Here is the spec talk definition of the default `@@new`

1. Let F be the this value.
2. If Type(F) is not Object, then throw a TypeError exception.
3. Let obj be OrdinaryCreateFromConstructor(F, "%ObjectPrototype").
4. ReturnIfAbrupt(obj).
5. Assert: Type(obj) is Object.
6. Let result be the result of calling the [[Call]] internal method of F, providing obj and argumentsList as the arguments.
7. ReturnIfAbrupt(result).
8. If Type(result) is Object then return result.
// a subclass whose constructor hows both inherited and local initialization
class Bar extends Foo {
    constructor() {
        super();
        this.y = 1; // new Bar creates an object with 'x' and 'y' properties
    }
}

// a subclass that uses its constructor as both a factory function and an initializer.
// Things start getting messy
class Baz extends Bar {
    constructor() {
        if (this === undefined) { // test probably inadequate
            // called as a factory function
            return new Baz;
        }
        // called as an initializing constructor
        super();
        this.z = 2; // new Baz creates an object with 'x' 'y' 'z' properties
    }
}

// alternatively, we might simplify the constructor by making it exclusively a factory function
// and moving initialization to a @@new method. but this is also messy
class Baz2 extends Bar {
    constructor() {
        if (this === undefined) return new Baz2; // Baz2() call case
    }
    static [Symbol.new]() {
        let obj = super(); // note will call our constructor but it does nothing because this != undefined
        // so we needs to explicitly call Bar constructor
        Bar.call(obj);
        obj.z = 2;
        return obj; // new Baz2 creates an object with 'x' 'y' 'z' properties
    }
}

// Baz Alternative 1 is easy to further subclass, using normal constructor initialization
class BazSub extends Baz {
    constructor() {
        super(); // won't trip the Baz factory test
        this.q = 3; // new BazSub creates an object with 'x' 'y' 'z' 'q' properties
    }
}

// Baz Alternative 2 is harder to subclass correctly
// There are a couple alternative approaches
// the first sub alternative restores constructor initialization behavior
class BazSub2a extends Baz2 {
    constructor() {
        // it would be wrong to super() here, because it would invoke Baz2 factory
        this.q = 3; // new BazSub2a creates object with 'x' 'y' 'z' 'q' properties, assuming @@new is over-ridden as follow
    }
    static [Symbol.new]() {
        let obj = super(); // allocates object and calls Bar constructor on it
        // and restore default constructor initialization call behavior
        let result = this.apply(obj, args);
        if ($Type(result) !== "Object") return obj;
        return result; // object from constructor with 'x' 'y' 'z' 'q' properties
    }
}

// the second sub alternative continues to do subclass initialization into the subclass @@new
class BazSub2b extends Baz2 {

    constructor() {
        // It would be wrong to super() here, because it would invoke Baz2 factory
        throw new TypeError("invalid subclass, extend via @@new method"); // just in case somebody super() calls us
    }

    static [Symbol.new]() {
        let obj = super();
        obj.q = 3;
        return obj; // new BazSub2b creates an object with 'x' 'y' 'z' 'q' properties
    }

    // In general, trying to use constructors as factory functions seriously complicates subclassing and probably should
    // be avoided in new class definitions. But most legacy ES built-ins have constructors with factory function behavior
    // and these sorts of issues have to be considered to make them usefully subclassable.
    // At least for the above examples, using @@new seems to complicate creating subclasses when constructor factory functions
    // are involved.
}

//class that have exotic objects as instances

// using @@new to allocated exotic instances
class P {
    constructor() {
        this.x = 1;
    }

    static [Symbol.new](...args) {
        // create copied from Function.prototype[Symbol.new]
        let proto = this.prototype;
        if (typeof proto !== "Object") proto = Object.prototype;
        let obj = Object.create(proto); // the target for the proxy
        let result = this.apply(result, args);
        obj = Proxy(obj, handler); // create a Proxy for the target
        let result = this.apply(obj, args); // call constructor with the Proxy as the this value
        if (typeof(result) !== "Object") return obj ;
        return result;
    }
}

// alternative 1, using @@create as currently in ES6
class P1 {
    constructor() {
        this.x = 1;
    }

    static [Symbol.create](){
        let obj = super[Symbol.create]();
        return Proxy(obj, handler);
    }
}

// Defining Object as a class using @@new

class Object extends null {
    constructor(...args) {
        return new Object(...args);
    }

    static [Symbol.new](value) {
        if (value === null) return new super;
        if (typeof value === "boolean") return new Boolean(value);
    }
}
if (typeof value == "number") return new Number(value);
if (typeof value == "string") return new String(value);
if (typeof value == "symbol") return $ToObject(value);
return value;
}

6-Object.js

//handwired Object using @@new

function Object(...args) {
  return new Object(...args)
}

Object.prototype = Object.create(null);
Object.prototype.constructor = Object;
Object.prototype[Symbol.new] = function(value) {
  if (value == null) return new super;
  if (typeof value == "boolean") return new Boolean(value);
  if (typeof value == "number") return new Number(value);
  if (typeof value == "string") return new String(value);
  if (typeof value == "symbol") return $ToObject(value);
  return value;
}

7-Number-class.js

class Number {
  constructor (...args) {
    if (args.length === 0) return +0;
    return +args[0];
  }
  static [Symbol.new] (...args) {
    let n = args.length === 0 ? +0 : +args[0];
    return $CreateWithSlots(this,"[[NumberData]]", [n])
  }
}

Set.js

//exploring defining Set, with and without @@new
//A objet with a private slot needs to be allowed as the Set instance
//Note that is both alternatives, the private slot is initailized before
//the instance is exposed to constructor code.

class Set {
  constructor (iterable) {
    if (iterable != null)
      for (let e of iterable) this.add(e);
  }
  static [Symbol.new] (...args) {
    let obj = $CreateWithSlots(this,"[[SetData]]", [undefined]);
    $SetNewSetData(obj);
    //initialize, copied from default @@new
    let result = this.apply(result, args);
    if ($Type(result) !== "Object") return obj ;
    return result;
  }
}

//or using @@create

class Set {
  constructor (iterable) {
    if (iterable != null)
      for (let e of iterable) this.add(e);
```javascript
static [Symbol.create]( ) {
    let obj = $CreateWithSlots(this,[[SetData]], [undefined]);
    $setNewSetData(obj);
    return obj;
}
```