ECMAScript 6
& TypeScript

-- *PART 2* --

ECMAScript 6 & TypeScript

- Enums
- Modules
- Types
- Classes
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Enums

- A way to organize a collection of related values
- Enum members have *numeric* values associated with them and can be either constant or computed
- TS Only. JS does not provide enums
- Enums are number based
- Syntax:

```
enum EnumName {
    elem1 [= initializer1],
    elem2 [= initializer2],
    …
    elemN [= initializerN]
}
```

Example

```
enum Characters {
    WalterWhite,
    SkylerWhite,
    SaulGoodman,
    JessePinkman,
    GusFring
}

var main = Characters.WalterWhite;

console.log(main);  // 0
console.log(main === Characters.WalterWhite);  // true
```

Enum Values

- Enum members have numeric values associated with them
- Generally:
  - First element receives a default value of 0
  - Other elements receive previous element's value + 1
  - Multiple elements can have same value
- However values can also be computed:
  - Expressions evaluating to a number
  - Expressions using previous members
  - Function calls
- Some computed values are known & defined at compile time, others only at runtime (e.g. function calls)

Example

```
function getGusValue() {
    return 99;
}

enum Characters {
    WalterWhite = 1,
    SkylerWhite = WalterWhite, // also 1
    SaulGoodman, // = 2  (prev + 1)
    JessePinkman = 10 * SaulGoodman,
    GusFring = getGusValue()
}

console.dir(Characters);
```

```
Object
    1:"SkylerWhite"
    2:"SaulGoodman"
    20:"JessePinkman"
    99:"GusFring"
    GusFring:99
    JessePinkman:20
    SaulGoodman:2
    SkylerWhite:1
    WalterWhite:1
```

Which Transpiles to...

```
function getGusValue() {
    return 99;
}

var Characters;

(function (Characters) {
    Characters[Characters["WalterWhite"] = 1] = "WalterWhite";
    Characters[Characters["SkylerWhite"] = 1] = "SkylerWhite";
    Characters[Characters["SaulGoodman"] = 2] = "SaulGoodman";
    Characters[Characters["JessePinkman"] = 20] = "JessePinkman";
    Characters[Characters["GusFring"] = getGusValue()] = "GusFring";
})(Characters || (Characters = {}));

console.dir(Characters);
```

Const Enums

- TS generated an object with both forward (name -> value) and reverse (value -> name) mappings, as we've seen earlier
- References to enum members are always emitted as property accesses, example:
  - console.log(*Characters*.JessePinkman);
- For a performance boost we can create *const* enums
- Const enum references use inline values
  - But then we can't use computed members ☹

Example

```
const enum Characters {
    WalterWhite = 1,
    SkylerWhite = WalterWhite,
    SaulGoodman,
    JessePinkman = 10 * SaulGoodman,
    // sorry, no computed values allowed  ☹ */
    GusFring = 99 /* getGusValue() */
}

console.log(Characters.JessePinkman);
```

Which Transpiles to…

```
// no enum object created
// values are inlined

console.log(20 /* JessePinkman */);
```

Browser Compatibility

- This is a TypeScript specific feature not supported natively by JS

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Classes
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Modules

- Before ES6, JS did not have modules, and so libraries were used instead. Now, ES6 finally introduced modules.
- Modules are executed within their own scope: declarations do not pollute the global namespace
- Modules are stored in files: one module per file
- Module name is the file name (w/o extension)
- The *export* and *import* statements are used to import/export module declarations respectively
- Two export types exist: named and default
  - Named exports are useful to export several values
  - Default exports are considered the “main” exported module value. Limited to single default per module.

Example – Named Exports

```
/* calculator.js */

const COEFFICIENT = 42;

export function calculate(x, y) {
    return x + COEFFICIENT * y;
}

export { COEFFICIENT };
```

```
/* application.js */

import { calculate, COEFFICIENT } from "./calculator";

console.log(calculate(10, 20)); // 42
console.log(COEFFICIENT); // 850
```

Example – Default Exports

```
/* calculator.js */

const COEFFICIENT = 42;

export default function calculate(x, y) {
    return x + COEFFICIENT * y;
}
```

```
/* application.js */

import calculate from "./calculator";   // no curly braces around calculate

console.log(calculate(10, 20)); // 850
```

A Word about Module Loaders

- As we’ve seen, modules can import/use one another
- The actual module files loading is performed by a *module loader*, responsible for:
  - Locating the module files
  - Fetching/loading them into memory
  - Handling module dependencies
  - Executing their code
- This is usually done in runtime (although can be done in compile time e.g. for dist bundling)
- Common module loaders include *requirejs* and *systemjs*

TS & Modules

- TS needs to know which module loader we will be using, as the compilation output differs for each one
- We define it using the compiler *module* option:

```
// tsconfig.json

{
    "compilerOptions": {
        "target": “es6",
        "module": "commonjs“ // other options: amd, system, es6, umd
    }
}
```

- We will now see how TS transpiles modules to be used for commonjs

Modules & TS – Named Exports

```
/* calculator.js */

var COEFFICIENT = 42;
exports.COEFFICIENT = COEFFICIENT;

function calculate(x, y) {
    return x + COEFFICIENT * y;
}
exports.calculate = calculate;
```

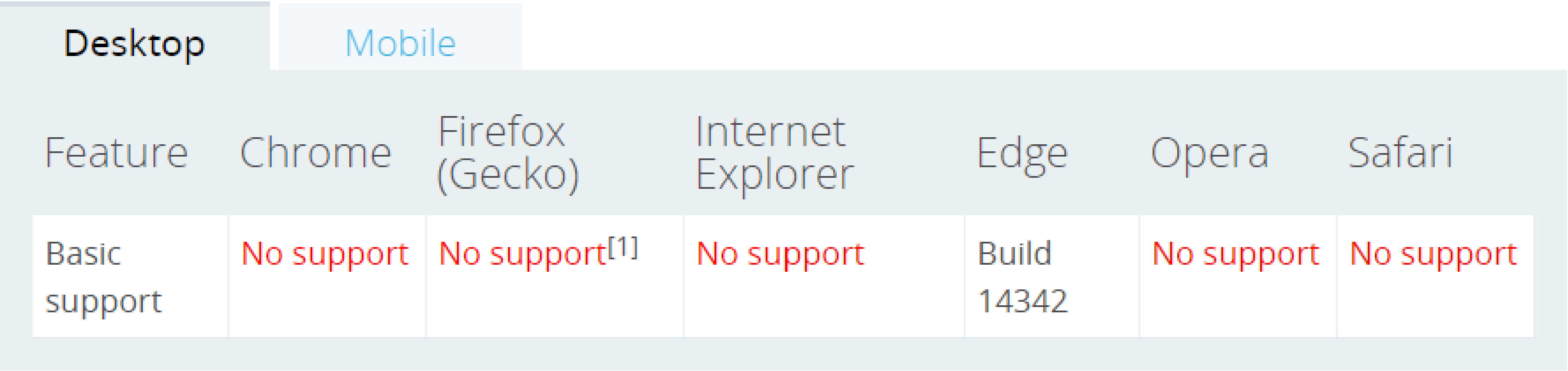
```
/* application.js */

var calculator_1 = require("./calculator");

console.log(calculator_1.calculate(10, 20));
console.log(calculator_1.COEFFICIENT);
```

Modules & TS – Default Exports

```
/* calculator.js */

var COEFFICIENT = 42;

function calculate(x, y) {
    return x + COEFFICIENT * y;
}

// module mode marker for interoperability
Object.defineProperty(exports, "__esModule", { value: true });
exports.default = calculate; // actual exported default value
```

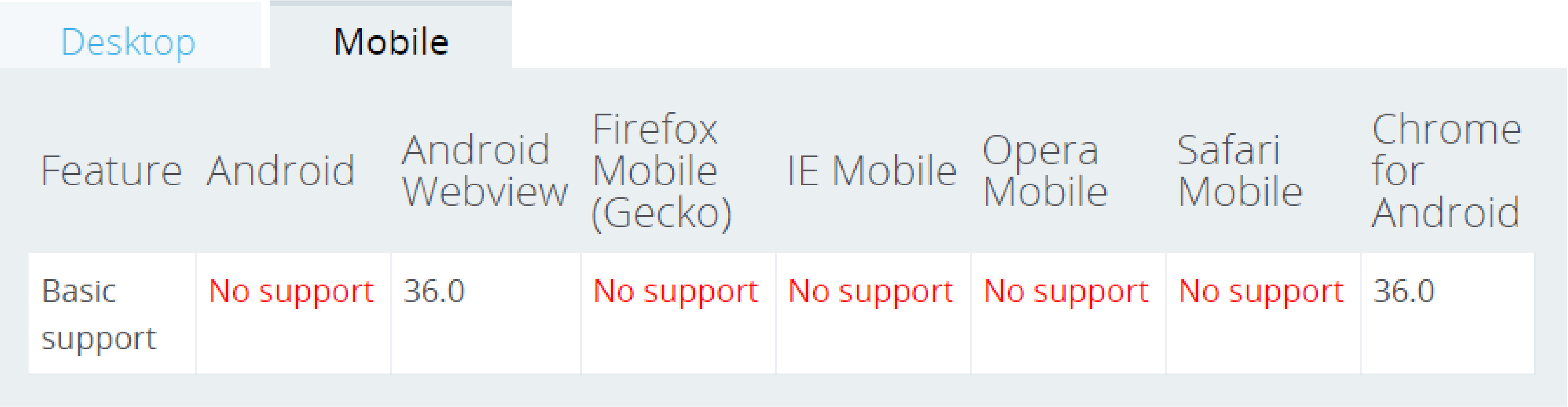
```
/* application.js */

var calculator_1 = require("./calculator");
console.log(calculator_1.default(10, 20)); // imported as “default”
```

Browser Compatibility - import



Desktop Mobile

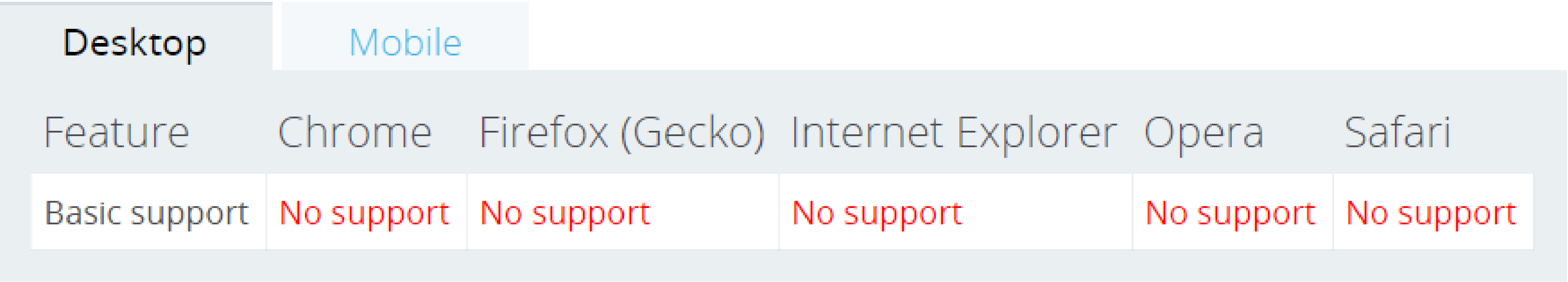
Feature	Chrome	Firefox (Gecko)	Internet Explorer	Edge	Opera	Safari
Basic support	No support	No support[1]	No support	Build 14342	No support	No support



Desktop Mobile

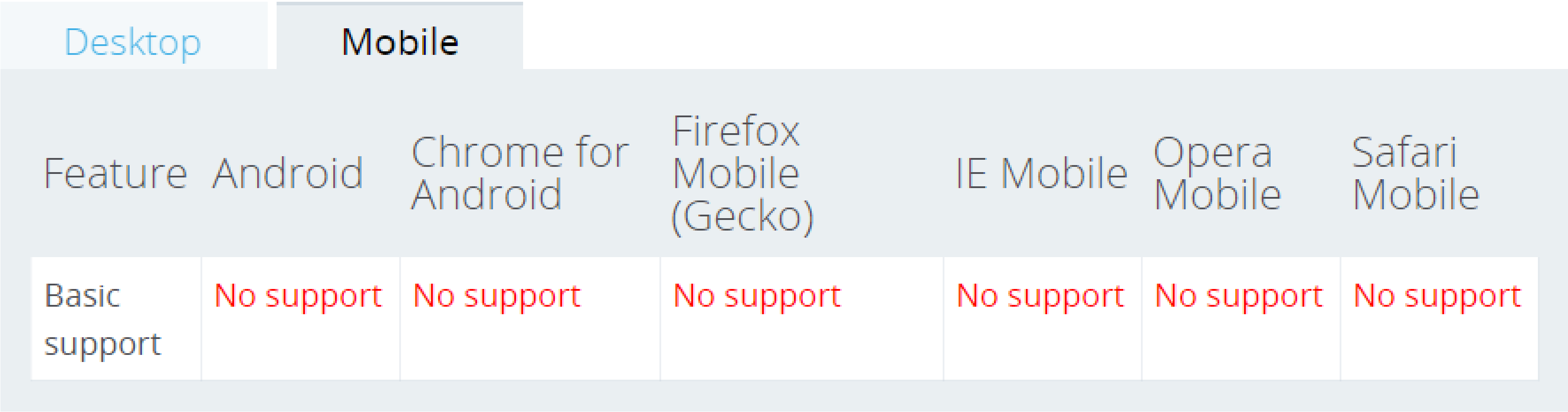
Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	36.0	No support	No support	No support	No support	36.0

Browser Compatibility - export



Desktop Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	No support	No support	No support	No support	No support



Desktop Mobile

Feature	Android	Chrome for Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	No support	No support	No support	No support	No support	No support

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Classes
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Types

- One of the main goals/reasons for using TypeScript
- Types enhance code quality and understandability, and allow us to catch errors at compile time
- JS natively has six (6) primitive data types:
  - string
  - number
  - boolean
  - null
  - undefined
  - symbol ← new in ES6
- TS uses these (and more e.g. interfaces, classes, arrays, …) for its typing system

Implicit Typing

- TS does not force using types; they are intentionally optional
- Any plain JS can be renamed .js -> .ts an will still compile
- TS attempts to infer types from the code and provide compile time type safety

```
var mynumber = 42; // TS implicitly infers type number

mynumber = "great number“; // assignment of a string value into a number typed var

// will compile but raise a compile time error
// Error: TS2322: Type 'string' is not assignable to type 'number'.
```

- However, using *explicit* types (next slide) greatly improve TS's ability to warn us of potential errors/bugs

Explicit Typing

- The basic *type annotations* are as follows:

```
var base: number = 123;

function multiply (num: number): number {
    return base * num;
}
```

- Anything that is available in the type *declaration space* can be used as a type annotation
- The declaration space includes JS primitive types, interfaces, enums, functions, classes, arrays

Primitives & Arrays

```
var num: number;
var str: string;
var bool: boolean;

num = 123;
num = 123.456;
num = '123'; // Error

str = '123';
str = 123; // Error

bool = true;
bool = false;
bool = 'false'; // Error
```

```
var boolArray: boolean[];

boolArray = [true, false];
console.log(boolArray[0]); // true
console.log(boolArray.length); // 2
boolArray[1] = true;
boolArray = [false, false];

boolArray[0] = 'false'; // Error!
boolArray = 'false'; // Error!
boolArray = [true, 'false']; // Error!
```

Interfaces

- TS’s primary way for composing multiple type annotations into a single named annotation

```
interface Name {
    first: string;
    second: string;
}

var name: Name;
name = { first: 'John', second: 'Doe’  }; // Okay

name = { first: 'John‘  }; // Error : `second` is missing
name = { first: 'John', second: 1337 }; // Error : `second` is the wrong type
```

Inline Type Annotation

- Instead of creating an interface we can annotate inline

```
var name: {
    first: string;
    second: string;
};

name = { first: 'John', second: 'Doe’  }; // Okay

name = { first: 'John‘  }; // Error : `second` is missing
name = { first: 'John', second: 1337 }; // Error : `second` is the wrong type
```

- Great for quickly providing a one off type annotation
- However, if repeatedly used consider refactoring into an interface (or a *type alias* covered later)

Special Types - any

- Beyond the primitive types there are few types with special meaning in TS: *any, null, undefined, void*
- ***any***:
  - Compatible with all types
  - Tells the compiler not to do any meaningful static analysis

```
var power: any;

// takes any and all types
power = '123';  // number
power = 123;   // string

// compatible with all types
var num: number;
power = num;
num = power;
```

Special Types – null & undefined

- treated the same as something of type *any*
- These literals can be assigned to any other type

```
var num: number;
var str: string;

// these literals can be assigned to anything
num = null;
str = undefined;
```

Special Types – void

- Use :*void* to signify that a function has no return type (and value)

```
function log(message: string): void {
    console.log(message);
}
```

Function Types

- Parameter & Return Type annotations

```
interface Person {
    name: string;
    age: number;
}

function getAge (person: Person): number {
    return person.age;
}
```

- Optional Parameters

```
function addCharacter (name: string, age?: number): void {
    // ..
}

addCharacter('Jon Snow', 24);
addCharacter('Sansa Stark');  // okay, age is optional
```

Function Overloading

- Allows us to define two or more functions with the same name but different signatures

```
class Person {
    constructor(public name:string, public age:number) {}
}

function getAge (x: Person[]): number;
function getAge (x: Person): number {
    if (x instanceof Person) {
        return x.age;
    }
    return group.map(p => p.age ).reduce((a1, a2) => (a1 + a2), 0);
}

var group = [];
group.push(new Person('Jack', 30));
group.push(new Person('Jill', 28));
group.push(new Person('Dave', 15));

console.log(getAge(group[0])); // 30
console.log(getAge(group)); // 73
```

Type Guards

- Allows narrowing down an object type within conditional block
- TS understands the variable type within that conditional block

```
// as seen in previous example

 if (x instanceof Person) {    // TS understands that within this block x is a of type Person
        return x.age;    // and therefore allows us to access the ‘age’ property
 }
```

- We can even create user defined type guards (out of scope)

Generics

- Many algorithms and data structures in computer science do not depend on the *actual type* of the object
- Allows us to define functions, classes and interfaces that are based on *type parameters*

```
// function based on the type parameter T
function reverse<T>(items: T[]): T[] {
    var reversed = [];
    for (let i = items.length - 1; i >= 0; i--) {
        reversed.push(items[i]);
    }
    return reversed ;
}

var numArr = [1, 2, 3];  // implicitly typed as :number[]
var numArrRev = reverse(numArr);  // returns an array of type :number[] , with values = 3,2,1

var strArr = [‘one’, ‘two’];  // implicitly typed as :string[]
var strArrRev = reverse(strArr);  // returns an array of type :string[] , with values = ‘two’, ‘one’
```

trainologic

Generics

- As a matter of fact, JS string’s prototype already has a .reverse() function
- TS itself uses generics to define its structure (in lib.d.ts)
- Meaning we get type safety when calling .reverse() on any array

```
/////////////////////////////
/// ECMAScript Array API (specially handled by compiler)
/////////////////////////////

interface Array<T> {

         /**
      * Reverses the elements in an Array.
      */
    reverse(): T[];
```

Union Type

- Allows a property to be one of multiple types (e.g string or a number)
- Denoted by the pipe sign | in a type annotation (e.g. string|number)

```
// can take a string or array of strings
function formatCommandline(command: string[ ]|string) {
    var line = '';
    if (typeof command === 'string') {
        line = command.trim();
    } else {
        line = command.join(' ').trim();
    }

    // do stuff with line:string …
}
```

Intersection Type

- Allows us to define a type having members of several types

```
function extend<T, U>(first: T, second: U): T & U {
    let result = <T & U> {};
    for (let id in first) {
        result[id] = first[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            result[id] = second[id];
        }
    }
    return result;
}

var x = extend({ a: "hello" }, { b: 42 }); // x now has both `a` and `b`
console.log(x.a, x.b); // hello 42
```

- Commonly used for mixins (which are convenient replacement for multiple inheritance we don’t have in JS)
- Note we’re not limited to two types only (e.g. T & U & V & W)

Tuple Type

- Tuples are finite ordered list of elements
- Syntax ***:[type1, type2, … typeN]***

```
var nameNumber: [string, number];

nameNumber = ['Saul Goodman', 5055034455];  // Okay
nameNumber = ['Saul Goodman', '5055034455']; // Error!

var [name, num] = nameNumber; // destructure
```

Type Alias

- Used for providing names for reusable type annotations
- Syntax ***type someName = anyValidTypeAnnotation***

```
type StrOrNum = string|number;

var sample: StrOrNum; // used like any other notation

sample = 123;  // okay
sample = '123'; // okay
sample = true;  // error
```

- Type aliases can be created for any type really

```
type Text = string | { text: string };  //  union
type Coordinates = [number, number];  // tuple
type Callback = (data: string) => void;  // callback
```

lib.d.ts

- A special declaration file that ships with every TS installation
- Contains the *ambient declarations* (next slide) for common JS constructs (JS runtimes and the DOM)
    - Automatically included in compilation context of TS projects
    - Makes it easy for us to start writing type checked JS code

```
var foo = 123;
var bar = foo.toString(); // okay since lib.d.ts is included

// but if we set compiler flag to noLib: true in tsconfig.json then …

var bar = foo.toString();  // ERROR: Property 'toString' does not exist on type 'number'.
```

Ambient Declarations

- Used to provide type information (definitions) for existing JS code / libraries, either 3rd party or our own
- Contain the type information but not the implementation
- This provides us with type-checking and auto-completion without the need to re-write the code in TS
- Files usually end with a **.d.ts** extension
- There are many ambient declarations already written for us (jquery, angular, moment, …)
- We can use dev tools such as *Typings* for fetching existing .d.ts files
- .d.ts files are actually a great source for documentation and good declaration practices to learn from

JS & Types

- The typing system we covered is TS specific
- JS knows nothing about it
- All types are completely removed when transpiled

```
class Person {
    constructor(public name:string,
                        public age:number) { }
}

function getAge(x: Person[]): number;
function getAge(x: Person): number {
    if (x instanceof Person) {
        return x.age;
    }
    return group.map(p => p.age )
       .reduce((a1, a2) => (a1 + a2), 0);
}
```

```
var Person = (function () {
    function Person(name, age) {
        this.name = name;
        this.age = age;
    }
    return Person;
}());

function getAge(x) {
    if (x instanceof Person) {
        return x.age;
    }
    return group.map(function (p) { return p.age; })
         .reduce(function (a1, a2) { return (a1 + a2); }, 0);
}
```

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Classes
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Classes

- ES5 classes are syntactic sugar over prototypical inheritance
- Classes provide simpler & clearer syntax for dealing with inheritance
- Classes can be defined in similar manner to function expressions and function declarations:

```
// class declaration
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}

var p = new Point(10, 20);
```

```
// class expression
var Point  = class {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}

var p = new Point(10, 20);
```

Classes – Hoisting

- As opposed to function declarations, class declarations are not hoisted
- Thus class declarations cannot be used before the declaration

```
// ReferenceError !
var p = new Point(10, 20);

// class declaration
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
```

```
// Okay
var f = calc(10, 20);

// function declaration
function calc (x, y) {
    return x * y;
}
```

Classes – Body & CTor

- The body class is the part within the curly braces {}
- This is where we define properties and methods
- Body code is executed in strict mode
- One special method is the *constructor*, for creating and initializing a class object instance

```
class Point { // body starts here
    constructor(x, y) {
        this.x = x;
        this.y = y;
        console.log('new point created');
    }
} // body ends here
```

Classes – Prototype Methods

- Methods are defined within the body as follows

```
class Westeros {

    this.kingdoms = [];
    this.maxKingdoms = 7;

    constructor() {
        console.log(“Westeros initialized”);
    }

    addKingdom(name) {
        if (this.kingdoms.length >= 7) {
            console.log(“Sorry, max kingdoms reached”);
            return;
        }
        this.kingdoms.push(name);
    }
}
```

Classes - Sub Classing

- The *extends* keyword is used to create a child class (sub-class)
- A class can only have a single superclass (i.e. single inheritance)
- The *super* keyword is used to access the parent class
  - *super()* invokes the object’s parent constructor
  - *super.someMethod()* invokes *someMethod* on the object's parent

```
class Dothraki {
    constructor(name) {
        this.name = name;
        console.log(
                name + “ created”);
    }
}
```

```
class DothrakiWarrior extends Dothraki{
    constructor(name, weapon) {
        super(name);
        this.weapon= weapon;
        console.log(“Weapon = “ + weapon);
    }
}
```

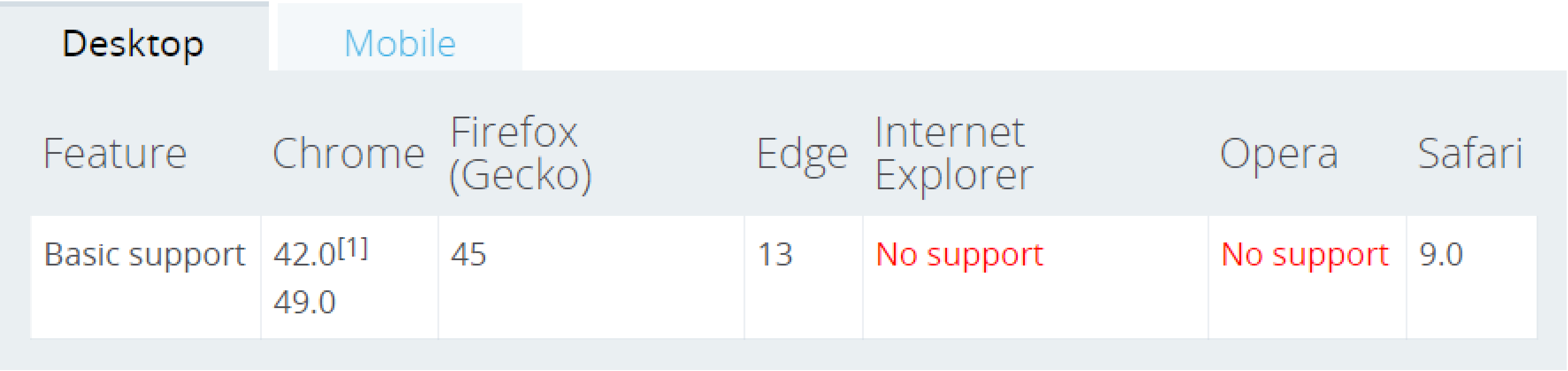
```
var khalDrogo = new DothrakiWarrior(“Khal Drogo”, “Sword”);

// Khal Drogo created \n Weapon = Sword
```

Classes – Static Methods

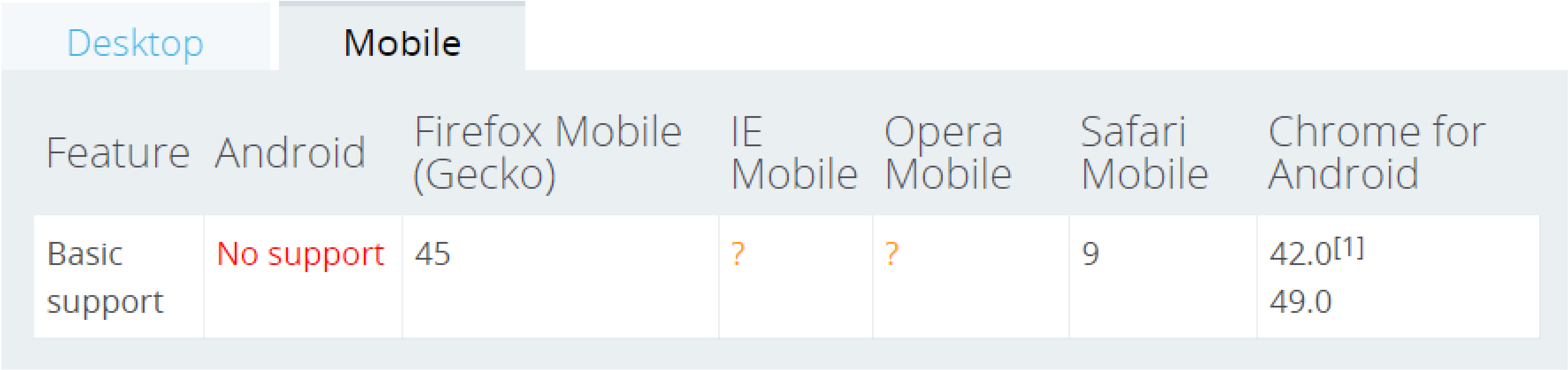
- The *static* keyword defines static methods (shared across all class instances)
- They are called using the class name (not an instance)

```
class Dothraki {

    constructor(name) {
        this.name = name;
        console.log(name + " created");
    }

    static greet() {
        console.log("Hello, kirekosi are yeri?");
    }
}

console.log(Dothraki.greet()); // Hello, kirekosi are yeri?
```

Browser Compatibility - Classes



Feature	Chrome	Firefox (Gecko)	Edge	Internet Explorer	Opera	Safari
Basic support	42.0[1] 49.0	45	13	No support	No support	9.0



Feature	Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	45	?	?	9	42.0[1] 49.0

Classes & TS

- TypeScript’s classes have some additional features which do not exist in ES6:
- **Types**: covered in previous section
- **Properties:** class value members (as opposed to methods)
- **Access Modifiers*** determine accessibility to class members:

Accessible On	public	private	protected
Class instances	yes	no	no
Class	yes	yes	yes
Class children	yes	no	yes

** At runtime these have no significance, but will raise errors in compile time if you incorrectly used.*

Classes – TS - Example

```
class Point {
    x: number;
    y: number;
    static instances: number = 0;

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

    add(point: Point) {
        return new Point(this.x + point.x, this.y + point.y);
    }

    static printNumInstances() {
        console.log("There are " + Point.instances + " points");
    }
}

var p1 = new Point(0, 10);
var p2 = new Point(10, 20);
var p3 = p1.add(p2); // {x:10,y:30}
Point.printNumInstances(); // There are 3 points
```

Classes – TS - Transpiled

```
class Point {
    x: number;
    y: number;
    static instances: number = 0;

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

    add(point: Point) {
        return new Point(
                this.x + point.x, this.y + point.y);
    }

    static printNumInstances() {
        console.log("There are " +
                Point.instances + " points");
    }
}
```

```
var Point = (function () {

    function Point (x, y) {
        Point.instances++;
    }

    Point.prototype.add = function (point) {
        return new Point(
                this.x + point.x, this.y + point.y);
    };

    Point.printNumInstances = function () {
        console.log("There are " +
                Point.instances + " points");
    };

    Point.instances = 0;

    return Point;

}());
```

Classes – Define Using Constructor

- A very common class member initialization is:

```
class Foo {
   x: number;
   constructor(x:number) {
      this.x = x;
   }
}
```

- TS thus provides a convenient shorthand annotation that does the same:

```
class Foo {
   constructor(public x:number) {
   }
}
```

Define Using Constructor – TS - Transpiled

```
class Foo {

    constructor(public x:number) {
    }

}
```

```
var Foo = (function () {

    function Foo(x) {
        this.x = x;
    }

    return Foo;

}());
```

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Types
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Iterators

- Iterators are a Behavioral Design Pattern common for OOP languages
- Used for processing/going over collations, which is a very common task
- *Iterators* bring the iteration concept directly into core JS
- Provide a mechanism for customizing the behavior of *for…of* loops
- Iterators are objects that know how to access collection items one at a time, keeping track of the current item
- An iterator’s *next()* method returns an object with two properties:
  - *done* – boolean indicating whether no more items left
  - *value* – the item value

Example

```
function makeOddIterator (array){
    var nextIndex = 0;

    return { // the iterator
        next: function() {
            var retval = nextIndex < array.length ?  {value: array[nextIndex], done: false} : {done: true};
            nextIndex += 2;
            return retval;
        }
    }
}

var iter = makeOddIterator(['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight']);

for (var item = iter.next(); !item.done; item = iter.next()) {
    console.log(item.value);
}
// one, three, five, seven
```

Iterables

- An object is *iterable* if it defines its iteration behavior
    - Such as which values are looped over in a *for..of* construct
- To be *iterable*, an object must implement the @@iterator method
- Some built-in types, such as Array or Map, have a default iteration behavior (e.g. Array, Map, String), while others (e.g Object) do not
- Some statements and expressions actually expect iterables:

```
for(let value of ["a", "b", "c"]){ // for...of loop
        // ...
}

[..."abc"]; // ["a", "b", "c"] // spread operator

[a, b, c] = new Set(["a", "b", "c"]); // destructuring assignment
```

User Defined Iterable - ES6

```
let iterable = {
    0: 'a',
    1: 'b',
    2: 'c',
    length: 3,
    [Symbol.iterator]() {
        let index = 0;
        return {
            next: () => {
                let value = this[index];
                let done = index >= this.length;
                index++;
                return { value, done };
            }
        };
    }
};
for (let item of iterable) {
    console.log(item); // 'a', 'b', 'c'
}
```

User Defined Iterable - TS

```
class IterableStuffCollection implements IterableIterator<any> {

    private pointer = 0;
    constructor(private stuff:any[]) { }

    public next():IteratorResult<any> {
        let value = this.stuff[this.pointer];
        let done = this.pointer >= this.stuff.length;
        this.pointer++;
        return {value, done};
    }

    [Symbol.iterator]():IterableIterator<any> {
        return this;
    }
}

var myStuff = new IterableStuffCollection(['XBox One', 42, Math.PI, 'pokemon go', {oh: 'yeah'}]);
for (let item of myStuff) {
    console.log(item);
}

// XBox One, 42, 3.141592653589793, pokemon go, { oh: 'yeah' }
```

User Defined Iterable – TS - Notes

- Previous code example require ES6 target

```
// tsconfig.json

"compilerOptions": {
        "target": "ES6“
 }
```

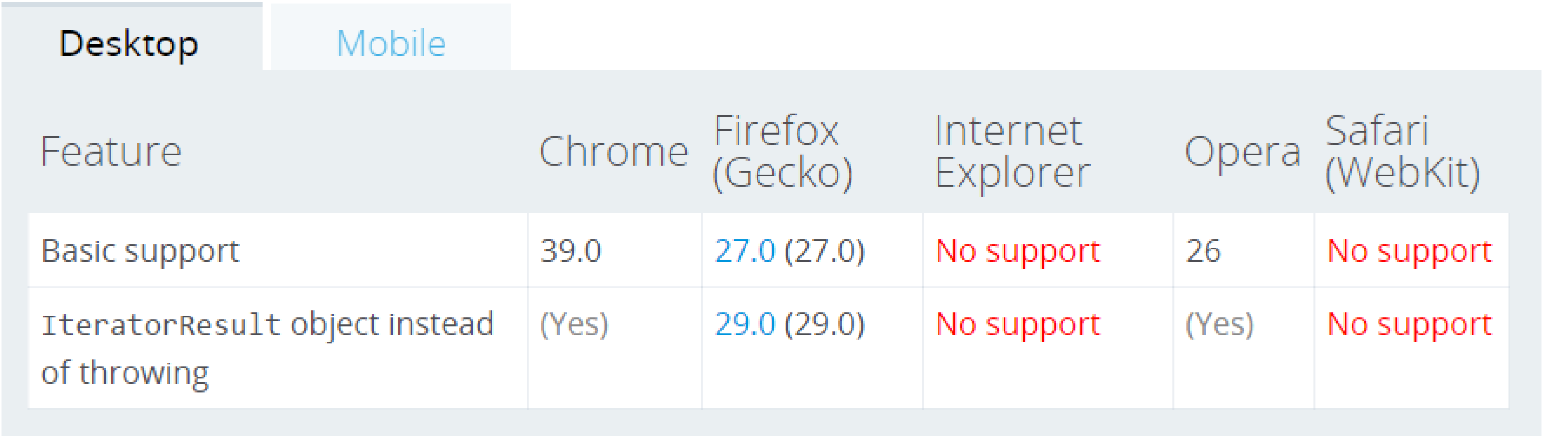
- It could also work with ES5 target, but will require:
  - JS engine supporting Symbol.iterator (nodejs 4+, Google Chrome)
  - Using ES6 lib with ES5 target (add *es6.d.ts* to your project)

```
// lib.es6.d.ts

interface IterableIterator<T> extends Iterator<T> {
    [Symbol.iterator](): IterableIterator<T>;
}
```

TS Transpiled Code

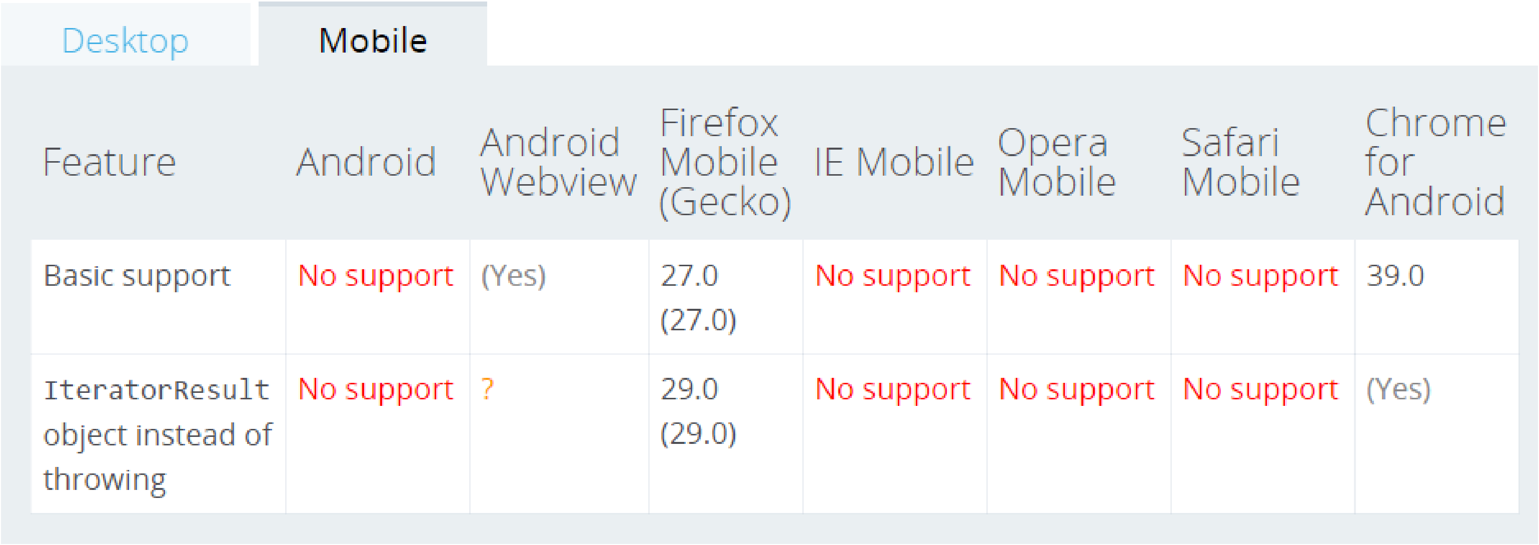
```
class IterableStuffCollection {
    constructor(stuff) {
        this.stuff = stuff;
        this.pointer = 0;
    }
    next() {
        let value = this.stuff[this.pointer];
        let done = this.pointer >= this.stuff.length;
        this.pointer++;
        return { value: value, done: done };
    }
     [Symbol.iterator]() {  // note that ES6 Symbol.iterator / iteration protocol is required
        return this;
    }
}
var myStuff = new IterableStuffCollection(['XBox One', 42, Math.PI, 'pokemon go', { oh: 'yeah' }]);
for (let item of myStuff) {
    console.log(item);
}
```

Browser Compatibility



Desktop | Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Basic support	39.0	27.0 (27.0)	No support	26	No support
`IteratorResult` object instead of throwing	(Yes)	29.0 (29.0)	No support	(Yes)	No support



Desktop | Mobile

Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	(Yes)	27.0 (27.0)	No support	No support	No support	39.0
`IteratorResult` object instead of throwing	No support	?	29.0 (29.0)	No support	No support	No support	(Yes)

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Types
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Generators

- Generators are a new breed of functions in JS, with a new syntax:

  function *

- Calling a generator function does not execute its body immediately
  - Instead, an *iterator* object for the function is returned
- We then iterate the generator by repeatedly calling *next()*
- next() executes the body function until the next *yield* expression returns a value
- Since the generator is really a function, we can call next() with arguments
- Execution can be further delegated to another generator function using a *yield * generator* expression

Generators - Motivation

1. **Lazy Iterators** – examples:
   - Return a finite or infinite list of values
   - Lazy execution/loading

2. **Externally Controlled Execution**
   - Allows a function to pause execution and pass control to the caller
   - Re-entering the function again later, while keeping context (variable bindings) across re-entrances
   - We can control its behavior by passing arguments to the generator

Generators – Lazy Iteration

```
function* idMaker() {   // generator function
  var index = 0;
  while(index < 3)   // note this is a finite iterator
    yield index++;
}

var gen = idMaker(); // returns iterator

console.log(gen.next().value);      // 0
console.log(gen.next().value);      // 1
console.log(gen.next().value);      // 2
console.log(gen.next().value);      // undefined
```

Generators – Function Args

```
function* addCallNumber (base) {
    var callNumber = 0;
    while (true) {
        yield base + callNumber++;
    }
}

var gen = addCallNumber(10); // invoke generator with argument(s)
console.log(gen.next().value);  // 10
console.log(gen.next().value);  // 11
console.log(gen.next().value);  // 12
```

Passing Arguments Into Generators

```
"use strict";

function* showPrevCurrGenerator() {

    var prev, curr;
    while (true) {
        console.log(‘-------');
        prev = curr;
        curr = yield;
        console.log(‘prev = ' + prev);
        console.log(‘curr = ' + curr);
    }
}

var gen = showPrevCurrGenerator();

gen.next(); // executes until the first yield
gen.next('First');
gen.next('Second');
gen.next('Third');
```

```
-------
prev = undefined
curr = First
-------
prev = First
curr = Second
-------
prev = Second
curr = Third
-------
```

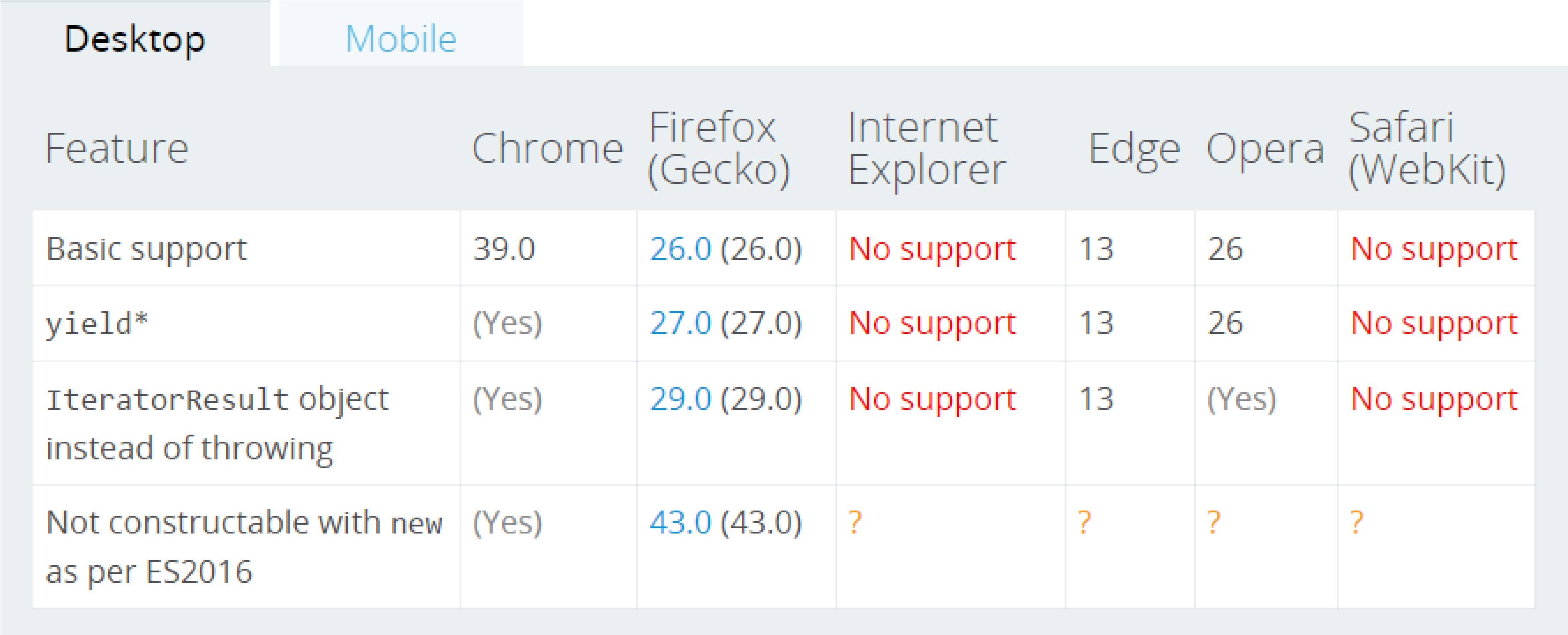
Generators – yield*

```
function* anotherGenerator(i) {
    yield i + 0.1;
    yield i + 0.2;
    yield i + 0.3;
}

function* generator(i){
    yield '0.01';
    yield* anotherGenerator(i);
    yield i * 10;
}

var gen = generator(10);

console.log(gen.next().value); // 0.01
console.log(gen.next().value); // 10.1
console.log(gen.next().value); // 10.2
console.log(gen.next().value); // 10.3
console.log(gen.next().value); // 100
```

TS Transpiled Code

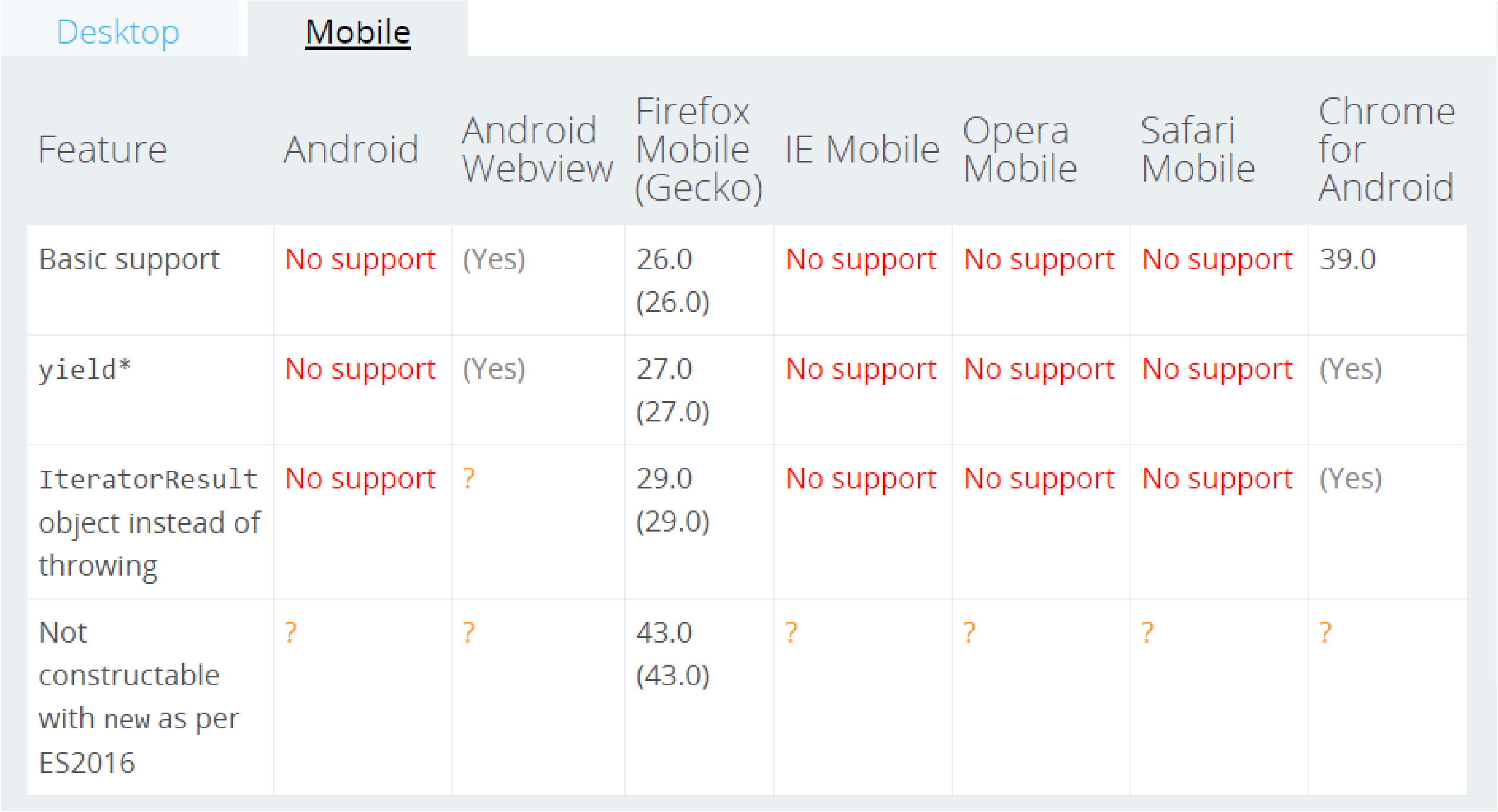
```
// This is a pure ES6 feature, TS doesn't do any magic
// Generators require ES6 support in the browser/node

function* idMaker() {
    var index = 0;
    while (index < 3) {
        yield index++;
    }
}
```

Browser Compatibility - Desktop


Desktop
Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Edge	Opera	Safari (WebKit)
Basic support	39.0	26.0 (26.0)	No support	13	26	No support
`yield*`	(Yes)	27.0 (27.0)	No support	13	26	No support
`IteratorResult` object instead of throwing	(Yes)	29.0 (29.0)	No support	13	(Yes)	No support
Not constructable with `new` as per ES2016	(Yes)	43.0 (43.0)	?	?	?	?

Browser Compatibility - Mobile



Feature	Android	Android Webview	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile	Chrome for Android
Basic support	No support	(Yes)	26.0 (26.0)	No support	No support	No support	39.0
`yield*`	No support	(Yes)	27.0 (27.0)	No support	No support	No support	(Yes)
`IteratorResult` object instead of throwing	No support	?	29.0 (29.0)	No support	No support	No support	(Yes)
Not constructable with `new` as per ES2016	?	?	43.0 (43.0)	?	?	?	?

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Types
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

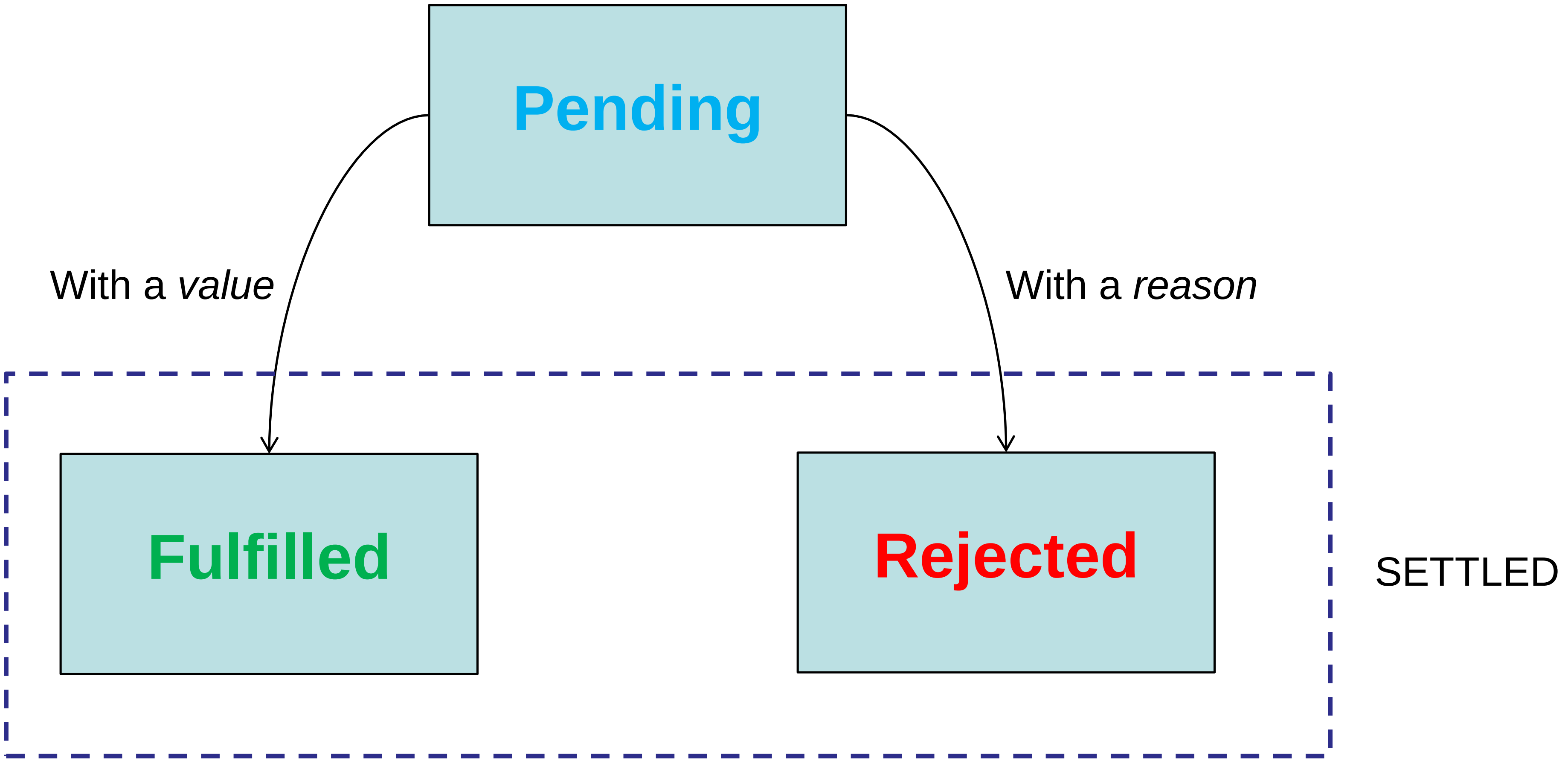
Promises

- A Promise represents an operation that hasn't completed yet, but is expected in the future
- Used for asynchronous computations
- Promises are chainable. This is a key benefit
- Syntax:

```
new Promise(function(resolve, reject) { ... } );
```

- The promise Ctor takes a single argument: an executor function
  - Executed immediately (even before returning the new Promise object)
  - *resolve* and *reject* functions are bound to the promise and calling them fulfills or rejects the promise, respectively
  - The executor function is expected to initiate some async work, and then invoke either *resolve* or *reject*

Promise States



Methods - *then*

- **Promise.then(onFulfilled, onRejected)**
  - Appends fulfillment and rejection handlers to the promise
  - Returns a new promise resolving to
    - The return value of the called handler (onFulfilled / onRejected)
    - Or to its original settled value if the promise was not handled (i.e. if the relevant handler onFulfilled or onRejected are not a function)
  - We say that promises are “thenable” objects
  - Allows us to create chains since *then()* returns a promise
    - We call this “composition”

Methods – *catch*

- **Promise.catch(onRejected)**
  - Appends a rejection handler callback to the promise
  - Returns a new promise resolving to
    - The return value of the callback if it is called
    - Or to original fulfillment value if the promise is fulfilled
  - Allows us to create chains since *catch()* returns a promise

Methods – Other

- **Promise.all(iterable)**
  - Takes a list of promises and returns a promise that
    - Resolves when all promises resolve
    - Or rejects as soon as any promise fails
- **Promise.race(iterable)**
  - Takes a list of promises and returns a promise that
    - Resolves as soon as any promise resolves
    - Or rejects as soon as any promise rejects
- **Promise.resolve(value) / Promise.reject(reason)**
  - Shortcuts returning an already resolved/rejected promise
  - Useful for example for initiating a chain

Example - Chaining

```
Promise.resolve(123)
    .then((res) => {
        console.log(res); // 123
        return 456;
    })
    .then((res) => {
        console.log(res); // 456
        return Promise.resolve(123);
    })
    .then((res) => {
        console.log(res);  // 123 : Notice `this` is called with the resolved value
        return Promise.resolve(123);
    })
```

Example – Aggregated Error Handling

```
Promise.reject(new Error('something bad happened'))
    .then((res) => {
        console.log(res); // not called
        return 456;
    })
    .then((res) => {
        console.log(res); // not called
        return Promise.resolve(123);
    })
    .then((res) => {
        console.log(res); // not called
        return Promise.resolve(123);
    })
    .catch((err) => {
        console.log(err.message); // something bad happened
    });
```

Example – *catch* Chaining

```
Promise.reject(new Error('something bad happened'))
    .then((res) => {
        console.log(res);  // not called
        return 456;
    })
    .catch((err) => {
        console.log(err.message);  // something bad happened
        return Promise.resolve(123);
    })
    .then((res) => {
        console.log(res);  // 123
    });
```

TS & Promises

- TS understands promises flow of values

```
Promise.resolve(123)
    .then((res)=>{
        // res is inferred to be of type `number`
        return true;
    })
    .then((res) => {
       // res is inferred to be of type `boolean`
    });
```

TS & Promises – cont.

- TS also understands unwrapping function calls that return a promise

```
function iReturnPromiseAfter1Second():Promise<string> {
    return new Promise((resolve)=>{
        setTimeout(()=>resolve("Hello world!"), 1000);
    });
}

Promise.resolve(123)
    .then((res)=>{
         // res is inferred to be of type `number`
         return iReturnPromiseAfter1Second();
    })
    .then((res) => {
        // res is inferred to be of type `string`
        console.log(res); // Hello world!
    });
```

TS – Converting CB to Promise

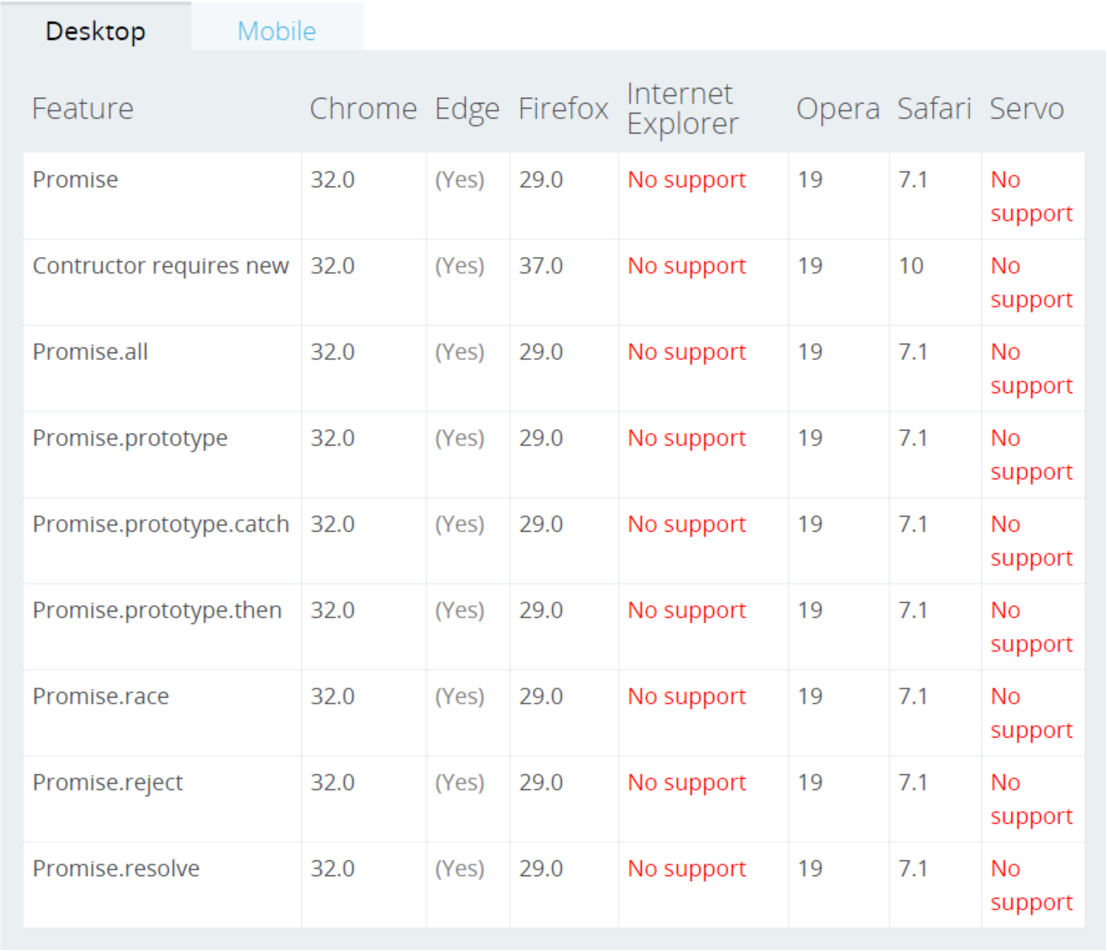
```
import fs = require('fs');

function readFileAsync (filename:string):Promise<any> {
    return new Promise((resolve,reject)=> {
        fs.readFile(filename,(err,result) => {
            if (err) reject(err);
            else resolve(result);
        });
    });
}
```

TS – Transpiled Code

- TS does not do any “magic” with promises
- It relies on ES6 promises or a polyfill

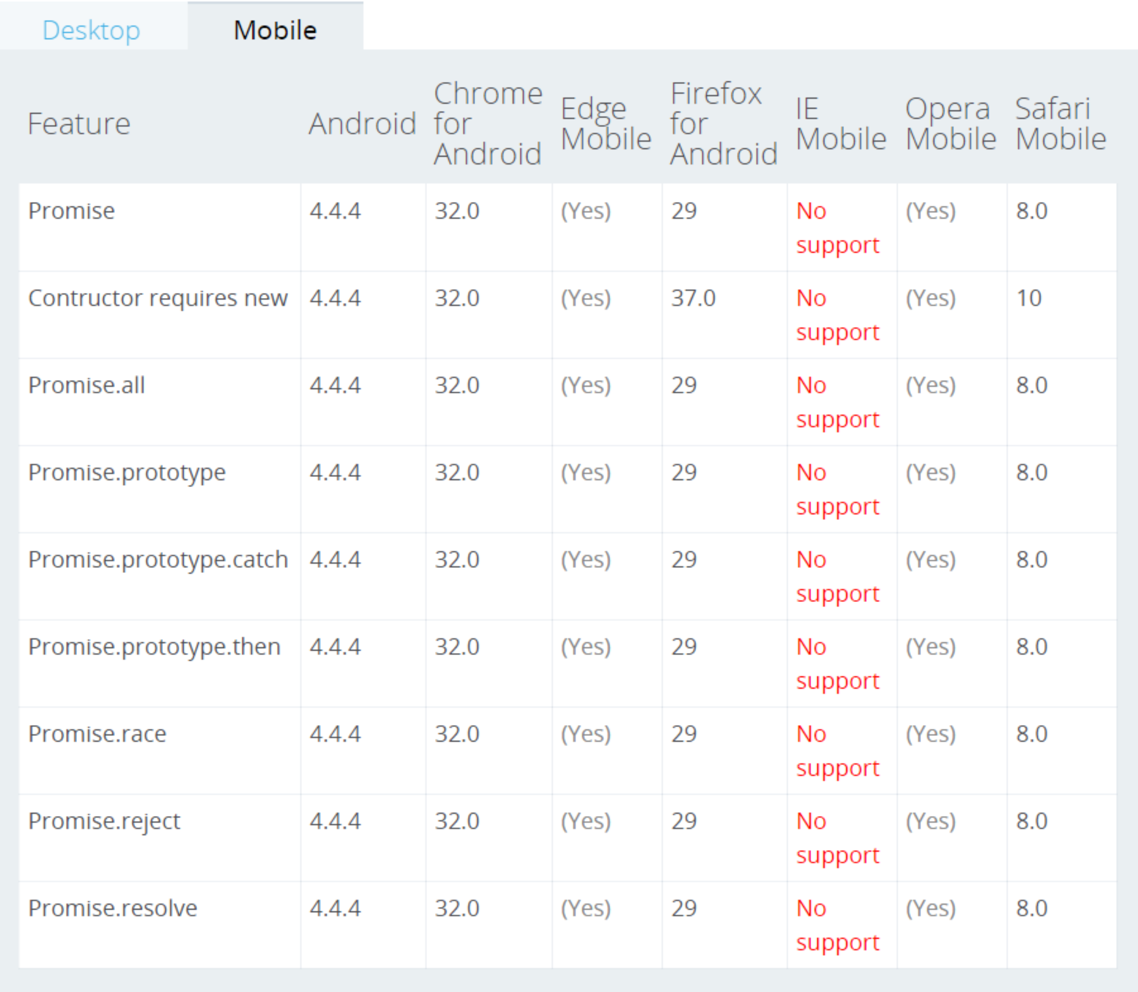
```
function iReturnPromiseAfter1Second() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Hello world!"), 1000);
    });
}
Promise.resolve(123)
    .then((res) => {
    return iReturnPromiseAfter1Second();
})
    .then((res) => {
    console.log(res); // Hello world!
});
```

Browser Compatibility - Desktop



Feature	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Servo
Promise	32.0	(Yes)	29.0	No support	19	7.1	No support
Contructor requires new	32.0	(Yes)	37.0	No support	19	10	No support
Promise.all	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.prototype	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.prototype.catch	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.prototype.then	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.race	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.reject	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.resolve	32.0	(Yes)	29.0	No support	19	7.1	No support

Browser Compatibility - Mobile



Feature	Android	Chrome for Android	Edge Mobile	Firefox for Android	IE Mobile	Opera Mobile	Safari Mobile
Promise	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Contructor requires new	4.4.4	32.0	(Yes)	37.0	No support	(Yes)	10
Promise.all	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.prototype	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.prototype.catch	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.prototype.then	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.race	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.reject	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0
Promise.resolve	4.4.4	32.0	(Yes)	29	No support	(Yes)	8.0

ECMAScript 6 & TypeScript

- Enumerable Types
- Modules
- Types
- Types
- Iterators
- Generators
- Promises
- Maps, Sets & Friends

Maps

- The Map object is a simple key/value dictionary
- Any value (both objects and primitive values) may be used as either a key or a value
- Syntax:

```
new Map([iterable])
```

- *Iterable* is an optional other iterable object whose elements are key-value pairs
- for…of looping on a map returns an [key, value] array (in insertion order) each iteration
- Key equality is based on “same value” algorithm
  - NaN is considered same as Nan (although in JS they’re not)
  - All other values go by the === semantics

Maps vs. JS Objects

- Similar in that both let us set/retrieve/delete/check values by keys
- The main differences are:
  - An Object has a prototype, so we might have default keys
  - Object keys are Strings or Symbols, but can be any value for Map
  - Map's size can be retrieved easily, difficult with an Object
- Still, in many cases it is perfectly okay to continue using Objects

Map Properties & Methods

- ***size*** – Returns the number of k/v pairs in the Map object
- ***clear()*** – Removes all k/v pairs
- ***delete(key)*** – Removes value, returns true/false if deleted/not-found
- ***entries()*** – returns a new Iterator containing an array of [k,v] pairs per each iteration, in insertion order
- ***keys()*** – Returns a new Iterator containing keys in insertion order
- ***values()*** – Returns a new Iterator containing values in insertion order
- ***forEach(cbFn [, this])*** – calls cbFn for each k/v pair in insertion order. If this is provided, will be applied to cbFn
- ***has(k)*** – Returns true if key exists in the Map
- ***get(k)*** – Returns the value if key k exists, undefined otherwise
- ***set(k, v)*** – Sets the value for the key, returns the Map (for chaining)
- ***[@@iterator]()*** – Returns a new Iterator containing [k,v] array for each element in insertion order

Maps – Example: Simple

```
var myMap = new Map();

var keyString = "a string",
    keyObj = {},
    keyFunc = function () {};

// setting the values
myMap.set(keyString, "value associated with 'a string'");
myMap.set(keyObj, "value associated with keyObj");
myMap.set(keyFunc, "value associated with keyFunc");

myMap.size;  // 3

// getting the values
myMap.get(keyString);     // "value associated with 'a string'"
myMap.get(keyObj);        // "value associated with keyObj"
myMap.get(keyFunc);       // "value associated with keyFunc"

myMap.get("a string");    // "value associated with 'a string‘” because keyString === 'a string'
myMap.get({});            // undefined, because keyObj !== {}
myMap.get(function() {}) ; // undefined, because keyFunc !== function () {}
```

Maps – Example: Iterating

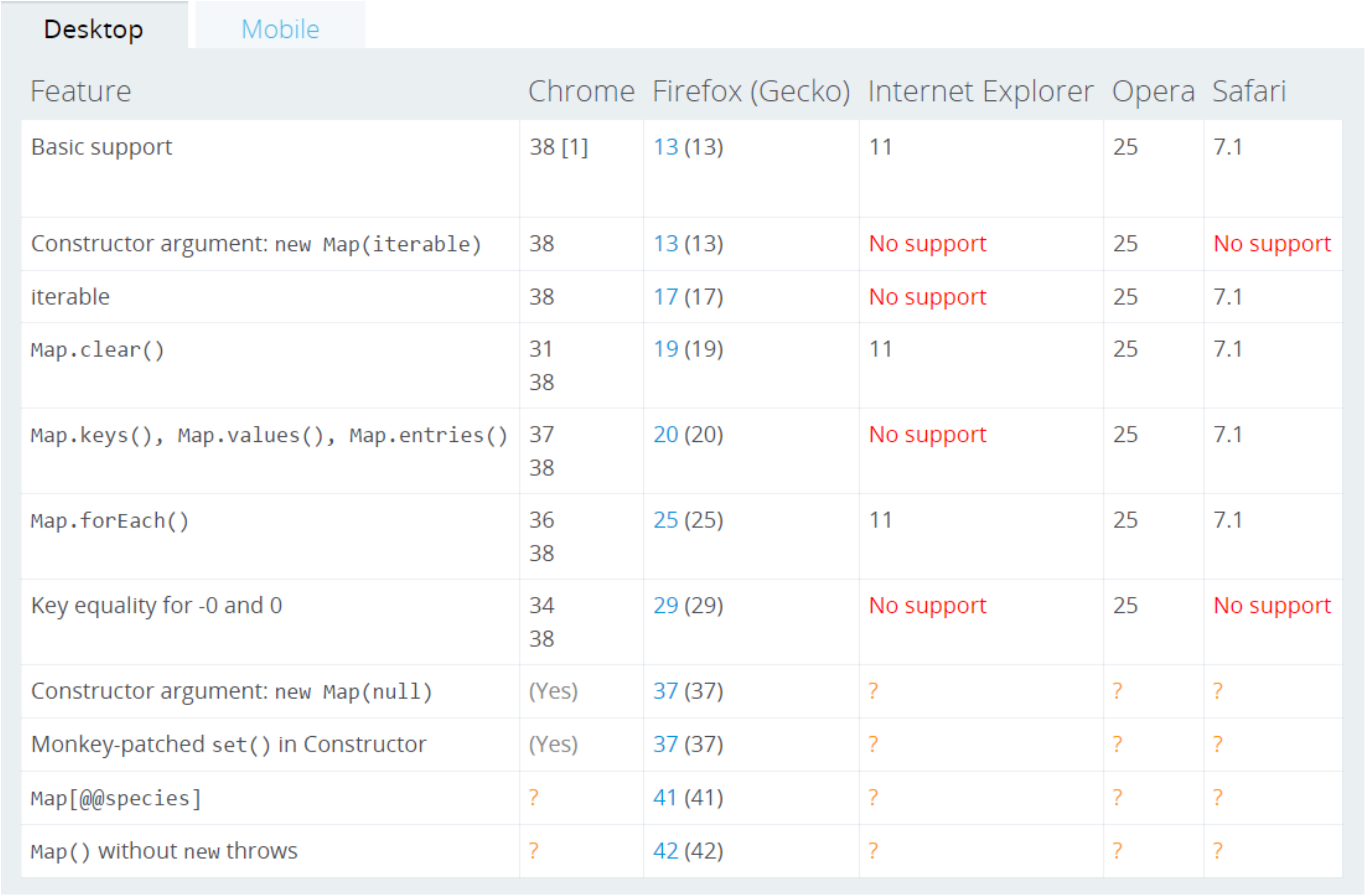
```
var myMap = new Map();

myMap.set(0, "zero");
myMap.set(1, "one");

for (var [key, value] of myMap) {  // 0 = zero, 1 = one
    console.log(key + " = " + value);
}

for (var key of myMap.keys()) {  // 0, 1
    console.log(key);
}

for (var value of myMap.values()) {  // zero, one
    console.log(value);
}

for (var [key, value] of myMap.entries()) {  // 0 = zero, 1 = one
    console.log(key + " = " + value);
}

myMap.forEach(function(value, key) {  // 0 = zero, 1 = one
    console.log(key + " = " + value);
});
```

TS – Transpiled Code

- TS does not do any “magic” with Maps
- It relies on ES6 Maps or a polyfill

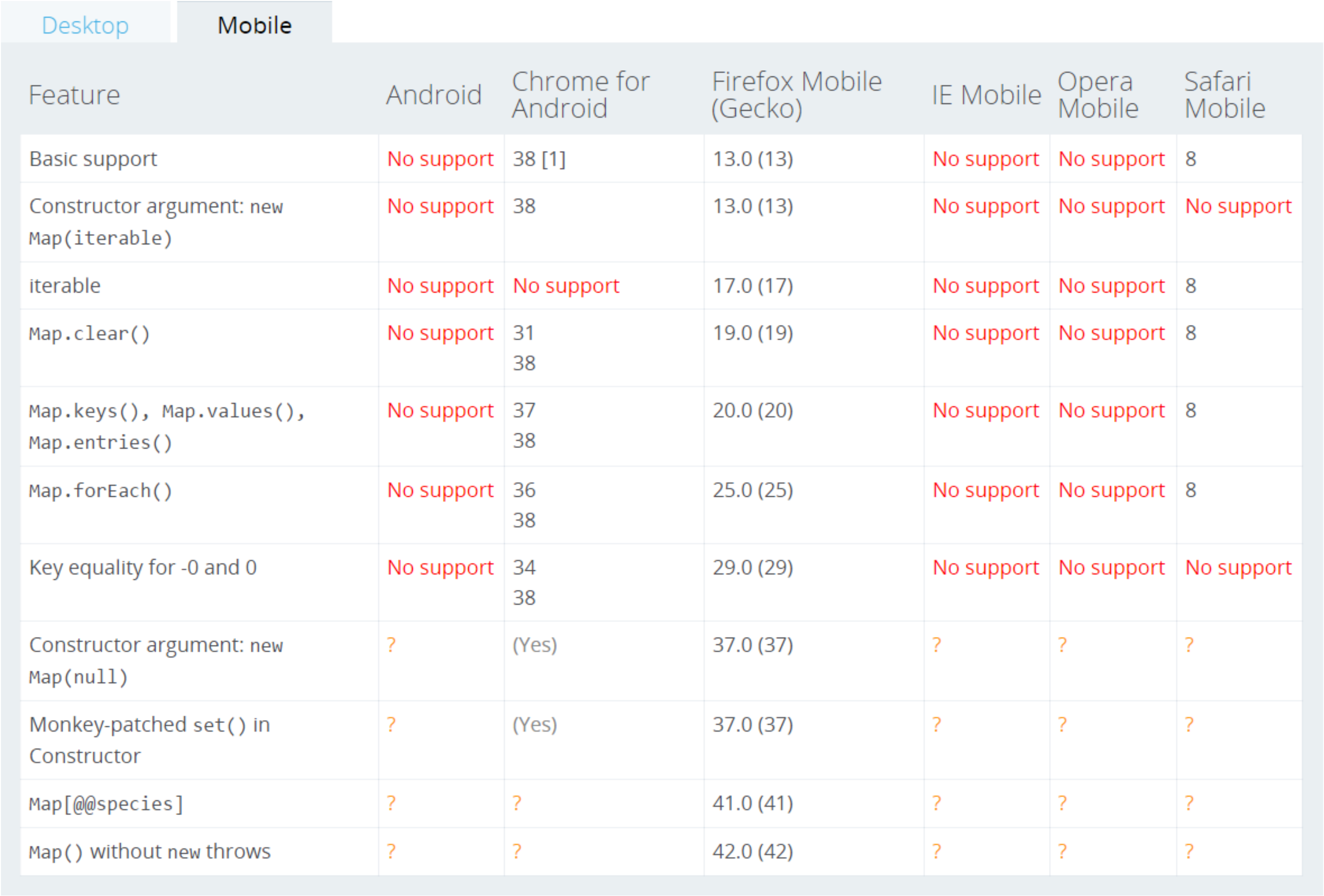
```
var myMap = new Map();

myMap.set(0, "zero");
myMap.set(1, "one");

for (var [key, value] of myMap) {
    console.log(key + " = " + value);
}
for (var key of myMap.keys()) {
    console.log(key);
}
for (var value of myMap.values()) {
    console.log(value);
}
for (var [key, value] of myMap.entries()) {
    console.log(key + " = " + value);
}
myMap.forEach(function (value, key) {
    console.log(key + " = " + value);
});
```

Browser Compatibility - Desktop


Desktop
Mobile

Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	38 [1]	13 (13)	11	25	7.1
Constructor argument: `new Map(iterable)`	38	13 (13)	No support	25	No support
iterable	38	17 (17)	No support	25	7.1
`Map.clear()`	31 38	19 (19)	11	25	7.1
`Map.keys(), Map.values(), Map.entries()`	37 38	20 (20)	No support	25	7.1
`Map.forEach()`	36 38	25 (25)	11	25	7.1
Key equality for -0 and 0	34 38	29 (29)	No support	25	No support
Constructor argument: `new Map(null)`	(Yes)	37 (37)	?	?	?
Monkey-patched `set()` in Constructor	(Yes)	37 (37)	?	?	?
`Map[@@species]`	?	41 (41)	?	?	?
`Map()` without `new` throws	?	42 (42)	?	?	?

Browser Compatibility - Mobile



Feature	Android	Chrome for Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	No support	38 [1]	13.0 (13)	No support	No support	8
Constructor argument: `new Map(iterable)`	No support	38	13.0 (13)	No support	No support	No support
iterable	No support	No support	17.0 (17)	No support	No support	8
`Map.clear()`	No support	31 38	19.0 (19)	No support	No support	8
`Map.keys(), Map.values(), Map.entries()`	No support	37 38	20.0 (20)	No support	No support	8
`Map.forEach()`	No support	36 38	25.0 (25)	No support	No support	8
Key equality for -0 and 0	No support	34 38	29.0 (29)	No support	No support	No support
Constructor argument: `new Map(null)`	?	(Yes)	37.0 (37)	?	?	?
Monkey-patched `set()` in Constructor	?	(Yes)	37.0 (37)	?	?	?
`Map[@@species]`	?	?	41.0 (41)	?	?	?
`Map()` without `new` throws	?	?	42.0 (42)	?	?	?

Sets

- Set objects are collections of values, which we can iterate according to insertion order
- Sets let us store unique values of any type, whether primitive values or object references
- Syntax:

```
new Set([iterable])
```

- If an iterable object is passed, all of its elements will be added to the new Set
- Value equality is similar to ===
- two objects are equal only if they refer to the exact same object

```
var set = new Set();
set.add({a:1});
set.add({a:1});
console.log(set.size) // 2
console.log([...set.values()]); // Array [ Object, Object ]
```

Set Properties & Methods

- ***size*** – Returns the number of elements pairs in the Set object
- ***add()*** – Appends a new element
- ***clear()*** – Removes all elements from the Set object
- ***delete(value)*** – Removes element and returns true/false if value existed(deleted) or not
- ***entries()*** – Returns a new Iterator object containing an array of [value, value] for each element, in insertion order
- ***forEach(cbFn [, this])*** – Calls cbFn for each value in the Set object in insertion order. If this is provided – will be applied to cbFn
- ***has(value)*** – Returns a boolean indicating whether value exists
- ***values()*** – Returns a new Iterator containing all element values
- ***keys()*** – Same as *values()*
- ***[@@iterator]()*** – Returns a new Iterator containing all values in insertion order

Sets – Example: Simple

```
var mySet = new Set();

mySet.add(1);
mySet.add(1); // does nothing, 1 is already in the set

mySet.add(5);
mySet.add("some text");
var o = {a: 1, b: 2};
mySet.add(o);

mySet.has(1); // true
mySet.has(3); // false
mySet.has(Math.sqrt(25));  // true (5 exists)
mySet.has("Some Text".toLowerCase()); // true
mySet.has(o); // true


mySet.size; // 4

mySet.delete(5); // removes 5 and returns true (5 existed before deletion)
mySet.has(5);    // false, 5 has been removed

mySet.size; // 3, we just removed one value
```

Sets – Example: Iterating

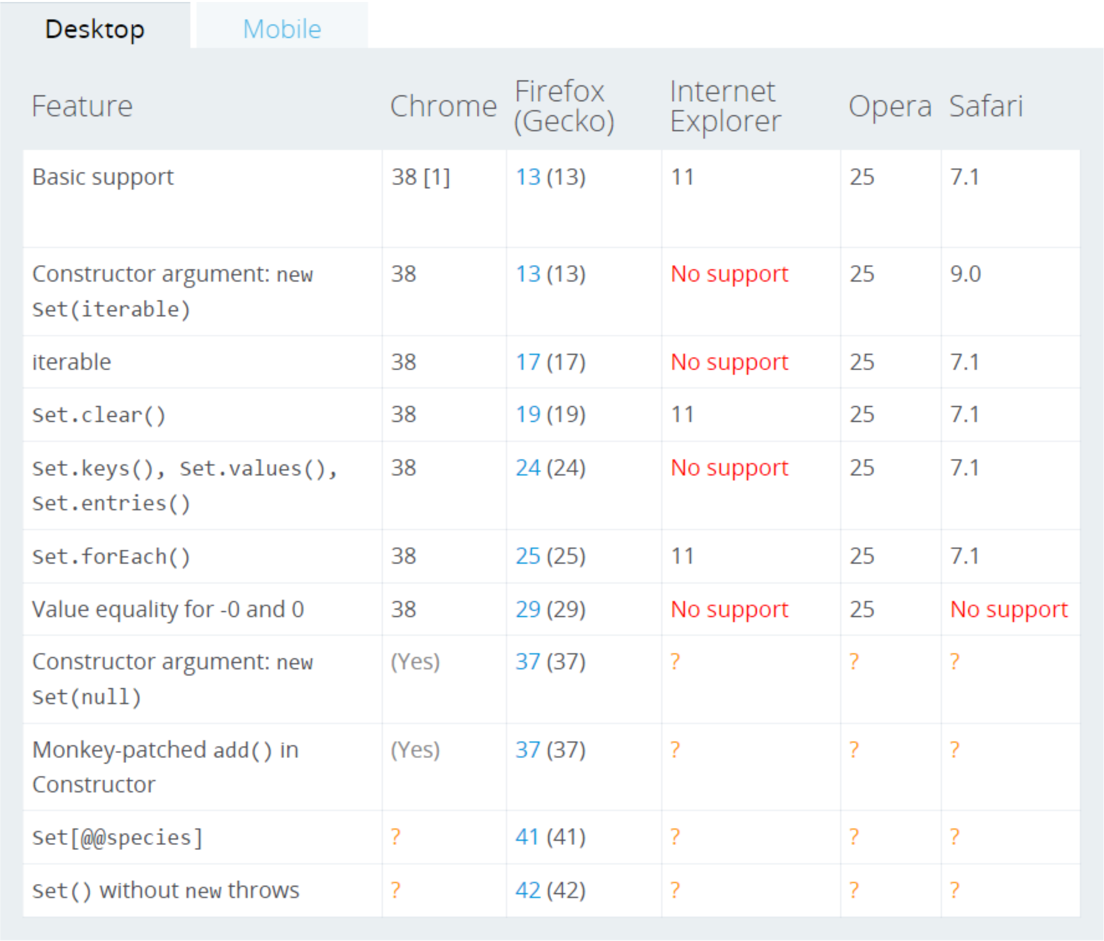
```
// ... continuing our previous example

for (let item of mySet) console.log(item); //  1, some text, Object {a: 1, b: 2}

for (let item of mySet.keys()) console.log(item); //  1, some text, Object {a: 1, b: 2}

for (let item of mySet.values()) console.log(item); //  1, some text, Object {a: 1, b: 2}

for (let [key, value] of mySet.entries()) console.log(key); //  1, some text, Object {a: 1, b: 2}

mySet.forEach(e => console.log(e)); //  1, some text, Object {a: 1, b: 2}

console.log([...mySet]); // [1, “some text”, Object]
```

TS – Transpiled Code

- TS does not do any “magic” with Sets
- It relies on ES6 Sets or a polyfill

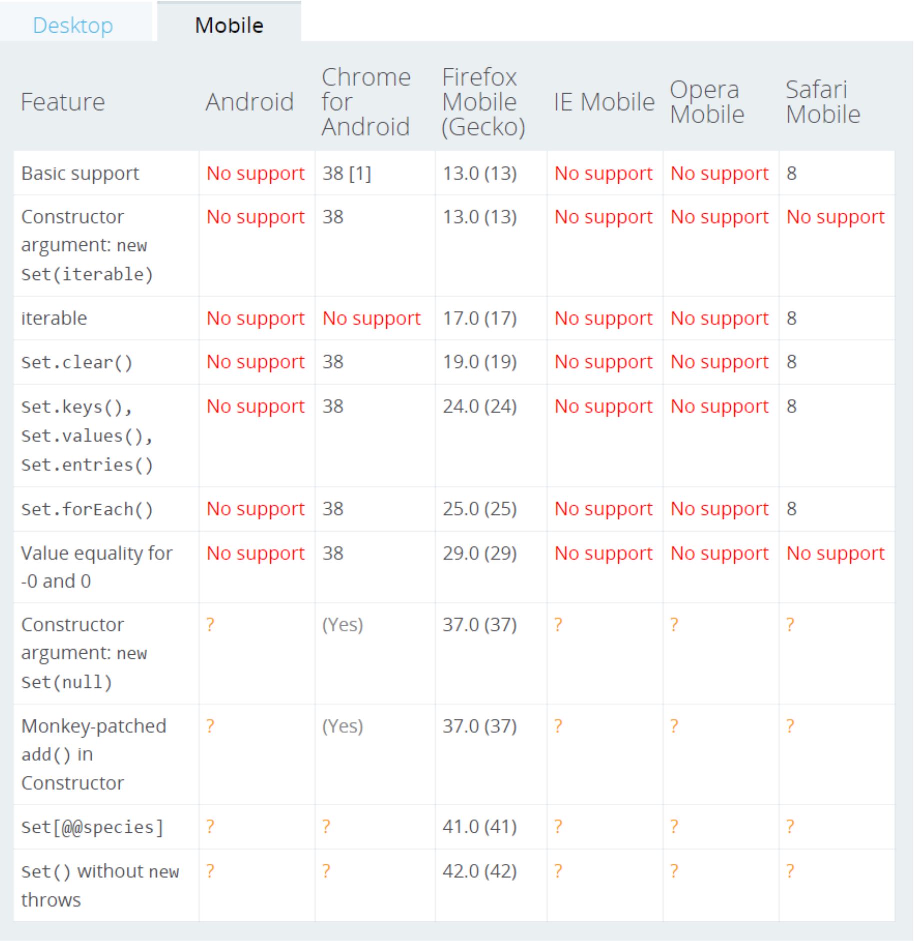
```
var mySet = new Set();

mySet.add(1);
mySet.add(5);
mySet.add("some text");

var o = { a: 1, b: 2 };
mySet.add(o);

mySet.has(1);
```

Browser Compatibility - Desktop



Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	38 [1]	13 (13)	11	25	7.1
Constructor argument: `new Set(iterable)`	38	13 (13)	No support	25	9.0
iterable	38	17 (17)	No support	25	7.1
`Set.clear()`	38	19 (19)	11	25	7.1
`Set.keys(), Set.values(), Set.entries()`	38	24 (24)	No support	25	7.1
`Set.forEach()`	38	25 (25)	11	25	7.1
Value equality for -0 and 0	38	29 (29)	No support	25	No support
Constructor argument: `new Set(null)`	(Yes)	37 (37)	?	?	?
Monkey-patched `add()` in Constructor	(Yes)	37 (37)	?	?	?
`Set[@@species]`	?	41 (41)	?	?	?
`Set()` without `new` throws	?	42 (42)	?	?	?

Browser Compatibility - Mobile



Feature	Android	Chrome for Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	No support	38 [1]	13.0 (13)	No support	No support	8
Constructor argument: `new Set(iterable)`	No support	38	13.0 (13)	No support	No support	No support
iterable	No support	No support	17.0 (17)	No support	No support	8
`Set.clear()`	No support	38	19.0 (19)	No support	No support	8
`Set.keys(), Set.values(), Set.entries()`	No support	38	24.0 (24)	No support	No support	8
`Set.forEach()`	No support	38	25.0 (25)	No support	No support	8
Value equality for -0 and 0	No support	38	29.0 (29)	No support	No support	No support
Constructor argument: `new Set(null)`	?	(Yes)	37.0 (37)	?	?	?
Monkey-patched `add()` in Constructor	?	(Yes)	37.0 (37)	?	?	?
`Set[@@species]`	?	?	41.0 (41)	?	?	?
`Set()` without `new` throws	?	?	42.0 (42)	?	?	?

WeakMap & WeakSet

- The “Weak” counterparts of Map and Set
- Weakly hold references to keys/values stored
- Adding an element to the collection does’nt increase reference count
- When the element is freed up, the collection will no longer contain that element
- Syntax:

```
new WeakMap([iterable])
new WeakSet([iterable])
```

WeakMap & WeakSet – Cont.

- When there are no more references (in our code) to an object stored in the collection, it is garbage collected
- That means there is no list of objects stored in the collection
- Therefore weak collections are not enumarable
- Available methods – WeakMap:
  - ***delete(), get(key), has(key), set(key, value)***
- Available methods – WeakSet:
  - ***add(value), get(value), has(value)***

WeakMap - Example

```
var wm = new WeakMap();

var keys = {
    key1: {}
};

wm.set(keys.key1, "some value associated with key");

console.log(wm.get(keys.key1)); // "some value associated with key"

delete keys.key1; // we’ll now delete the key object

console.log(wm.get(keys.key1)); // undefined
```

WeakSet - Example

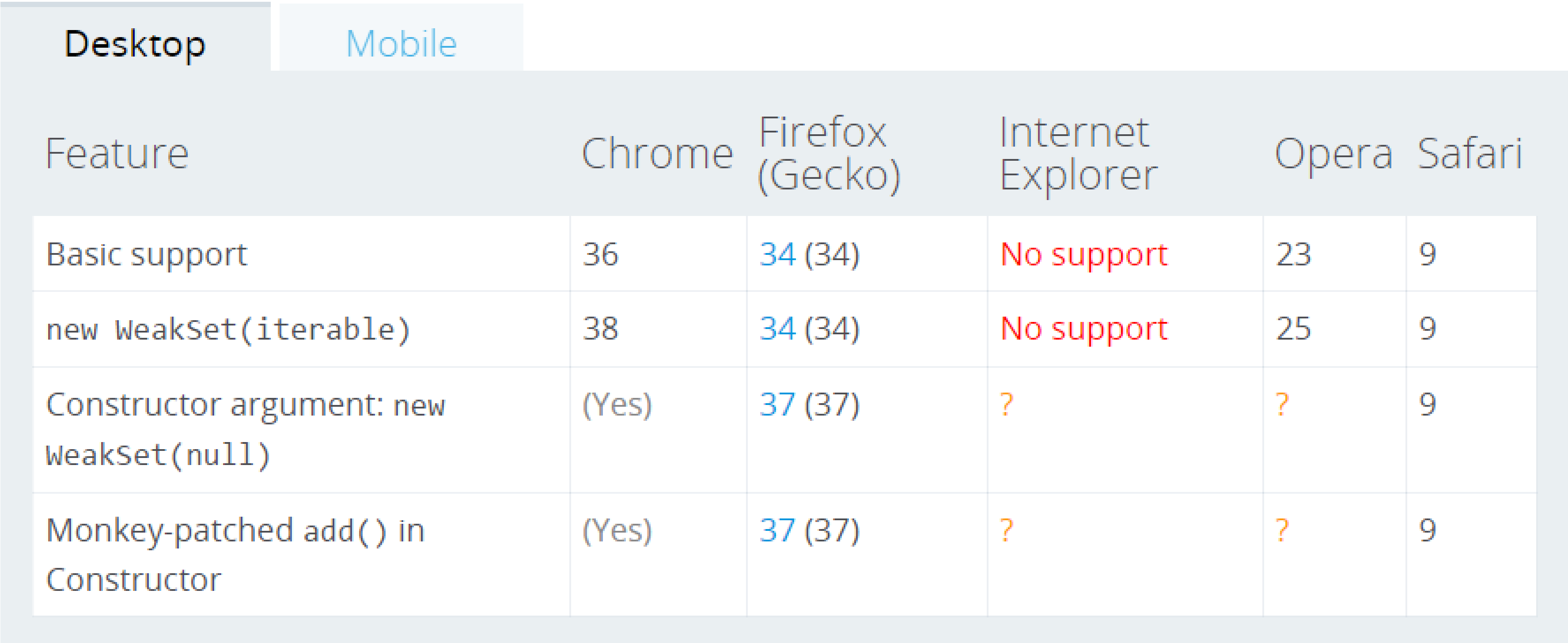
```
var ws = new WeakSet();

var keys = {
    key1: {}
};

ws.add(keys.key1);

console.log(ws.has(keys.key1)); // true

delete keys.key1; // we'll now delete the key object

console.log(ws.has(keys.key1)); // false
```

TS – Transpiled Code

- TS does not do any “magic” with weak collections
- It relies on the ES6 feature or polyfills

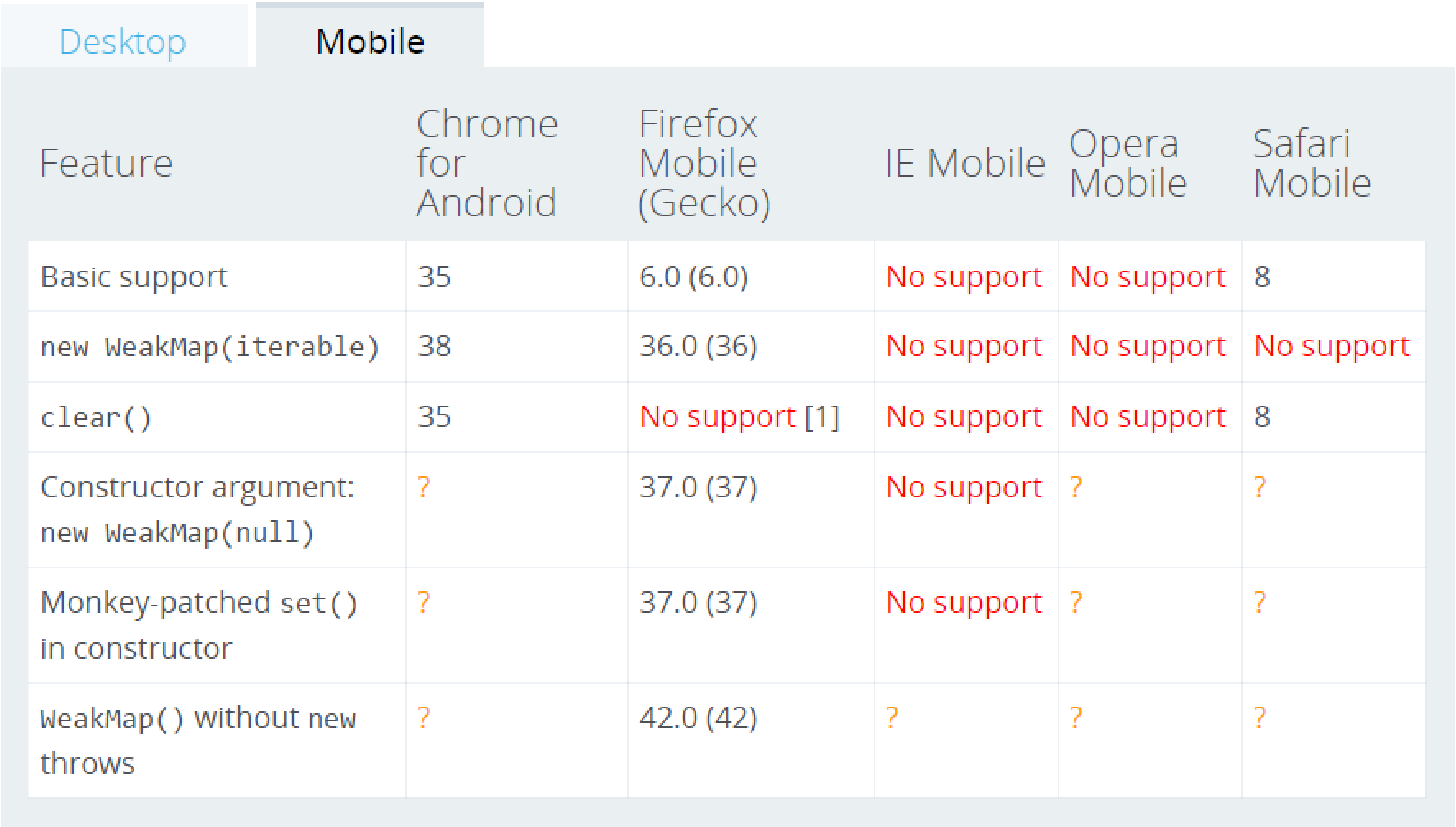
```
var ws = new WeakSet();
var wm = new WeakMap();
```

Browser Compatibility – WeakMap Desktop



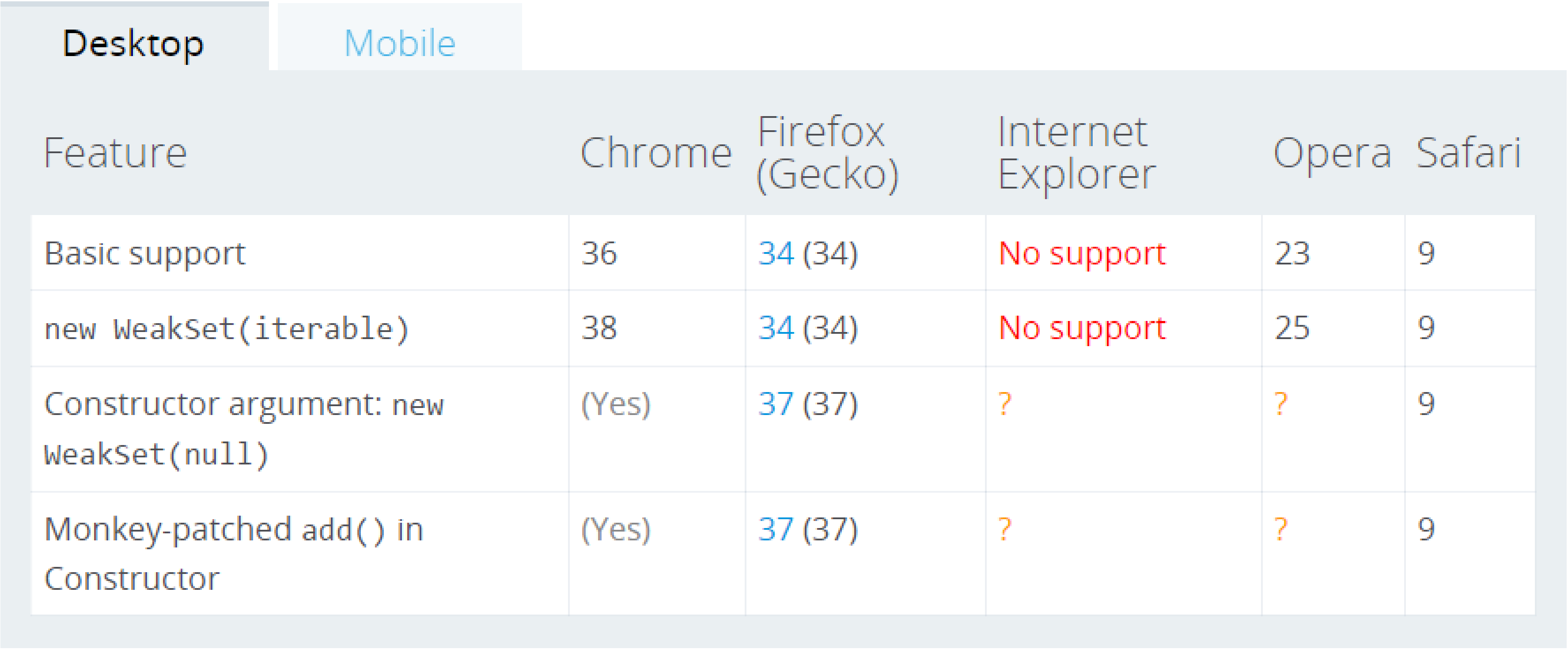
Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	36	34 (34)	No support	23	9
`new WeakSet(iterable)`	38	34 (34)	No support	25	9
Constructor argument: `new WeakSet(null)`	(Yes)	37 (37)	?	?	9
Monkey-patched `add()` in Constructor	(Yes)	37 (37)	?	?	9

Browser Compatibility – WeakMap Mobile



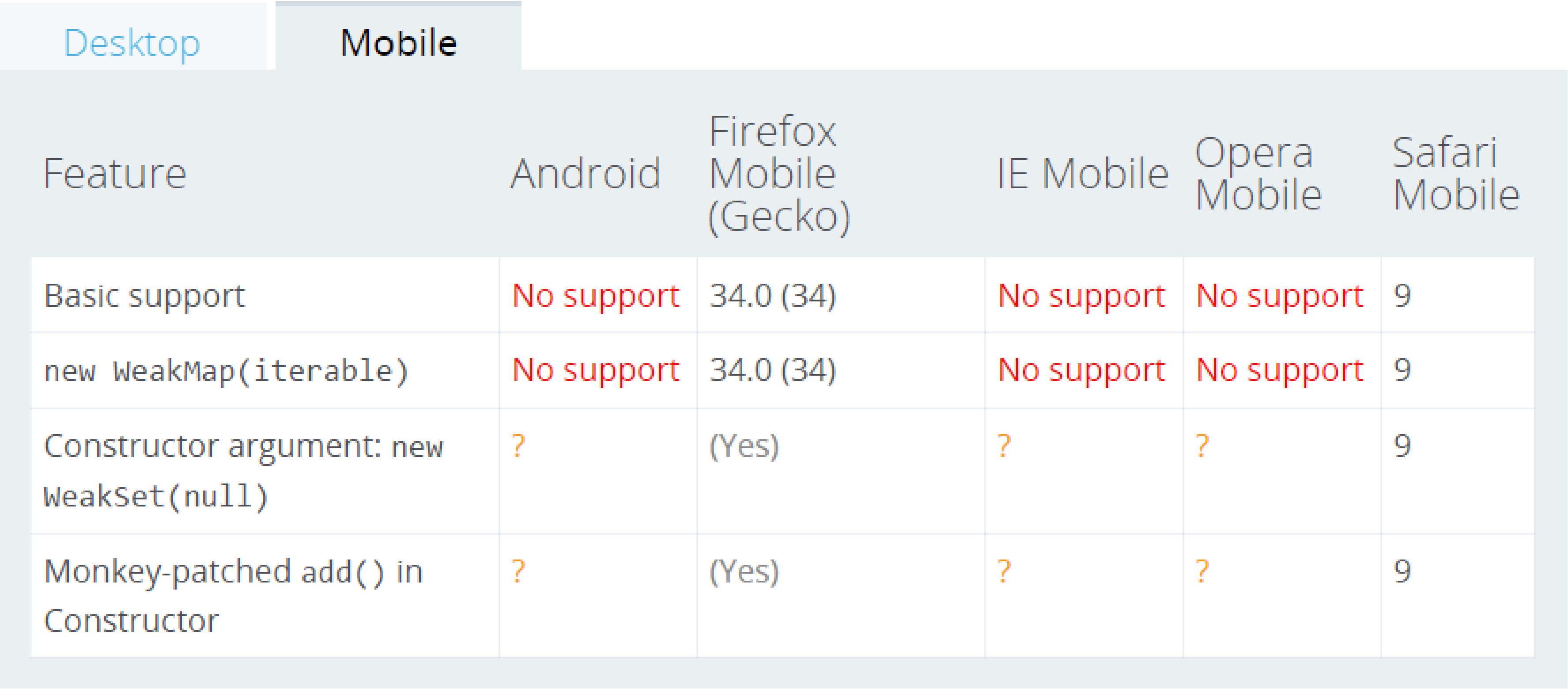
Feature	Chrome for Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	35	6.0 (6.0)	No support	No support	8
`new WeakMap(iterable)`	38	36.0 (36)	No support	No support	No support
`clear()`	35	No support [1]	No support	No support	8
Constructor argument: `new WeakMap(null)`	?	37.0 (37)	No support	?	?
Monkey-patched `set()` in constructor	?	37.0 (37)	No support	?	?
`WeakMap()` without `new` throws	?	42.0 (42)	?	?	?

Browser Compatibility – WeakSet Desktop



Feature	Chrome	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	36	34 (34)	No support	23	9
`new WeakSet(iterable)`	38	34 (34)	No support	25	9
Constructor argument: `new WeakSet(null)`	(Yes)	37 (37)	?	?	9
Monkey-patched `add()` in Constructor	(Yes)	37 (37)	?	?	9

Browser Compatibility – WeakSet Mobile



Desktop Mobile

Feature	Android	Firefox Mobile (Gecko)	IE Mobile	Opera Mobile	Safari Mobile
Basic support	No support	34.0 (34)	No support	No support	9
`new WeakMap(iterable)`	No support	34.0 (34)	No support	No support	9
Constructor argument: `new WeakSet(null)`	?	(Yes)	?	?	9
Monkey-patched `add()` in Constructor	?	(Yes)	?	?	9

Who's awesome?
You are
quickmeme.com