prototypes becomes much cleaner, although under the hood, it's still using the same prototype-based inheritance:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log('Hello, my name is ' + this.name);
  }
}
const bob = new Person('Bob');
bob.greet(); // Hello, my name is Bob
```

ES6 Extends

```
class Person {
  // Person methods and properties
}
class Employee extends Person {
  // Employee methods and properties
}
// Create an instance of Employee
const jane = new Employee();
```