# ipaddr.js — an IPv6 and IPv4 address manipulation library [Build Status](https://travis-ci.org/whitequark/ipaddr.js)

ipaddr.js is a small (1.9K minified and gzipped) library for manipulating IP addresses in JavaScript environments. It runs on both CommonJS runtimes (e.g. [nodejs](http://nodejs.org)) and in a web browser.

ipaddr.js allows you to verify and parse string representation of an IP address, match it against a CIDR range or range list, determine if it falls into some reserved ranges (examples include loopback and private ranges), and convert between IPv4 and IPv4-mapped IPv6 addresses.

## Installation

npm install ipaddr.js

or

bower install ipaddr.js

## API

ipaddr.js defines one object in the global scope: ipaddr. In CommonJS, it is exported from the module:

var ipaddr = require('ipaddr.js');

The API consists of several global methods and two classes: ipaddr.IPv6 and ipaddr.IPv4.

### Global methods

There are three global methods defined: ipaddr.isValid, ipaddr.parse and ipaddr.process. All of them receive a string as a single parameter.

The ipaddr.isValid method returns true if the address is a valid IPv4 or IPv6 address, and false otherwise. It does not throw any exceptions.

The ipaddr.parse method returns an object representing the IP address, or throws an Error if the passed string is not a valid representation of an IP address.

The ipaddr.process method works just like the ipaddr.parse one, but it automatically converts IPv4-mapped IPv6 addresses to their IPv4 counterparts before returning. It is useful when you have a Node.js instance listening on an IPv6 socket, and the net.ivp6.bindv6only sysctl parameter (or its equivalent on non-Linux OS) is set to 0. In this case, you can accept IPv4 connections on your IPv6-only socket, but the remote address will be mangled. Use ipaddr.process method to automatically demangle it.

### Object representation

Parsing methods return an object which descends from ipaddr.IPv6 or ipaddr.IPv4. These objects share some properties, but most of them differ.

#### Shared properties

One can determine the type of address by calling addr.kind(). It will return either "ipv6" or "ipv4".

An address can be converted back to its string representation with addr.toString(). Note that this method:

* does not return the original string used to create the object (in fact, there is no way of getting that string)
* returns a compact representation (when it is applicable)

A match(range, bits) method can be used to check if the address falls into a certain CIDR range. Note that an address can be (obviously) matched only against an address of the same type.

For example:

var addr = ipaddr.parse("2001:db8:1234::1");

var range = ipaddr.parse("2001:db8::");

addr.match(range, 32); // => true

Alternatively, match can also be called as match([range, bits]). In this way, it can be used together with the parseCIDR(string) method, which parses an IP address together with a CIDR range.

For example:

var addr = ipaddr.parse("2001:db8:1234::1");

addr.match(ipaddr.parseCIDR("2001:db8::/32")); // => true

A range() method returns one of predefined names for several special ranges defined by IP protocols. The exact names (and their respective CIDR ranges) can be looked up in the source: [IPv6 ranges](https://github.com/whitequark/ipaddr.js/blob/master/src/ipaddr.coffee#L186) and [IPv4 ranges](https://github.com/whitequark/ipaddr.js/blob/master/src/ipaddr.coffee#L71). Some common ones include "unicast" (the default one) and "reserved".

You can match against your own range list by using ipaddr.subnetMatch(address, rangeList, defaultName) method. It can work with a mix of IPv6 or IPv4 addresses, and accepts a name-to-subnet map as the range list. For example:

var rangeList = {

documentationOnly: [ ipaddr.parse('2001:db8::'), 32 ],

tunnelProviders: [

[ ipaddr.parse('2001:470::'), 32 ], // he.net

[ ipaddr.parse('2001:5c0::'), 32 ] // freenet6

]

};

ipaddr.subnetMatch(ipaddr.parse('2001:470:8:66::1'), rangeList, 'unknown'); // => "tunnelProviders"

The addresses can be converted to their byte representation with toByteArray(). (Actually, JavaScript mostly does not know about byte buffers. They are emulated with arrays of numbers, each in range of 0..255.)

var bytes = ipaddr.parse('2a00:1450:8007::68').toByteArray(); // ipv6.google.com

bytes // => [42, 0x00, 0x14, 0x50, 0x80, 0x07, 0x00, <zeroes...>, 0x00, 0x68 ]

The ipaddr.IPv4 and ipaddr.IPv6 objects have some methods defined, too. All of them have the same interface for both protocols, and are similar to global methods.

ipaddr.IPvX.isValid(string) can be used to check if the string is a valid address for particular protocol, and ipaddr.IPvX.parse(string) is the error-throwing parser.

ipaddr.IPvX.isValid(string) uses the same format for parsing as the POSIX inet\_ntoa function, which accepts unusual formats like 0xc0.168.1.1 or 0x10000000. The function ipaddr.IPv4.isValidFourPartDecimal(string) validates the IPv4 address and also ensures that it is written in four-part decimal format.

#### IPv6 properties

Sometimes you will want to convert IPv6 not to a compact string representation (with the :: substitution); the toNormalizedString() method will return an address where all zeroes are explicit.

For example:

var addr = ipaddr.parse("2001:0db8::0001");

addr.toString(); // => "2001:db8::1"

addr.toNormalizedString(); // => "2001:db8:0:0:0:0:0:1"

The isIPv4MappedAddress() method will return true if this address is an IPv4-mapped one, and toIPv4Address() will return an IPv4 object address.

To access the underlying binary representation of the address, use addr.parts.

var addr = ipaddr.parse("2001:db8:10::1234:DEAD");

addr.parts // => [0x2001, 0xdb8, 0x10, 0, 0, 0, 0x1234, 0xdead]

A IPv6 zone index can be accessed via addr.zoneId:

var addr = ipaddr.parse("2001:db8::%eth0");

addr.zoneId // => 'eth0'

#### IPv4 properties

toIPv4MappedAddress() will return a corresponding IPv4-mapped IPv6 address.

To access the underlying representation of the address, use addr.octets.

var addr = ipaddr.parse("192.168.1.1");

addr.octets // => [192, 168, 1, 1]

prefixLengthFromSubnetMask() will return a CIDR prefix length for a valid IPv4 netmask or null if the netmask is not valid.

ipaddr.IPv4.parse('255.255.255.240').prefixLengthFromSubnetMask() == 28

ipaddr.IPv4.parse('255.192.164.0').prefixLengthFromSubnetMask() == null

subnetMaskFromPrefixLength() will return an IPv4 netmask for a valid CIDR prefix length.

ipaddr.IPv4.subnetMaskFromPrefixLength(24) == "255.255.255.0"

ipaddr.IPv4.subnetMaskFromPrefixLength(29) == "255.255.255.248"

broadcastAddressFromCIDR() will return the broadcast address for a given IPv4 interface and netmask in CIDR notation.

ipaddr.IPv4.broadcastAddressFromCIDR("172.0.0.1/24") == "172.0.0.255"

networkAddressFromCIDR() will return the network address for a given IPv4 interface and netmask in CIDR notation.

ipaddr.IPv4.networkAddressFromCIDR("172.0.0.1/24") == "172.0.0.0"

#### Conversion

IPv4 and IPv6 can be converted bidirectionally to and from network byte order (MSB) byte arrays.

The fromByteArray() method will take an array and create an appropriate IPv4 or IPv6 object if the input satisfies the requirements. For IPv4 it has to be an array of four 8-bit values, while for IPv6 it has to be an array of sixteen 8-bit values.

For example:

var addr = ipaddr.fromByteArray([0x7f, 0, 0, 1]);

addr.toString(); // => "127.0.0.1"

or

var addr = ipaddr.fromByteArray([0x20, 1, 0xd, 0xb8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1])

addr.toString(); // => "2001:db8::1"

Both objects also offer a toByteArray() method, which returns an array in network byte order (MSB).

For example:

var addr = ipaddr.parse("127.0.0.1");

addr.toByteArray(); // => [0x7f, 0, 0, 1]

or

var addr = ipaddr.parse("2001:db8::1");

addr.toByteArray(); // => [0x20, 1, 0xd, 0xb8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]