

Contents

[Contents](#)

[Benefits of this Library](#)

[Supported IP Address Formats](#)

[Core classes](#)

[Parse String Representation of IP Address or Host Name](#)

[Address Sections](#)

[IP Address Operations](#)

[Parse String Representation of MAC Address](#)

[MAC Address Operations](#)

[IPv6 - MAC Address Integration](#)

[Address Framework](#)

[Conversion to String Representation of Address](#)

[Searching Text of Databases for all Addresses in a Given Subnet](#)

[Containment and Subnet Membership](#)

[DNS Resolution and URLs](#)

[Sorting and Comparisons](#)

[Networks and Caches](#)

Benefits of this Library

The library was intended to satisfy the following primary goals:

- **Support parsing of all common host and ipv4/ipv6 address formats in usage** plus some additional formats (see below or see javadoc for the IPAddressString class for the extensive list)
- **Support both the parsing and representation of subnets**, either those specified by network prefix length or those specified with ranges of segment values. For example, all strings in the list below represent the same IPv4 subnet:
 - with CIDR network prefix length: 1.2.0.0/16
 - with mask: 1.2.0.0/255.255.0.0
 - wildcard segments: 1.2.*.*
 - range segments: 1.2.0-255.0-255
 - range using inet_aton format: 0x1.0x2.0x0-0xffff
 - SQL-style single wildcards to end segments: 1.2.____.____
 - IPv4 mapped IPv6: ::ffff:1.2.0.0/112

- **Allow the separation of address parsing from host parsing.** In some cases you may have an address, in others you may have a host name, in some cases either one, so this supports all three options (for instance, when validating invalid input "1.2.3.a" as an address only, it will not be treated as a host with DNS lookup attempted in the way that `InetAddress.getByName` does)
- **Allow control over which formats are allowed**, whether IPv4/6, or subnets, or `inet_aton` formats, and so on.
- **Support the production of collections of valid IPv4 and IPv6 address strings of different formats for a given address.**
Some addresses can have hundreds of thousands of potential string representations (when you consider hex capitalization, ipv6 compression, and leading zeros, the various IPv4 and IPv6 formats, and combinations of all the above), although there are generally a handful of commonly used formats. Generating these strings, whether the handful of commonly used strings or whether the exhaustive lists of all possible strings, can help when searching or matching addresses in databases or text.
- **Support parsing of all common MAC Address formats in usage**
- **Integration of MAC Address with IPv6**
- **Polymorphism** in the code for IPv4 and IPv6 for applications which must support both transparently. You can write generic non-version-specific code to validate addresses, connect to addresses, print addresses, mask addresses, etc.
- **Thread-safety and immutability.** The core objects (host names, address strings, addresses, address sections, address segments) are all immutable (like `java.lang.String` or `java.lang.Integer` instances). They do not change their underlying value.
- **Manipulation of addresses with prefixes, masks, sections, segments and subnetting:** applying network prefixes, masking, splitting addresses into sections, joining address segments into larger segments, generating new subnets and supernets from existing addresses and subnets, iterating through subnets, checking containment within subnets, ipv4-ipv6 conversion, ipv6-mac conversion
- **Sorting and comparison** of host names, addresses, address strings and subnets
- **Integrate with the java classes `InetAddress`, `Inet6Address`, `Inet4Address`.**

Supported IP Address Formats

This includes, those supported by the well-known routines `inet_aton` and `inet_pton`, the subnet formats listed above, all combinations of the above, and others:

- all the formats supported by `inet_pton` and `inet_aton`
- all the formats supported by `nmap`
- the subnet formats listed above, whether prefixed, masked, wildcards, ranges
- IPv6 canonical, compressed (1::1), mixed (1:2:3:4:5:6:1.2.3.4), [bracketed], and so on
- Hex values
- IPv6 base 85 values
- * represents all addresses both ipv4 and ipv6
- /x with no associated address is interpreted as the mask for prefix length x
- "" the empty string is considered the default loopback

For a more detailed list of formats parsed, some examples are below, or see the javadoc for `IPAddressString`.

Subnet formats

- **CIDR (Classless Inter-Domain Routing) prefix length subnets**
Adding the prefix length /x creates the subnet for that network prefix length. The subnet 1.2.0.0/16 is the set of all addresses starting with 1.2

- **Wildcard (* _) and range (-) subnets:**
 - * denotes all possible values in one or more segments, so 1.*.* or just 1.* is equivalent to 1.0.0.0/8
 - 0-1 denotes the range from 0 to 1
 - _ replaces any digit at the end of a segment, for example 1_ represents 10 to 19 in decimal or 10 to 1f in hex
- **Combinations:**
 - Applying a prefix length to a subnet simply applies the prefix to every element of the subnet. 1.*.0.0/16 is the same subnet of addresses as 1.0.0.0/8

For a more detailed list of formats parsed, some examples are below, or see the javadoc for `IPAddressString`.

Core classes

The core classes are `HostName`, `IPAddressString`, and `MACAddressString` along with the `Address` base class and its subclasses `IPAddress`, `IPv4Address`, `IPv6Address`, and `MACAddress`. If you have a textual representation of an IP address, then start with `HostName` or `IPAddressString`. If you have numeric bytes or integers, then start with `IPv4Address`, `IPv6Address` or `MACAddress`. Note that address instances can represent either a single address or a subnet. If you have an address or host with a port, then use `HostName`.

Parse String Representation of IP Address or Host Name

`IPAddressString` is used to convert. You can use one of `getAddress` or `toAddress`, the difference being whether parsing errors are handled by exception or not.

```
IPAddress address = new IPAddressString("1.2.3.4").getAddress();
if(address != null) {
    //use address here
}

or

try {
    IPAddress address = new IPAddressString("1.2.3.4").toAddress();
    //use address here
} catch (AddressStringException e) {
    String msg = e.getMessage();//detailed message indicating issue
}
```

If you have either a host name or an address, you can use `HostName`:

```
public static void main(String[] args) {
    check(new HostName("[::1]"));
    check(new HostName("*"));
    check(new HostName("a.b.com"));
}

static void check(HostName host) {
    if(host.isAddress()) {
        System.out.println("address: " + host.asAddress().toCanonicalString());
    } else if(host.isAddressString()) {
        System.out.println("address string with ambiguous address: " +
            host.asAddressString());
    } else {

```

```
        System.out.println("host name with labels: " +
            Arrays.asList(host.getNormalizedLabels()));
    }
}
```

Output:

```
address: ::1
address string with ambiguous address: *
host name with labels: [a, b, com]
```

Format Examples

Many formats are supported. For instance, the address 1:2:3:0:0:6:: can be represented many ways as shown.

```

static void parse(String formats[]) {
    for(String format : formats) {
        System.out.println(new IPAddressString(format).getAddress());
    }
}

static void parseHost(String formats[]) {
    for(String format : formats) {
        System.out.println(new HostName(format).getAddress());
    }
}

public static void main(String[] args) {
    String formats[] = {
        "1:2:3:0:0:6::",
        "1:2:3:0:0:6:0:0",
        "1:2:3::6:0:0",
        "0001:0002:0003:0000:0000:0006:0000:0000",
        "1:2:3::6:0.0.0.0",
        "1:2:3:0:0:6::",
        "008JQW0V70(=61h*;$LC",
        "0x00010002000300000000000060000000"
    };
    parse(formats);
    String hostFormats[] = {
        "[1:2:3:0:0:6::]",
        "[1:2:3:0:0:6:0:0]",
        "[1:2:3::6:0:0]",
        "[0001:0002:0003:0000:0000:0006:0000:0000]",
        "[1:2:3::6:0.0.0.0]",
        "[1:2:3:0:0:6::]",
        "[008JQW0V70(=61h*;$LC]",
        "[0x00010002000300000000000060000000]",
        "0.0.0.0.0.0.0.0.6.0.0.0.0.0.0.0.0.0.0.0.3.0.0.2.0.0.1.0.0.0.ip6.arpa",
        "1-2-3-0-0-6-0-0.ipv6-literal.net"
    };
    parseHost(hostFormats);
}

```

Output:

1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0

```

1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0
1:2:3:0:0:6:0:0

```

Subnet strings are supported as well, using CIDR prefix notation or characters indicating range ('-' for a specific range or '*' for full range segments).

For instance, the subnet ffff::/104 can be represented many ways:

```

static void parse(String formats[]) {
    for(String format : formats) {
        System.out.println(new IPAddressString(format).getAddress().toPrefixedEquivalent());
    }
}

static void parseHost(String formats[]) {
    for(String format : formats) {
        System.out.println(new HostName(format).getAddress().toPrefixedEquivalent());
    }
}

public static void main(String[] args) {
    String prefixedFormats[] = {
        "ffff::/104",
        "ffff:0:0:0:0:0:0:0/104",
        "ffff:0000:0000:0000:0000:0000:0000:0000/104",
        "ffff::/104",
        "ffff::0.0.0.0/104",
        "=q{+M|w0(0e05^EGP660/104"
    };
    String rangeFormats[] = {
        "ffff:0:0:0:0:0:0-ff:*",
        "ffff::0-ff:*",
        "ffff::0-ff:*",
        "0xfffff000000000000000000000000000-0xfffff0000000000000000000000000fffff"
    };
    parse(prefixedFormats);
    parse(rangeFormats);
    String hostFormats[] = {
        "[ffff::]/104",
        "[ffff:0:0:0:0:0:0:0]/104",
        "[ffff:0000:0000:0000:0000:0000:0000:0000]/104",
        "[ffff::]/104",
        "[ffff::0.0.0.0]/104",
        "[=q{+M|w0(0e05^EGP660]/104",

        "[ffff:0:0:0:0:0:0-ff:]",
        "[ffff::0-ff:]",
        "[ffff::0-ff:]",
    };
}

```

```

        "[0xffff0000000000000000000000000000-0xffff00000000000000000000ffff]",
        "*. *.*.*.*.*.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.f.f.f.f.ip6.arpa",
        "ffff-0-0-0-0-0-0-0.ipv6-literal.net/104"
    };
    parseHost(hostFormats);
}

```

Output:

[illegible]

Delimited Segments

The subnet formats allow you to specify ranges of values. However, if you wish to parse addresses in which values are delimited, then you can use the methods `parseDelimitedSegments(String)` and `countDelimitedAddresses(String)` in `IPAddressString`. `parseDelimitedSegments` will provide an iterator to traverse through the individual addresses.

For example, given "1.2.3.4.5.6" `parseDelimitedSegments` will iterate through "1.3.4.6", "1.3.5.6", "2.3.4.6" and "2.3.5.6". You can then construct `IPAddressString` instances from those individual strings for parsing.

Address or Host Name Validation Options

Validation options allow you to restrict the allowed formats, whether you wish to support just IPv4 or IPv6, or whether you wish to support just single addresses, or whether you wish to allow different variants.

For IP addresses, you have the class `IPAddressStringParameters` which can be passed into `IPAddressString`, and for host names you have the class `HostNameParameters` which can be passed into `HostName`.

You constrict validation options using the appropriate builder classes that are nested classes in the options classes. Here is an example constructing host options along with nested address options within:

```

HostNameParameters HOST_OPTIONS_EXAMPLE = new HostNameParameters.Builder().
    allowEmpty(false).
    setEmptyAsLoopback(false).
    setNormalizeToLowercase(true).
    allowBracketedIPv6(true).
    allowBracketedIPv4(true).getAddressOptionsBuilder().
        allowPrefix(true).
        allowMask(true).
        setRangeOptions(RangeParameters.WILDCARD_AND_RANGE).
        allow_inet_aton(true).
        allowEmpty(false).
        setEmptyAsLoopback(false).
        allowAll(false).
        allowPrefixOnly(false).
    getIPv4AddressParametersBuilder().
        allowPrefixLengthLeadingZeros(true).
        allowPrefixesBeyondAddressSize(false).
        allowWildcardedSeparator(true).
    getParentBuilder().getParentBuilder().toParams();

```

The default options used by the library are permissive and not restrictive.

Host Name or Address with Port

For an address or host with port, use `HostName`. IPv6 addresses with ports must appear as `[ipv6Address]:port` to resolve the ambiguity of the colon separator, consistent with RFC 2732, 3986, 4038 and other RFCs.

```

HostName hostName = new HostName("[::1]:80");
System.out.println("host: " + hostName.getHost() + " address: " + hostName.getAddress() + " port: " + hostName.getPort());
hostName = new HostName("localhost:80");
System.out.println("host: " + hostName.getHost() + " port: " + hostName.getPort());
hostName = new HostName("127.0.0.1:80");
System.out.println("host: " + hostName.getHost() + " address: " + hostName.getAddress() + " port: " + hostName.getPort());

```

Output:

```

host: 0:0:0:0:0:0:0:1 address: 0:0:0:0:0:0:0:1 port: 80
host: localhost port: 80
host: 127.0.0.1 address: 127.0.0.1 port: 80

```

IP Version Determination and IPv4/v6 Conversion

With an `IPAddress` or `IPAddressString` object, you can check the version with `isIPv4()` and `isIPv6()` and get the appropriate subclass with `toIPv4()` or `toIPv6()`. You can also make use of `isIPv4Convertible()` and `isIPv6Convertible()` to do further conversions if the address is IPv4 mapped, or you can use your own determination (IPv4 translated, 6to4, 6over4, IPv4 compatible, etc) of how to convert back and forth using `IPAddressConverter`.

```

IPAddressString str = new IPAddressString("::ffff:1.2.3.4");
if(str.isIPv6()) {
    IPv6Address ipv6Address = str.getAddress().toIPv6();
    System.out.println(ipv6Address.toMixedString());
    if(ipv6Address.isIPv4Convertible()) {
        IPv4Address ipv4Address = ipv6Address.toIPv4();
        System.out.println(ipv4Address.toNormalizedString());
    }
}

```

Output:

```
::ffff:1.2.3.4
1.2.3.4
```

Once you have an `IPAddress` instance, there are methods to convert to bytes, to sections, to subnets or network prefixes, to masks, to `java.net.InetAddress`, to different textual representations, and so on.

Prefixed Addresses and `to.getHostAddress()`

This library will parse CIDR prefix IP addresses such as `10.1.2.3/24`.

When you have a prefixed address, like `10.1.2.3/24` which has the prefix length of 24, this notation may typically denote the single, distinct interface address `10.1.2.3` with the first 24 bits indicating the IPv4 block, or the the IPv4 block of 255 addresses which includes `10.1.2.3` (although typically the block would be denoted `10.1.2.0/24` with zeros outside the length of the prefix).

In this library `10.1.2.3/24` represents the IPv4 block (subnet) rather than just the single address `10.1.2.3` paired with its prefix length of 24. This avoids the inconsistency where `10.1.2.3/24` would be a single address, while the similar address with host portion zeroed, `10.1.2.0/24`, would denote a set of 255 addresses, a heuristic commonly used by network administrators. In this library, any prefixed address denotes the address block.

However, there are cases when you'd want to extract the original individual address paired with the prefix. For that there are the methods `getHostAddress` or `toHostAddress` which will provide the original host address `10.1.2.3` when parsing `10.1.2.3/24`. You can also extract only the prefix length by calling `getNetworkPrefixLength()` on the `IPAddressString` instance.

```
IPAddressString ipAddressString = new IPAddressString("10.1.2.3/24");
IPAddress address = ipAddressString.getAddress();
IPAddress hostAddress = ipAddressString.getHostAddress();
Integer prefixLength = ipAddressString.getNetworkPrefixLength();
System.out.println(address);
System.out.println(address.toCanonicalWildcardString());
System.out.println(hostAddress);
System.out.println(prefixLength);
```

Output:

```
10.1.2.0/24
10.1.2.*
10.1.2.3
24
```

In this library, `10.1.2.3/24` is equivalent to `10.1.2.*` when it comes to equality or comparison. The prefix is stored in the address, so the former will be aware that it is "prefixed", printing strings that incorporate the prefix and providing the network section upon calls to `getNetworkSection()` in `IPAddress`. If you have the latter address with no prefix length, you can convert to the former using the method `toPrefixedEquivalent()` in `IPAddress`.

Parse Non-Segmented Addresses – Hex, Octal, IPv6 Base 85

Typically, the segments or other punctuation identify a string as a host name an IPv4 address or an IPv6 address. The parser will also parse single segment values or a range of single segment values.

In such cases, ambiguity between IPv4 and IPv6 is resolved by the number of digits in the string, as there is no segment separator. The number of digits is 32 for IPv6 hexadecimal, 20 for IPv6 base 85 (see RFC 1924), and 11 or less for IPv4, which can be octal, hexadecimal, or decimal. For IPv4, digits are presumed decimal unless preceded by `0x` for hexadecimal or `0` for octal, as is consistent with the `inet_aton` routine. For IPv6, 32 digits are considered hexadecimal and the preceding `0x` is optional.

```
IPAddressString ipAddressString = new IPAddressString("4)+k&C#VzJ4br>0wv%Yp");//base 85
```


Address Sections

Addresses can be broken up into sections, and reconstituted from sections. A section is a series of segments. You can get the section for the full address by calling `getSection()`, or you can get subsections by calling one of the variants, either `getSection(int)` or `getSection(int, int)`, which return a subsection spanning the given indices. You can also get the segments in an address by calling `getSegment(int)` or one of the variants of `getSegments`, either on the address or on a section of the address.

You can also reconstitute an address from a section or array of segments using the appropriate address constructor, if your section or array of segments has the correct number of segments for the address type.

Host and Network Sections of IP Address

Use `getHostSection()` and `getNetworkSection()` to get the host and network sections of an IP address as indicated by prefix length.

```
IPAddress address = new IPAddressString("1.2.3.4").getAddress();
IPAddressSection network = address.getNetworkSection(16, true);
IPAddressSection host = address.getHostSection(16);
System.out.println(network.toCanonicalString());
System.out.println(host.toCanonicalString());
```

Output:

1.2/16

3.4

Once you have a section of an address, most of the same methods are available as those available with addresses themselves.

IP Address Operations

There are various methods for obtaining masking, subnets, and so on.

The methods `mask`, `maskNetwork`, `bitwiseOr`, `bitwiseOrNetwork`, `removePrefixLength`, `adjustPrefixBySegment`, `adjustPrefixLength`, `applyPrefixLength`, and `setPrefixLength` allow you to apply and adjust prefix lengths and masks to addresses and subnets. When applying an operation to a subnet, the operation is applied to every member of the subnet, so the result must something be representable with contiguous ranges, or you will get `AddressTypeException`. For instance, masking the subnet block of 255 addresses 0.0.0.0/24 with 0.0.0.128 results in the two addresses 0.0.0.0 and 0.0.0.128 which is not contiguous. However, this will not happen when using standard masking and subnetting techniques typical with IPv4 and IPv6 routing.

The following code demonstrates how to get the lowest address in a prefixed subnet using any one of three methods: `getLowest`, `mask`, or `removePrefixLength`.

```
String addr = "1.2.3.4";
IPAddress address = new IPAddressString(addr).getAddress();
int prefixLength = 16;
IPAddress maskWithPrefixLength = new IPAddressString("/") + prefixLength).
    getAddress(address.getIPVersion());

IPAddress mask = address.getNetwork().getNetworkMask(16, false);
System.out.println("mask with prefix length " + maskWithPrefixLength);
System.out.println("mask " + mask);

//mask
IPAddress maskedAddress = address.mask(mask);
System.out.println("address " + address + " masked " + maskedAddress);
IPAddress subnet = address.applyPrefixLength(prefixLength);
IPAddress maskedSubnet = subnet.mask(mask);
```

```

System.out.println("subnet " + subnet + " masked " + maskedSubnet);

//getLower
IPAddress lowestAddressInSubnet = subnet.getLower();
System.out.println("lowest in subnet " + lowestAddressInSubnet);
System.out.println(maskedAddress.equals(maskedSubnet));
System.out.println(maskedAddress.equals(lowestAddressInSubnet));

//removePrefixLength
IPAddress prefixRemoved = subnet.removePrefixLength(true);
System.out.println("prefix removed " + prefixRemoved);

```

Output:

```

mask with prefix length 255.255.0.0/16
mask 255.255.0.0
address 1.2.3.4 masked 1.2.0.0
subnet 1.2.0.0/16 masked 1.2.0.0
lowest in subnet 1.2.0.0
true
true
prefix removed 1.2.0.0

```

Polymorphism

Simply change the string "1.2.3.4" in the code above to an IPv6 address like "a:ffff:b:c:d::f" and the code all works the same.

Output:

```

mask with prefix length ffff::/16
mask ffff::
address a:ffff:b:c:d::f masked a::
subnet a::/16 masked a::
lowest in subnet a::
true
true

```

Handling Subnets and Network Prefixes

There are various additional methods for managing networks not listed above, such as:

- `toPrefixedEquivalent`: Converts an address with wildcards to an equivalent address with a prefix length. The resulting address will span the same range of values as the original.
- `toMinPrefixedEquivalent`: Converts an address to an equivalent address with the smallest possible prefix length. The resulting address will span the same range of values as the original.
- `getMaskPrefixLength`: Will return the prefix length for a network or host mask, or null otherwise
- `append`, `prepend`, `replace`: Add or replace segments in sections. To remove, use a variant of `getSection`
- `adjustPrefixBySegment`: Will create a larger network or smaller subnet address by increasing or decreasing the prefix length to the next segment boundary. If prefix length is increased, the additional prefix bits can be either zero or the full range of possible values.
- `removePrefixLength`, `adjustPrefixLength`, `applyPrefixLength`, `setPrefixLength`: Add, remove or adjust prefix lengths by the indicated values. There are variants to control whether extended prefix lengths have zeros for the added bits, or whether the bits remain full range. The default behavior when extending a prefix length is to insert zeros in the bits added to the prefix. The difference between `applyPrefixLength` and `setPrefixLength` is that the former does not shorten a prefix length; `applyPrefixLength` will not decrease the size of a subnet.

- `mask`, `maskNetwork`, `bitwiseOr`, `bitwiseOrNetwork`: apply masks to subnets and addresses. The network variants allow you to mask just the network portion and apply a prefix length, while the others remove any existing prefix length.
- `reverseBits`, `reverseBytes`, `reverseBytesPerSegment`, `reverseSegments`: useful for handling endianness (network byte order sometimes requires bytes be reversed), or DNS lookup, or other reasons for reversing bits and bytes.
- `subtract`: Computes the subnet difference, the set of addresses in the subnet but not in the argument subnet. Subtracts a given subnet from the receiver subnet, returning an array of subnets for the result.

Subnetting

Subnetting can be accomplished using various address manipulation methods. Given a prefixed IP address, you can extend the prefix length and insert bits for an extended prefix and new subnet of the previous address block, as shown in the following example.

```
int originalPrefix = 18, adjustment = 4;
IPAddress address = new IPAddressString("207.0.64.0").getAddress();
IPAddress address2 = address.setPrefixLength(originalPrefix);
IPAddress prefixExtension = new IPAddressString("0.0.4.0").getAddress();

IPAddress subnet1 = address2.adjustPrefixLength(adjustment); //extend the prefix length
System.out.println(subnet1);

IPAddress subnet2 = subnet1.bitwiseOrNetwork(prefixExtension, originalPrefix + adjustment); //adjust the extended prefix
System.out.println(subnet2);
```

Output:

```
207.0.64.0/18
207.0.64.0/22
207.0.68.0/22
```

Here is the same subnetting operation using segment replacement:

```
IPv4Address address = new IPAddressString("207.0.64.0/18").getAddress().toIPv4();
IPv4AddressSection replacementSection = new IPAddressString("0.0.68.0/22").getAddress().toIPv4().getSection(2);
IPAddress subnet = new IPv4Address(address.getSection().replace(replacementSection, 2));
System.out.println(subnet);
```

Output:

```
207.0.68.0/22
```

Parse String Representation of MAC Address

Conversion is like IP address. `MACAddressString` is used to convert. You can use one of `getAddress` or `toAddress`, the difference being whether parsing errors are handled by exception or not.

```
MACAddress address = new MACAddressString("01:02:03:04:0a:0b").getAddress();
if(address != null) {
    //use address here
}
```

or

```
try {
```

```

        MACAddress address = new MACAddressString ("01:02:03:04:0a:0b").toAddress();
        //use address here
    } catch (AddressStringException e) {
        String msg = e.getMessage();//detailed message indicating issue
    }
}

```

Various Formats of MAC Addresses

MAC Addresses are expected to be in hexadecimal. However, there are a wide variety of accepted formats for MAC addresses:

aa:bb:cc:dd:ee:ff

aa-bb-cc-dd-ee-ff

aa bb cc dd ee ff

aabb.ccdd.eeff

aabbccddeeff

aabbcc-ddeeff

For the non-segmented format (aabbccddeeff), all 12 digits are required to avoid ambiguity. Additionally, MAC addresses can be either 48 or 64 bits, and so for each such format there is the 64-bit equivalent:

aa:bb:cc:dd:ee:ff:11:22

aa-bb-cc-dd-ee-ff-11-22

aa bb cc dd ee ff 11 22

aabb.ccdd.eeff.1122

aabbccddeeff1122

aabbcc-ddeeff1122

For the non-segmented 64-bit format (aabbccddeeff1122), all 16 digits are required to avoid ambiguity.

As with IP addresses, you can specify ranges using ‘*’ and ‘-’ like aa-bb:*:*:cc:dd:ee. The range character for addresses that use the dash ‘-’ character as a separator is ‘|’, like aa|bb-*-*:cc-dd-ee.

Format Examples

For instance, the address 0a:0b:0c:0d:0e:0f can be represented many ways:

```

static void parseMAC(String formats[]) {
    for(String format : formats) {
        System.out.println(new MACAddressString(format).getAddress());
    }
}

public static void main(String[] args) {
    String formats[] = {
        "a:b:c:d:e:f",
        "0a:0b:0c:0d:0e:0f",
    }
}

```

```

        "a:b:c:d:e:f",
        "0a-0b-0c-0d-0e-0f",
        "0a0b0c-0d0e0f",
        "0a0b.0c0d.0e0f",
        "0a 0b 0c 0d 0e 0f",
        "0a0b0c0d0e0f"
    };
    parseMAC(formats);
}

```

Output:

```

0a:0b:0c:0d:0e:0f
0a:0b:0c:0d:0e:0f
0a:0b:0c:0d:0e:0f
0a:0b:0c:0d:0e:0f
0a:0b:0c:0d:0e:0f
0a:0b:0c:0d:0e:0f
0a:0b:0c:0d:0e:0f
0a:0b:0c:0d:0e:0f

```

Ranges can be specified in the same way as for IP addresses ('-' for a specific range or '*' for full range segments).

For instance, the address range ff:0f:aa-ff:00-ff:00-ff:00-ff can be represented many ways:

```

static void parseMAC(String formats[]) {
    for(String format : formats) {
        System.out.println(new MACAddressString(format).getAddress());
    }
}

public static void main(String[] args) {
    String formats[] = {
        "ff:f:aa-ff:00-ff:00-ff:00-ff",
        "ff:f:aa-ff:*.:*.*",
        "ff:0f:aa-ff:*.:*.*",
        "ff-0f-aa|ff-*.~.*",
        "ff0faa|ff0fff-.*",
        "ff0f.aa00-ffff.*",
        "ff 0f aa-ff * * *",
        "ff0faa000000-ff0fffffffffff"
    };
    parseMAC(formats);
}

```

Output:

```

ff:0f:aa-ff:*.:*.*
ff:0f:aa-ff:*.:*.*
ff:0f:aa-ff:*.:*.*
ff:0f:aa-ff:*.:*.*
ff:0f:aa-ff:*.:*.*
ff:0f:aa-ff:*.:*.*
ff:0f:aa-ff:*.:*.*
ff:0f:aa-ff:*.:*.*

```

Delimited Segments

The range formats allow you to specify ranges of values. However, if you wish to parse addresses in which values are delimited, then you can use the methods `parseDelimitedSegments(String)` and `countDelimitedAddresses(String)` in `MACAddressString`. `parseDelimitedSegments` will provide an iterator to traverse through the individual addresses.

For example, given "1,2:3:4,5:6:7:8", `countDelimitedAddresses` will return 4 for the possible combinations: "1:3:4:6:7:8", "1:3:5:6:7:8", "2:3:4:6:7:8" and "2:3:5:6:7:8". With each string obtained from `parseDelimitedSegments` you can construct a `MACAddressString` instance.

MAC Address Validation Options

Validation options allow you to restrict the allowed formats.

You have the class `MACAddressStringParameters` which can be passed into `MACAddressString`

You constrict validation options using the appropriate builder classes that are nested classes in the options classes. Here is an example:

```
MACAddressStringParameters MAC_ADDRESS_OPTIONS_EXAMPLE = new MACAddressStringParameters.Builder().
    allowEmpty(false).
    allowAll(false).
    getFormatBuilder().
        setRangeOptions(RangeParameters.NO_RANGE).
        allowLeadingZeros(true).
        allowUnlimitedLeadingZeros(false).
        allowWildcardedSeparator(false).
        allowShortSegments(true).
    getParentBuilder().
    toParams();
```

The default options used by the library are permissive and not restrictive.

MAC Address Prefix Lengths

Prefix length in this library is defined general as the bit-length of the portion of the address that is not specific to an individual address but common amongst a group of addresses. For IP addresses, this is explicitly defined as part of the address using the '/' character. For IP addresses, the prefix is potentially variable and depends on how subnets have been allocated.

MAC Addresses don't have the exact same concept of prefix length as IP addresses. But concept of prefix can be applied in a more implicit sense. For instance, a MAC address is typically divided into an OUI (Organizationally Unique Identifier) and ODI (Organizational Defined Identifier), so you might consider the OUI bits as the prefix. There are other ways of assigning MAC address blocks as well, such as IAB (individual address block), or MA-S/MA-M/MA-L (MAC Address block small, medium, and large), in which a certain number of higher bits are provided as an identifier to organizations from which they can create various extended identifiers using the lower bits. There is generally a pre-defined set of high bits that can be considered a prefix. This prefix is not variable and was typically assigned by the IEEE. However, there is no explicit way to represent a MAC address with an associated prefix, so the prefix is implicit to the address when it is created.

Within this library, the prefix for a MAC address is defined as the largest number of high bits for which an address represents all addresses with the same set of higher bits.

For instance, the prefix length of `aa:*.**:**` is 8 because the address represents all addresses that start with the same 8 bits "aa". The prefix length of `aa:*.cc:***` is 24 because the address represents all addresses that start with the same 16 bits `aa:*.cc`. The address `aa:bb:cc:dd:ee:ff` does not have a prefix or prefix length as it represents just a single address.

This means that on the MAC side, the prefix length is implicit and based upon the address, while on the IP address side, the prefix length is explicitly defined.

MAC Address Operations

Many of the same address operations available for IP addresses are available for MAC addresses, including the prefix operations (although prefixes have somewhat different meaning for MAC as described in the previous section of this document), the section and segment access methods, iterators, containment, and reversal of bits, bytes and segments.

The reverse operations are useful for for "MSB format", "IBM format", "Token-Ring format", and "non-canonical form", where the bits are reversed in each byte of a MAC address.

IPv6 – MAC Address Integration

There is a standardized procedure for converting MAC addresses to IPv6 given a IPv6 64 bit prefix, as described in [IETF RFC 2464](#), [RFC 3513](#), and [RFC 4944](#). It details how to combine a 48 bit MAC address with a 64 bit IPv6 network prefix to produce a 128 bit IPv6 address. This is done by first constructing a 64-bit extended unique IPv6 interface identifier (EUI-64) from the MAC address. This library has implemented the same MAC / IPv6 integration in multiple ways.

Starting with a MAC address or section and with the IPv6 prefix, you can construct an IPv6 address.

```
public IPv6Address(IPv6Address prefix, MACAddress eui)
public IPv6Address(IPv6AddressSection section, MACAddress eui)
public IPv6Address(IPv6AddressSection section, MACAddressSection eui)
public IPv6Address(IPv6AddressSection section, MACAddressSection eui, CharSequence zone)
```

There are equivalent methods in the class `MACAddress` for producing the link local address which has a pre-defined prefix, or for producing the host (interface identifier) address section of an IPv6 address.

```
public IPv6Address toLinkLocalIPv6()
public IPv6AddressSection toEUI64IPv6()
```

There is a similar method in `MACAddressSection`

```
public IPv6AddressSection toEUI64IPv6()
```

A MAC address is either 48 or 64 bits. To be converted to IPv6, the 48 bit address has segments inserted (two 1 byte MAC segments with value 0xfffe) to extend the address to 64 bits. For a 64 bit MAC address to be convertible, those same two segments must match the expected value of 0xfffe. The following methods in `MACAddress` check will 64 bit addresses to ensure that are compatible with IPv6 by checking that the value of those two segments matches 0xfffe. Note that the `asMAC` argument allows you to extend using 0xffff rather than 0xfffe which is another manner by which a 48 bit MAC address can be extended to 64 bits. However, for purposes of extending to an IPv6 address the argument should be `false` so that the EUI-64 format is used.

```
public boolean isEUI64(boolean asMAC)
public MACAddress toEUI64(boolean asMAC)
```

There are similar methods in `MACAddressSection`

```
public boolean isEUI64(boolean asMAC)
public boolean isEUI64(boolean asMAC, boolean partial)
public MACAddressSection toEUI64(boolean asMAC)
```

Given an existing EUI-64 section you can use the `prepend` and `append` methods to create a full IPv6 address section containing all segments:

```
public IPv6AddressSection prepend(IPv6AddressSection other)
```



```
public IPv6AddressSection append(IPv6AddressSection other)
```

and from there you just construct the address:

```
public IPv6Address(IPv6AddressSection section)
```

To go the reverse direction IPv6 to MAC, there is a method in `IPv6Address` to produce a MAC address:

```
public MACAddress toEUI(boolean extended)
```

and another in `IPv6AddressSection` that uses whatever part of the interface identifier is included in the section to produce a MAC address section:

```
public MACAddressSection toEUI(boolean extended)
```

For instance, if the IPV6 address section is the 8 bytes corresponding to the network prefix of an IPV6 address section, then the resulting MAC address section will be 0 bytes. If the IPV6 address section is the 8 bytes corresponding to the interface identifier, and that identifier has the required 0xfffe values in the 5th and 6th bytes, then the MAC address section will be the full 8 bytes of an EUI-64 MAC address.

The following code is an example of constructing IPv6 addresses from a MAC address:

```
public static void main(String args[]) {
    MACAddressString macStr = new MACAddressString("aa:bb:cc:dd:ee:ff");
    MACAddress macAddress = macStr.getAddress();
    IPv6Address linkLocal = macAddress.toLinkLocalIPv6();
    System.out.println(linkLocal);

    IPAddressString ipv6Str = new IPAddressString("1111:2222:3333:4444::/64");
    IPv6Address ipv6Address = ipv6Str.getAddress().toIPv6();
    IPv6Address macIpv6 = new IPv6Address(ipv6Address, macAddress);
    System.out.println(macIpv6);
}
```

Output:

```
fe80:0:0:0:a8bb:ccff:fedd:eeff
1111:2222:3333:4444:a8bb:ccff:fedd:eeff
```

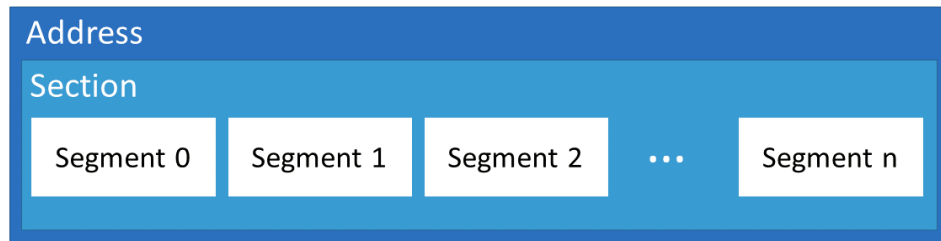
Address Framework

Much like there is a Java collections framework, there is an address framework to the `IPAddress` library. It is a unified set of inter-related interfaces, abstract implementations, and algorithms for all addresses and address components. It allows you to manipulate these items independently of implementation details, and provides standard interfaces to addresses and address components for code reuse and polymorphism. It represents the common structure of addresses, sections, segments, and so on.

You might wish to manipulate address components of different shapes and sizes transparently, or you may wish to manipulate different types of addresses or different address versions transparently. One element of the interface is the ability to convert any address component to bytes, whether division, segment, section, or address.

There is a hierarchy for the standard Address and Address Component data structures, which are addresses, sections of addresses, and segments of equal byte size inside those address sections.

Segments are equal length, each an exact number of bytes



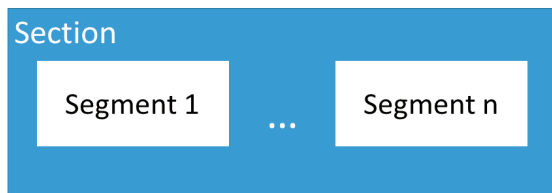
Examples

IPv4: 0.0.255.255

IPv6: fe80:0:0:0:204:61ff:aaab:bbbb

MAC: aa-bb-cc-dd-ee-ff

Sections have a variable number of segments



There is a more diversified hierarchy for non-standard address structures, in which addresses or address sections might be divided into divisions of unequal length, or of non-integral bytesize.

Addresses can be split into sections or reconstituted into division groupings.
Divisions need not be equal length, nor start and end on byte boundaries

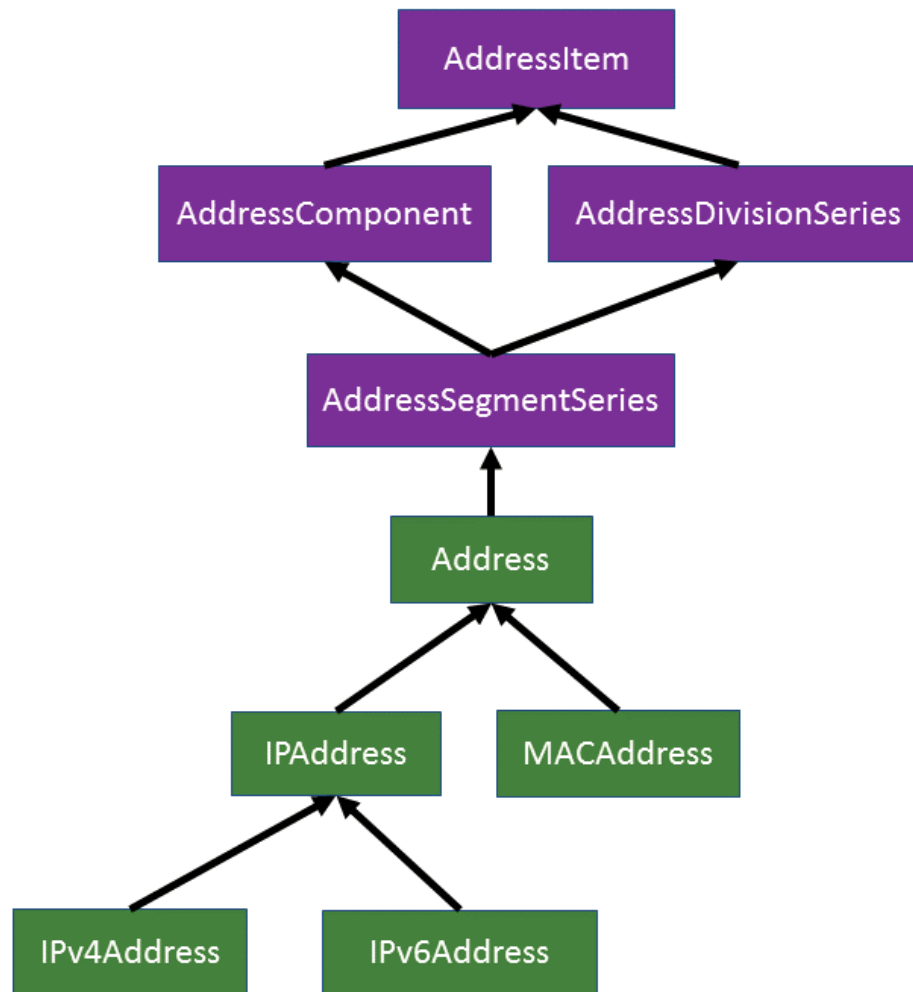


Examples

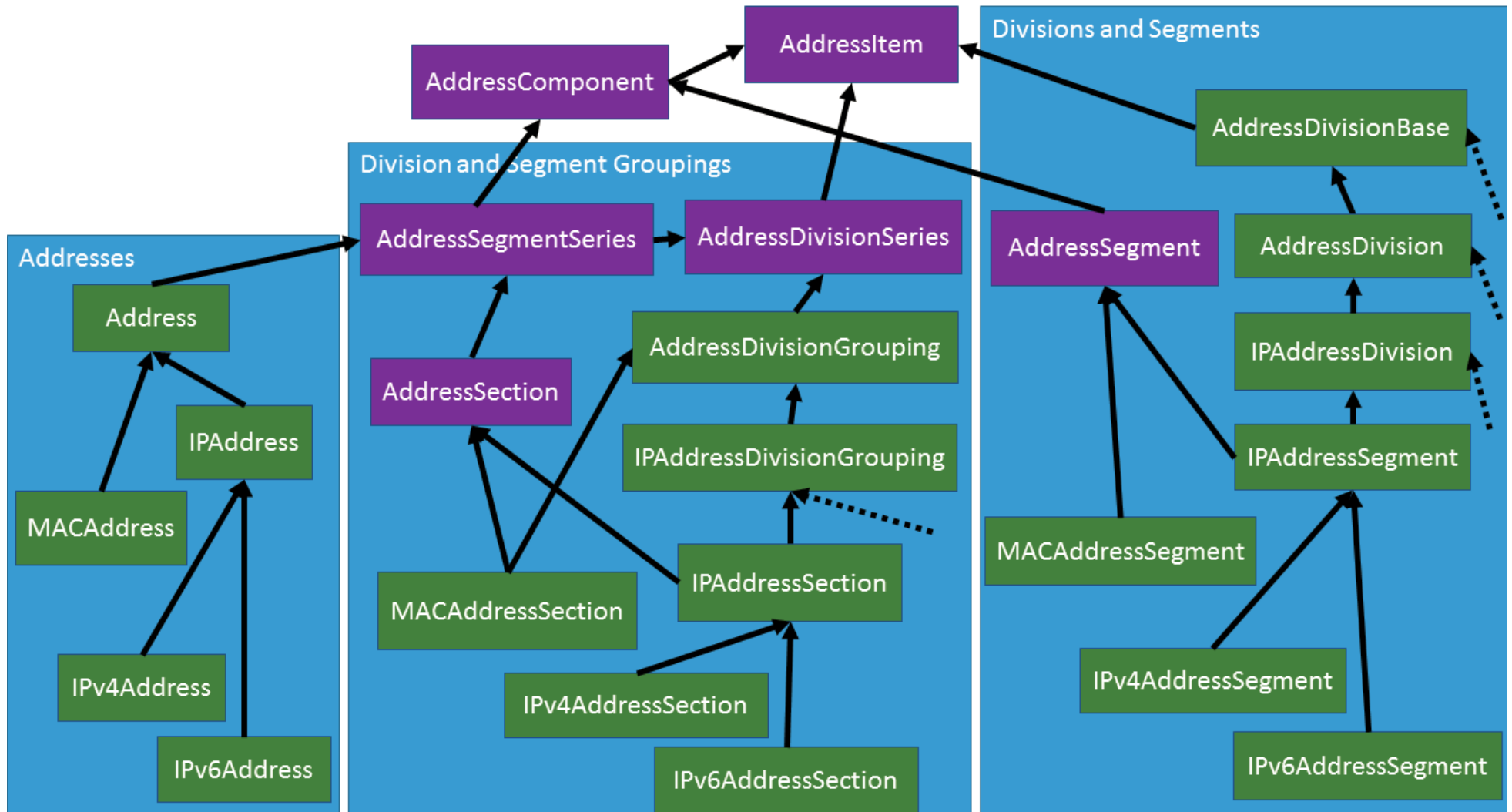
IPv4: 0.0.65535

IPv6 with IPv4 embedded: fe80:0:0:0:204:61ff:254.157.241.86

The address hierarchy of interfaces (purple) and classes (green) is shown:



The full class hierarchy showing addresses, sections, division groupings, segments and divisions is shown here, separated into the three primary categories shown above. The dashed lines indicate there are a few less-prominent classes in the library not shown in the diagram.



Conversion to String Representation of Address

`AddressSegmentSeries` objects (such as `Address` and `AddressSection`) have methods that produce strings in various formats: `toCanonicalString`, `toNormalizedString`, `toCompressedString`, and `toHexString`.

```
public static void main(String[] args) {
    printStrings(new MACAddressString("a:bb:c:dd:e:ff").getAddress());
    printStrings(new MACAddressString("a:bb:c:*").getAddress());
    printStrings(new IPAddressString("a:bb:c::/64").getAddress());
    printStrings(new IPAddressString("1.2.3.4").getAddress());
}
```

```

    printStrings(new IPAddressString("1.2.3.4/16").getAddress());
    printStrings(new IPAddressString("a:bb:c::dd:e:ff").getAddress());
}
static void printStrings(AddressSegmentSeries series) {
    System.out.println(series.toCanonicalString());
    System.out.println(series.toNormalizedString());
    System.out.println(series.toCompressedString());
    System.out.println(series.toHexString(true));
    System.out.print("lower: " + series.getLower() + " bytes:");
    System.out.println(Arrays.toString(series.getBytes()));
    System.out.print("upper: " + series.getUpper() + " bytes:");
    System.out.println(Arrays.toString(series.getUpperBytes()));
    System.out.println();
}

```

Output:

```

0a-bb-0c-dd-0e-ff
0a:bb:0c:dd:0e:ff
a:bb:c:dd:e:ff
0x0abb0cdd0eff
lower: 0a:bb:0c:dd:0e:ff bytes:[10, -69, 12, -35, 14, -1]
upper: 0a:bb:0c:dd:0e:ff bytes:[10, -69, 12, -35, 14, -1]

0a-bb-0c-**-**
0a:bb:0c:***:
a:bb:c:***:
0x0abb0c000000-0x0abb0cffffff
lower: 0a:bb:0c:00:00:00 bytes:[10, -69, 12, 0, 0, 0]
upper: 0a:bb:0c:ff:ff:ff bytes:[10, -69, 12, -1, -1, -1]

a:bb:c::/64
a:bb:c:0:0:0:0/64
a:bb:c::/64
0x000a00bb000c0000000000000000-0x000a00bb000c0000ffffff
lower: a:bb:c:0:0:0:0 bytes:[0, 10, 0, -69, 0, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0]
upper: a:bb:c:0:ffff:ffff:ffff bytes:[0, 10, 0, -69, 0, 12, 0, 0, -1, -1, -1, -1, -1, -1, -1]

1.2.3.4
1.2.3.4
1.2.3.4
0x01020304
lower: 1.2.3.4 bytes:[1, 2, 3, 4]
upper: 1.2.3.4 bytes:[1, 2, 3, 4]

1.2.0.0/16
1.2.0.0/16
1.2.0.0/16
0x01020000-0x0102ffff
lower: 1.2.0.0 bytes:[1, 2, 0, 0]
upper: 1.2.255.255 bytes:[1, 2, -1, -1]

a:bb:c::dd:e:ff
a:bb:c:0:0:dd:e:ff
a:bb:c::dd:e:ff
0x000a00bb000c0000000000dd000e00ff
lower: a:bb:c:0:0:dd:e:ff bytes:[0, 10, 0, -69, 0, 12, 0, 0, 0, 0, 0, -35, 0, 14, 0, -1]
upper: a:bb:c:0:0:dd:e:ff bytes:[0, 10, 0, -69, 0, 12, 0, 0, 0, 0, 0, -35, 0, 14, 0, -1]

```

The `IPAddress` and `IPAddressSection` classes and their version-specific subclasses have additional methods to produce specific strings representing the address:

```

public static void main(String[] args) {
    IPAddress address = new IPAddressString("a:b:c::e:f").getAddress();
    print(address);
    address = new IPAddressString("a:b:c::").getAddress();
    print(address);
    address = new IPAddressString("a:b:c*::").getAddress();
    print(address);
}

static void print(IPAddress address) {
    System.out.println(address.toCanonicalString());
    System.out.println(address.toFullString());
    System.out.println(address.toNormalizedString());
    System.out.println(address.toSQLWildcardString());
    System.out.println(address.toSubnetString());
    try {
        if(address.isIPv6()) {
            System.out.println(address.toIPv6().toMixedString());
            System.out.println(address.toIPv6().toBase85String()); //throws for a:b:c*:: because the address cannot be represented as a range of two
single values
        }
        System.out.println(address.toBinaryString()); //throws for a:b:c*:: because the address cannot be represented as a range of two single values
        System.out.println(address.toHexString(true)); //throws for a:b:c*:: because the address cannot be represented as a range of two single values
    } catch (AddressTypeException e) {}
    System.out.println();
}

```

Output:

[illegible]

```
000a:000b:000c:0000-ffff:0000:0000:0000:0000
a:b:c:*:0:0:0:0
a:b:c:%:0:0:0:0
a:b:c:*::
a:b:c:*::
```

IPv6 string methods

There are additional methods for `Ipv6`, the methods `toMixedString()` and `toBase85String()`

UNC Strings

The method `toUNCHostName()` produces the UNC IP-literal string.

```
IPAddressString ipAddressString = new IPAddressString("2001:db8::1");
IPAddress address = ipAddressString.getAddress();
System.out.println(address.toUNCHostName());

ipAddressString = new IPAddressString("1.2.3.4");
address = ipAddressString.getAddress();
System.out.println(address.toUNCHostName());
```

Output:

2001-db8-0-0-0-0-1.ipv6-literal.net
1.2.3.4

DNS Lookup Strings

The methods `toReverseDNSLookupString()` will produce a string for DNS lookup. If you wish to do a DNS lookup for a subnet rather than a full address, you can use `getNetworkSection()` to provide the network section you wish to lookup. You can specify the prefix length in either the string itself or the call to get the network section.

```
IPAddressString ipAddressString = new IPAddressString("2001:db8::1");
IPAddress address = ipAddressString.getAddress();
System.out.println(address.toReverseDNSLookupString());

IPAddressSection addressSection = address.getNetworkSection(64);
System.out.println(addressSection.toReverseDNSLookupString());

ipAddressString = new IPAddressString("2001:db8::1/64");
address = ipAddressString.getAddress();
System.out.println(address.toReverseDNSLookupString());

addressSection = address.getNetworkSection();
System.out.println(addressSection.toReverseDNSLookupString());
```

Output:

```
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.b.d.0.1.0.0.2.ip6.arpa  
0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa  
*,*,*,*,*,*,*,*,*,*,*.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa  
0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa
```


General String Methods

The methods `toCanonicalString` and `toCompressedString` are available for any address or address section.

The methods `toHexString` and `toNormalizedString` are available for any address component.

Avoiding Prefix Length Indicator in Strings

To choose to print wildcards '*' and range characters '-' as opposed to using prefix length, there are additional methods:

```
public static void main(String[] args) {
    IPAddress address = new IPAddressString("a:b:c::/64").getAddress();
    print(address);
    address = new IPAddressString("a:b:c:*>::/64").getAddress();
    print(address);
}

static void print(IPAddress address) {
    System.out.println(address.toCanonicalString());
    System.out.println(address.toCanonicalWildcardString());
    System.out.println(address.toFullString());
    System.out.println(address.toNormalizedString());
    System.out.println(address.toSQLWildcardString());
    System.out.println();
}
```

Output:

```
a:b:c::/64
a:b:c:0:*:*:*/64
000a:000b:000c:0000:0000:0000:0000:0000/64
a:b:c:0:0:0:0:0/64
a:b:c:0:%:%%:%%

a:b:c:*>::/64
a:b:c:*:*:*:*/64
000a:000b:000c:0000-ffff:0000:0000:0000:0000/64
a:b:c:*:0:0:0:0/64
a:b:c:%:%%:%%:%%
```

IP Version-dependent Strings

Some strings are version-dependent:

```
public static void main(String[] args) {
    //toSubnetString() prefers '*' for IPv4 and prefix for IPv6
    System.out.println(new IPAddressString("1.2.0.0/16").getAddress().toSubnetString());
    System.out.println(new IPAddressString("a:b::/64").getAddress().toSubnetString());

    //converts IPv4-mapped to a:b:c:d:e:f:1.2.3.4 notation
```

```

        System.out.println(new IPAddressString("::ffff:a:b").getAddress().toConvertedString());
    }

```

Output:

```

1.2.*.*
a:b::/64
::ffff:0.10.0.11

```

Collections of IP Address Strings

Alternatively, you can produce collections of strings:

```

public static void main(String[] args) {
    IPAddress address = new IPAddressString("a:b:c::e:f").getAddress();
    print(address);
}

private static void print(IPAddress address) {
    //print(address.toAllStrings()); produces many strings
    print(address.toStandardStrings());
    print(new String[] {""});
}

public static void print(String strings[]) {
    for(String str : strings) {
        System.out.println(str);
    }
}

```

Output:

```

a:b:c:0:0:0:0.14.0.15
a:b:c:0:0:0:000.014.000.015
000a:000b:000c:0000:0000:0000:0.14.0.15
000a:000b:000c:0000:0000:0000:000.014.000.015
A:B:C:0:0:0:0.14.0.15
A:B:C:0:0:0:000.014.000.015
000A:000B:000C:0000:0000:0000:0.14.0.15
000A:000B:000C:0000:0000:0000:000.014.000.015
a:b:c::0.14.0.15
a:b:c::000.014.000.015
000a:000b:000c::0.14.0.15
000a:000b:000c::000.014.000.015
A:B:C::0.14.0.15
A:B:C::000.014.000.015
000A:000B:000C::0.14.0.15
000A:000B:000C::000.014.000.015
a:b:c:0:0:0:e:f
000a:000b:000c:0000:0000:0000:000e:000f
A:B:C:0:0:0:E:F
000A:000B:000C:0000:0000:0000:000E:000F
a:b:c::e:f
000a:000b:000c::000e:000f
A:B:C::E:F

```

```
000A:000B:000C::000E:000F
```

The String collections can be customized with `toNormalizedString(StringOptions params)`

Note that string collections never have duplicate strings. The String collections can be customized with `toStrings(IPStringBuilderOptions options)`.

Searching Text of Databases for all Addresses in a Subnet

Suppose you wanted to search for all addresses from a subnet in a large amount of text data. For instance, suppose you wanted to search for all addresses in the text from the subnet a:b:0:0::/64. You can start a representation of just the network prefix section of the address, then you can get all such strings for that prefix.

```
IPAddressSection prefix = new IPAddressString("a:b::").getAddress().
    getNetworkSection(64, false);
String strings[] = prefix.toStandardStringCollection().toStrings();
for(String str : strings) {
    System.out.println(str);
}
```

Output:

```
a:b:0:0
000a:000b:0000:0000
A:B:0:0
000A:000B:0000:0000
a:b::
000a:000b::
A:B::
000A:000B::
```

If you need to be more stringent or less stringent about the address formats you wish to search, then you can use `toStringCollection(IPStringBuilderOptions options)` with an instance of `IPv6StringBuilderOptions`.

Searching for those strings will find the subnet addresses. However, you may get a few false positives, like "a:b::d:e:f:a:b". To eliminate the false positives, you can just emulate in Java the SQL code produced below for the SQL database search, using substrings constructed from the segment separators.

For a MySQL database search:

```
public static void main(String[] args) {
    IPAddressSection prefix = new IPAddressString("a:b::").
        getAddress().getNetworkSection(64, false);
    StringBuilder sql = new StringBuilder("Select rows from table where ");
    prefix.getStartsWithSQLClause(sql, "column1");
    System.out.println(sql);
}
```

Output:

```
Select rows from table where ((substring_index(column1,':',4) = 'a:b:0:0') OR ((substring_index(column1,':',3) = 'a:b:') AND (LENGTH (column1) -
LENGTH(REPLACE(column1, ':', '')) <= 6)))
```

For IPv4, another way to search for a subnet like 1.2.0.0/16 would be to do a SELECT with the following string

```
public static void main(String[] args) {
    String wildcardString = new IPAddressString("1.2.0.0/16").
        getAddress().toSQLWildcardString();
    System.out.println(wildcardString);
}
```

Output:

```
1.2.%.%
```

Then your SQL search string would be like:

```
Select rows from table where column1 like 1.2.%.%
```

Containment and Subnet Membership

To check whether an IP address is contained by a subnet:

```
IPAddress address = new IPAddressString("1.2.0.0/16").getAddress();
System.out.println(address.contains(new IPAddressString("1.2.3.4").getAddress()));
System.out.println(address.contains(new IPAddressString("1.2.3.0/24").getAddress()));
System.out.println(address.contains(new IPAddressString("1.2.3.0/25").getAddress()));
System.out.println(address.contains(new IPAddressString("1.1.0.0").getAddress()));
```

Output:

```
true
true
true
false
```

The contains method is not restricted to IP addresses or IP address prefixed addresses. There is a contains method for every Address or AddressSection.

There is also an assortment of iterators for addresses, sections, and segments which represent multiple values. There is an iterator(), getLower() method and getUpper() method for every address component.

DNS Resolution and URLs

If you have a string that can be a host or an address and you wish to resolve to an address, create a `HostName` and use `HostName.toResolvedAddress()`. If you wish to obtain a string representation to be part of a URL, use `HostName.toNormalizedString()`.

Sorting and Comparisons

Comparing and sorting can be useful for storing addresses in certain types of data structures. All of the core classes implement `java.lang.Comparable`. Different representations of the same address or subnet are considered equal. Different representations of the same set of addresses are considered equal. However, `HostName` instances, `IPAddressString` instances, and `IPAddress` instances are not equal even when representing the same address or subnet.

The library provides `IPAddressComparator` and some implementations for comparison purposes. `IPAddress` uses the subclass `CountComparator`. When comparing subnets, you can either emphasize the count of addresses, or you can emphasize the values of the lower or upper address represented by the subnet, and comparators are provided for those variations.

Networks and Caches

The library provides singleton network objects for each address type. The network objects have methods for obtaining masks and loopbacks. The networks also provide creator objects that create addresses, sections, and segments, and they perform caching of these address components for efficient memory usage and performance.

The `network()` methods in `IPv4Address`, `IPv6Address`, and `MACAddress` provide access to the respective network objects. Each network has an associated address creator object accessible from `getAddressCreator()` providing methods for address component creation that make use of the aforementioned caches. If you wished to use your own subclasses of the address classes, you could also use your own network object, and your own address creator object, by replacing the `network` field of the respective address class that you have subclassed, and optionally the `createAddressCreator` method in the network class. That would allow you to control which types are created within the IP address library.

The `IPAddressNetwork` class defines `IPAddressStringCache` and `HostNameCache`. These cache classes allow you to cache identifier strings. Note that identifier strings will cache their associated addresses, whether parsed or resolved. Also note that addresses cache their associated strings and various other objects. Therefore, these cache classes go a long way towards allowing you to avoid creating the same objects and running the same code frequently. These caches do quick lookups using either bytes or strings, which might be ideal for some applications that handle many addresses or host names.