Contents

Benefits of this Library

The library was intended to satisfy the following primary goals:

- **Parsing of all host name and ipv4/ipv6 address formats in common usage** plus some additional formats (see below or see javadoc for the IPAddressString class for the extensive list)
- **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
  - Hexadecimal values: 0x01020000-0x0102ffff
  - Octal values: 000100400000-000100577777
- **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 when parsing**, whether IPv4/6, or subnets, or inet_aton formats, and so on.
- **Produce all common address strings of different formats for a given IPv4 or IPv6 address and produce collections of such strings**
  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**
- **Produce all common MAC address strings of different formats**
- **Integration of MAC Address with IPv6 with standard conversions**
- **Integration of IPv4 Address with IPv6 through common address conversions**

- **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, produce address strings, mask addresses, etc. You can make use of the address framework which is agnostic towards address version.
- **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. For multi-threaded apps this is valuable.
- **Address modifications**, such as altering prefix lengths, masking, splitting into sections and segments, splitting into network and host sections, reconstituting from sections and segments
- **Address Operations**, such as getting the prefix block subnet for a prefixed address, iterating through subnets, iterating through prefix blocks, incrementing and decrementing addresses by integral values, reversing address bits for endianness or DNS lookup, subtracting subnets from other subnets, intersection of subnets, merging subnets, checking containment of addresses in subnets, listing subnets covering a span of addresses
- **Sorting and comparison** of host names, addresses, address strings and subnets
- **Integrate with the primitive java types and the standard java classes InetAddress, Inet6Address, Inet4Address, and BigInteger.**

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
- all the formats produced by netstat involving hosts/addresses with ports/services
- 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]:port, ::1:service, 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 or 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 address or subnet for that network prefix length, depending upon the configuration for prefix handling. 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 or 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**, as well as the sequential address class **IPAddressSeqRange** and its subclasses **IPv4AddressSeqRange** and **IPv6AddressSeqRange**. 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 either an address or host name, or you have something with a port or service name, 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

```
String str = "1.2.3.4";
try {
    IPAddress address = new IPAddressString(str).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::",
        "008JQWOV7O(=61h*;$LC",
        "0x00010002000300000000000600000000"
    };
    parse(formats);
    String hostFormats[] = {
                "[1:2:3:0:0:6::]",
        "[1:2:3:0:0:6:0:0]",
        "[1:2:3::6:0:0]",
        "[0001:0002:0003:0000:0000:0006:0000:0000]",
        "[1:2:3::6:0.0.0.0]",
        "[1:2:3:0:0:6::]",
        "[008JQWOV7O(=61h*;$LC]",
        "[0x00010002000300000000000600000000]",
        "0.0.0.0.0.0.0.0.6.0.0.0.0.0.0.0.0.0.0.0.3.0.0.0.2.0.0.0.1.0.0.0.ip6.arpa",
        "1-2-3-0-0-6-0-0.ipv6-literal.net"
    };
    parseHost(hostFormats);
}
```

Output:

1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::6:0:0
1:2:3::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().assignPrefixForSingleBlock());
    }
}

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

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(OeO5^EGP660/104"
         };
    String rangeFormats[] = {
        "ffff:0:0:0:0:0:0-ff:*",
        "ffff::0-ff:*",
        "ffff::0-ff:*",
        "0xffff00000000000000000000000000-0xffff0000000000000000000000ffffff"
         };
    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(OeO5^EGP660]/104",

                "[ffff:0:0:0:0:0:0-ff:*]",
        "[ffff::0-ff:*]",
        "[ffff::0-ff:*]",
        "[0xffff00000000000000000000000000-0xffff0000000000000000000000ffffff]",
        "*.*.*.*.*.*.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.f.f.f.f.ip6.arpa",
        "ffff-0-0-0-0-0-0-0.ipv6-literal.net/104"
    };
    parseHost(hostFormats);
}
```

Output:

ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104

ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104
ffff::/104

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. The method 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 or Service Name

For an address or host with port or service name, use HostName. IPv6 addresses with ports should appear as [ipv6Address]:port to resolve the ambiguity of the colon separator, consistent with RFC 2732, 3986, 4038 and other RFCs. However, this library will parse IPv6 addresses without the brackets. You can use the “expectPort” setting of HostNameParameters to resolve ambiguities when the brackets are absent.

```
    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: ::1 address: ::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 instance of IPAddressConverter for some other suitable conversion (IPv4 translated, 6to4, 6over4, IPv4 compatible, or other).

```
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
```

Should you wish to change the default IPv4/IPv6 conversions from IPv4 mapped to something else, you can override the pair of methods `toIPv4()` and `isIPv4Convertible()` in your own `IPv6Address` subclass and/or the pair of methods `toIPv6()` and `isIPv6Convertible()` in your own `IPv4Address` subclass.

Prefixed Addresses and toHostAddress()

This library will parse CIDR prefix IP addresses such as 10.1.2.3/24. Network addresses like 10.1.2.0/24 and a:b:c:d::/64 are parsed as the prefix block of addresses for the indicated prefix length (alternative options for parsing prefix lengths are described in the section on prefix length handling).

If you have an individual address or a subnet with prefix length and you want to get the address instance representing the entire block for that prefix, you can call `toPrefixBlock()`. In the reverse direction, given an entire block for a prefix, you can get the IPv4 network address or the IPv6 anycast address by calling `getLower()`.

When parsing, you can always obtain the individual address without the prefix length attached. 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 get the prefix length by calling `getNetworkPrefixLength()` on the `IPAddressString` or `IPAddress` instance.

The example code below shows some of these methods and the default prefix length handling behavior.

```
static void printPrefixedAddresses(String addressStr) {
    IPAddressString ipAddressString = new IPAddressString(addressStr);
    IPAddress address = ipAddressString.getAddress();
    System.out.println("count: " + address.getCount());
    IPAddress hostAddress = ipAddressString.getHostAddress();
    IPAddress prefixBlock = address.toPrefixBlock();
        Integer prefixLength = ipAddressString.getNetworkPrefixLength();
        System.out.println(address);
    System.out.println(address.toCanonicalWildcardString());
    System.out.println(hostAddress);
    System.out.println(prefixLength);
    System.out.println(prefixBlock);
        System.out.println();
}

printPrefixedAddresses("10.1.2.3/24");//individual address
printPrefixedAddresses("10.1.2.0/24");//network
```

Output:

```
count: 1
10.1.2.3/24
10.1.2.3
10.1.2.3
24
10.1.2.0/24

count: 256
10.1.2.0/24
10.1.2.*
10.1.2.0
24
10.1.2.0/24
```

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

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

With non-segmented addresses, ambiguity between IPv4 and IPv6 is resolved by the number of digits in the string. 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
    IPAddress address = ipAddressString.getAddress();
    System.out.println(address);
```

```
ipAddressString = new IPAddressString("10800000000000000000080800200c417a");//hex IPv6
address = ipAddressString.getAddress();
System.out.println(address);

ipAddressString = new IPAddressString("0x01020304");//hex IPv4
address = ipAddressString.getAddress();
System.out.println(address);

ipAddressString = new IPAddressString("000100401404");//octal IPv4
address = ipAddressString.getAddress();
        System.out.println(address);
```

Output:

```
1080::8:800:200c:417a
1080::8:800:200c:417a
1.2.3.4
1.2.3.4
```

Parse Special Host Names – Reverse DNS Host Name, IPv6 Literal UNC Host Name

A couple of standardized host formats are recognized, namely the reverse DNS host format, and the UNC IPv6 literal host format.

```
        HostName hostName = new HostName("a.7.1.4.c.0.0.2.0.0.8.0.8.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.0.1.ip6.arpa");
    System.out.println(hostName.asAddress());
    hostName = new HostName("4.3.2.1.in-addr.arpa");
    System.out.println(hostName.asAddress());
        hostName = new HostName("1080-0-0-0-8-800-200c-417a.ipv6-literal.net");
        System.out.println(hostName.asAddress());
```

Output:

```
1080::8:800:200c:417a
1.2.3.4
1080::8:800:200c:417a
```

A couple of methods in HostName are available as well:

```
public boolean isUNCIPv6Literal()
public boolean isReverseDNS()
```

Parse IPv6 Zone or Scope ID

The IPv6 zone or scope ID is recognized, denoted by the '%' character. It can be retrieved by IPv6Address.getZone().

```
        IPAddress addr = new IPAddressString("::%eth0").getAddress();
    if(addr.isIPv6()) {
        System.out.println(addr.toIPv6().getZone());
    }
```

Output:

```
eth0
```

IP Address and Numeric Values

In addition to the range of string formats that can be parsed to produce IPAddress instances, you can also obtain instances of IPAddress from byte arrays, java.net.InetAddress, arrays of address segments, or individual integer segment values.

For IPv4 you have the additional option of constructing an address from a 32-bit integer. For IPv6, you have the additional options of constructing from a java.lang.BigInteger and constructing from MAC addresses.

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

Networks

Each of the different address types (IPv6, IPv4, MAC) has an associated singleton network object. The network objects are used for caching, for configuration, and have methods for obtaining masks and loopbacks. Each of the network objects have singleton creator objects that are used to create addresses, sections, and segments, and those creator objects perform caching of these address components for efficient memory usage and performance. All internally created address components and addresses are created by the creator object, whether by the string parsing engine or by address object manipulation.

The defaultIpv6Network(), defaultIpv4Network(), and defaultMACNetwork() methods in Address provide access to the respective network objects. Each network's associated creator object is accessible from getAddressCreator().

Using your own address classes

Users can create their own subclasses of the address and address component classes (eg your own subclasses of IPv6Address, IPv6AddressSection and/or IPv6AddressSegment).

When doing so, you may wish to use your own network classes as well. To do so, you

- override getNetwork()
- override getIPv6Network(), getIPv4Network(), and getMACNetwork(), if any of those methods exist in the class you are subclassing. Several of those methods may exist in the same class because address components sometimes create components of other address versions or types (eg embedding IPv4 in IPv6, or MAC integration with IPv6).

Once you have done so, your new address classes are using your own network classes as well.

When using your own network classes, you can override the method createAddressCreator() or the method getAddressCreator() in AddressNetwork, and in so doing provide your own creator objects, in which you can override any or all of the creation methods to use your own classes.

You can make the string parsing engine use your own address or network classes. To do this, your code should create your own string validation objects using the builder classes MACAddressStringParameters.Builder, IPAddressStringParameters.Builder and HostNameParameters.Builder. With HostNameParameters.Builder you must get the IPAddressStringParameters.Builder sub-builder by calling getAddressOptionsBuilder(). With IPAddressStringParameters.Builder you must get the IPv4 or IPv6 sub-builder by calling one of getIPv6Parameters() or getIPv4Parameters(). You then call setNetwork() on the builder or sub-builder objects to supply your own network. The initial builder can create your own validation objects with toParams(). Those validation objects can be supplied to the appropriate constructor of IPAddressString or MACAddressString, and the subsequent string parsing with those instances will use the supplied networks.

Prefix Length Handling

Prefix lengths are supported both in parsing, as indicated in the section above on parsing, as well as when directly constructing addresses or sections. Addresses and sections store their prefix lengths and the prefix is incorporated in numerous address operations as well as when producing strings. For instance, an address will provide the network section upon calls to getNetworkSection() in IPAddress, and will supply the prefix length when calling getNetworkPrefixLength().

Given an address with no prefix length, you can convert to an address with prefix length using the methods assignPrefixForSingleBlock() or assignMinPrefixForBlock() in IPAddress, or any of the methods that allow you to set any given prefix length directly such as setPrefixLength(int).

Prefix Length and Equality

In this library, the subnet with prefix length 10.1.2.*/24 is equivalent the non-prefixed address 10.1.2.* as they both contain the same set of addresses. When it comes to equality or comparison, the prefix length has no effect.

Alternative Options for Prefix Length Handling

As indicated in the section on parsing, when constructing an IP address with prefix length, you can end up with an individual address or a subnet if you provide a network address. This is also true for address and address sections constructed from numeric values rather than strings. When you specify the network address of a subnet with its corresponding prefix length, then the resulting object will be the subnet, and not an individual address.

In version 3 and earlier, prefixed addresses and sections were always converted to the subnet block of addresses for that prefix, regardless of whether you specified the network address or any other address.

In the current version, there are three options available, the version 3 legacy behavior, the new default behavior, and a third option:

1. **ALL_PREFIXED_ADDRESSES_ARE_SUBNETS**: An address with a prefix length is always the subnet block for that prefix. This is the legacy behavior, the behaviour for versions 3 and under of this library. An address like 1.2.3.4/16 is equivalent to 1.2.*.*/16, and a:b:c:d::/64 is equivalent to a:b:c:d:*:*:*:*/64

2. **PREFIXED_ZERO_HOSTS_ARE_SUBNETS***:* An address with a prefix length and a zero-valued host is the subnet block for that prefix. An address with a host value of non-zero is a single address corresponding to a single host. This is the new default behavior for this library since it is a convention with common usage.

   For example, the IPv4 address 1.2.3.4/16 is just a single address and not a subnet, while 1.2.0.0/16 is the prefix block, it is the subnet 1.2.*.*/16.

   This behavior is akin to the common convention used by many network administrators, many routers and many applications. With IPv4, an address with a host value of zero is known as the network address. It often represents the subnet of all addresses with that same prefix. It is often used for routing. The all-zero address 0.0.0.0 is conventionally known as INADDR_ANY (any address on the local machine), it is the address that returns true for java.net.Inet4Address.isAnyLocalAddress(), and when paired with prefix zero it is known as the default route (the route for all addresses).

   With IPv6, the convention is similar and this option results in the same behaviour. It is a standard convention with IPv6 to denote a subnet like 1:2:3:4::/64, with a zero-valued (and usually compressed) host, and that subnet represents the prefix block all addresses starting with that prefix 1:2:3:4. An address with a host of zero is known as the anycast address. The all-zero address '::' is the value of IN6ADDR_ANY_INIT, the analog to the IPv4 INADDR_ANY), and the address that returns true for java.net.Inet6Address.isAnyLocalAddress().

3. **EXPLICIT_SUBNETS**: The third option is the one in which a subnet block derived from a prefix length must be explicitly defined, you must list the range of values explicitly. Any subnet must be explicitly defined, like 1.2.*.*/16, while 1.2.0.0/16 is just a single address.

Version 3 and earlier versions of the library provided only the first behavior. All three options are available starting with version 4 of this library, while the default behavior is the second options listed and is most consistent with common conventions used widely for IP addresses.

The options above are generally geared towards IP addresses. With MAC Addresses prefix lengths are implicit for the most part (and cannot be specified as part of an address string explicitly). MAC address prefix lengths are discussed in more detail in the section below "MAC Address Prefix Lengths" With MAC there is no difference between options 2 and 3 since there is no analog to the zero-host/network/anycast address. With MAC, when using an operation that allows you to supply a prefix, the first option above results in addresses that span all values beyond the prefix, while options 2 and 3 do not.

Configuring Alternative Prefix Length Handling

To choose one of the non-default prefix length handling options, such as the legacy option, you make use of the PrefixConfiguration enum type and the network objects.

Each address version/type (IPv4, IPv6, MAC) has an associated network object and each network object has a PrefixConfiguration setting. Prefix length has a similar meaning for MAC but with some key differences outlined below. By default, all networks use the same setting of PREFIXED_ZERO_HOSTS_ARE_SUBNETS stored in a single static field in the AddressNetwork class.

By calling setPrefixConfiguration in AddressNetwork and you can change this default setting for all networks. Or, you can call setPrefixConfiguration in one or more of IPv4AddressNetwork, IPv6AddressNetwork, or MACAddressNetwork, to set each one individually. Typically, you would change the setting when the application starts. Should you choose to do so at a different point in the application lifetime, it should be noted that the behavior is undefined when using pre-existing address objects following such a change. It is recommended that the prefix length configuration remain constant throughout the lifetime of an application, or that existing address objects be discarded when the configuration is changed.

If discarding existing address objects, keep in mind that the library caches address and address component objects. To discard those cached objects you can clear those caches by calling defaultIpv6Network(), defaultIpv4Network(), and defaultMACNetwork() to get the default network objects and then call clearCaches() on each one.

If two applications using the IPAddress library at the same time in the same process require conflicting prefix configurations, then one option to achieve this is to use different class loaders in the separate apps, loading the IPAddress library in separate class loaders. In fact, many popular java applications already use class loaders for code separation, such as the Eclipse framework which uses them to separate plugins and the Apache Tomcat web server which uses them to separate webapps.

If you cannot load the library in separate class loaders, then another option is available to you. This option is to use your own network classes. The procedure for using your own network classes is described above in the section of this document on networks. The first step us to subclass the address and address component classes. The next step is to override the key methods for accessing the networks in those classes to supply your own network subclasses which will have their own non-default configuration. When using your own network classes, you can override the getPrefixConfiguration() method of AddressNetwork to avoid using the default configuration setting and use your own. In your network, you will also need to override either the createAddressCreator() or getAddressCreator() methods to supply your own address creator subclass instance that will create your own address and address component classes.

The following example shows how to create an IPAddressString that will use a customized network class along with customized address component classes:

```
static class MyIPv6Address extends IPv6Address {

    public MyIPv6Address(byte[] bytes, Integer prefixLength) {
        super(bytes, prefixLength);
    }

    public MyIPv6Address(MyIPv6AddressSection section, CharSequence zone) {
        super(section, zone);
    }

    public MyIPv6Address(MyIPv6AddressSection section) {
        super(section);
    }

    @Override
    public IPv6AddressNetwork getNetwork() {
        return myIPv6Network;
    }
}

static class MyIPv6AddressSection extends IPv6AddressSection {

    public MyIPv6AddressSection(byte[] bytes, Integer prefixLength) {
        super(bytes, prefixLength);
    }

    public MyIPv6AddressSection(IPv6AddressSegment[] segments, int startIndex, boolean cloneSegments) {
        super(segments, startIndex, cloneSegments);
```

```
        }

        public MyIPv6AddressSection(IPv6AddressSegment[] segments, Integer prefixLength) {
            super(segments, prefixLength);
        }

        @Override
        public IPv6AddressNetwork getNetwork() {
            return myIPv6Network;
        }
    }

    static class MyIPv6AddressSegment extends IPv6AddressSegment {

        public MyIPv6AddressSegment(int lower, Integer segmentPrefixLength) {
            super(lower, segmentPrefixLength);
        }

        public MyIPv6AddressSegment(int lower, int upper, Integer segmentPrefixLength) {
            super(lower, upper, segmentPrefixLength);
        }

        @Override
        public IPv6AddressNetwork getNetwork() {
            return myIPv6Network;
        }
    }


    static IPv6AddressNetwork myIPv6Network = new IPv6AddressNetwork() {
        @Override
        public PrefixConfiguration getPrefixConfiguration() {
            return PrefixConfiguration.ALL_PREFIXED_ADDRESSES_ARE_SUBNETS;
        }

        @Override
        protected IPv6AddressCreator createAddressCreator() {
            return new IPv6AddressCreator() {
                @Override
                public IPv6AddressSection createSection(byte bytes[], Integer prefix) {
                    return new MyIPv6AddressSection(bytes, prefix);
                }

                @Override
                public IPv6AddressSegment createSegment(int value, Integer segmentPrefixLength) {
                    return new MyIPv6AddressSegment(value, segmentPrefixLength);
                }

                @Override
                public IPv6AddressSegment createSegment(int lower, int upper, Integer segmentPrefixLength) {
                    return new MyIPv6AddressSegment(lower, upper, segmentPrefixLength);
                }

                @Override
                protected IPv6AddressSegment createSegmentInternal(int value, Integer segmentPrefixLength, CharSequence addressStr, int originalVal,
                        boolean isStandardString, int lowerStringStartIndex, int lowerStringEndIndex) {
                    return new MyIPv6AddressSegment(value, segmentPrefixLength);
                }

                @Override
                protected IPv6AddressSection createPrefixedSectionInternal(IPv6AddressSegment segments[], Integer prefix) {
                    return new MyIPv6AddressSection(segments, prefix);
                }

                @Override
                protected IPv6AddressSection createSectionInternal(IPv6AddressSegment segments[]) {
                    return new MyIPv6AddressSection(segments, 0, false);
                }

                @Override
                protected IPv6Address createAddressInternal(IPv6AddressSection section, HostIdentifierString from) {
                    return new MyIPv6Address((MyIPv6AddressSection) section);
                }

                @Override
                protected IPv6Address createAddressInternal(IPv6AddressSection section, CharSequence zone, HostIdentifierString from) {
                    return new MyIPv6Address((MyIPv6AddressSection) section, zone);
                }
            };
        }
    };


    IPAddressStringParameters params = new IPAddressStringParameters.Builder().
            getIPv6AddressParametersBuilder().setNetwork(myIPv6Network).getParentBuilder().toParams();
```

```
IPv6Address myAddr = new IPAddressString("1::1/64", params).getAddress().toIPv6();
```

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 masking, obtaining subnets, and so on.

Summary of IP Address Operations

Here is a summary, a non-exhaustive list, in no specific order, of operations for transforming addresses and subnets:

- **toPrefixBlock**: Provides the subnet for the entire prefix block using the existing or a supplied prefix length (toPrefixBlock is redundant when using prefix configuration ALL_PREFIXED_ADDRESSES_ARE_SUBNETS because prefixed addresses are always a prefix block with that configuration). You can use isPrefixBlock to determine if the operation would have no effect on the address because it is already a prefix block.
- **assignPrefixForSingleBlock** (formerly toPrefixedEquivalent): Converts a subnet address with no prefix length to an equivalent address with a prefix length. The resulting address will span the same range of values as the original. You can use getPrefixLengthForSingleBlock (formerly getEquivalentPrefix) to get the resultant prefix prior to the operation, or isSinglePrefixBlock (formerly isRangeEquivalentToPrefix) to determine if the operation would have no effect on the address because it is already a single prefix block.
- **assignMinPrefixForBlock** (formerly 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. You can use getMinPrefixLengthForBlock (formerly getMinPrefix) to get the resultant prefix prior to the operation.
- **mergeToPrefixBlocks, mergeToSequentialBlocks**: Given a list of addresses or subnets, merges them into the minimal list of prefix block subnets or sequential block subnets
- **join**: Given a list of address ranges, merges them into the minimal list of address ranges
- **spanWithPrefixBlocks, spanWithSequentialBLocks**: Given a pair of addresses or subnets, finds the minimal list of prefix block subnets or sequential block subnets that span all addresses between the pair
- **append**, **prepend**, **replace**: Add or replace segments in sections and addresses. On addresses you must always maintain the correct number of segments, so only the replace operation is provided. You can obtain a section by either constructing one directly or by calling a variant of getSection on an address.
- **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 potential values.
- **withoutPrefixLength** Convert to the same address but with no prefix length
- **removePrefixLength, adjustPrefixLength, applyPrefixLength, setPrefixLength**: Add, remove or adjust prefix lengths by the indicated values. There are variants with a boolean argument to control whether extended prefix lengths have zeros for the added bits, or whether the bits retain their values from when they were on the other side of the prefix boundary. 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. removePrefixLength is like withoutPrefixLength but will convert the former host bits to zero

- **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.
- **intersect**: Computes the subnet conjunction, the set of addresses in both the receiver subnet and the argument subnet.
- **increment**: Given a positive or negative integer increment value, if the address is a subnet, provides the address that is the given increment into the list of subnet addresses. Any increment exceeding the count of addresses in the subnet is simply added to the final address in the subnet. If the address is an individual address, adds the given increment to the address value to produce a new address.
- **toIPv4, toIPv6, toEUI**: Use either standard or customized address conversions to go from one address version/type to another
- **iterator**: iterates through the individual addresses of a subnet
- **prefixBlockIterator**: iterates through the individual prefix block subnets of a larger subnet

Queries for Prefix Lengths and Prefix Blocks

In some cases you may need to know if you have a prefix block subnet, such as the /64 block a:b:c:d::/64 or the /16 block 1.2.0.0/16, in which the subnet contains all the addresses with the same prefix of a given length. The following methods allow you to query for prefix blocks and lengths of blocks.

- **containsPrefixBlock**: Answers whether the subnet contains the prefix block of addresses for a given prefix length (regardless of whether the address is assigned a different prefix length or has none assigned).
- **containsSinglePrefixBlock**: Like containsPrefixBlock, but queries if the subnet contains *only* the prefix block of addresses for a given prefix length. In other words, it answers whether the subnet contains the full block of address for just a single prefix
- **isPrefixBlock**: Like containsPrefixBlock but using the prefix length assigned to the address.
- **isSinglePrefixBlock**: Like containsSinglePrefixBlock but using the prefix length assigned to the address
- **getBlockMaskPrefixLength**: (formerly getMaskPrefixLength): Will return the prefix length for a network or host mask, if it is one
- **getPrefixLengthForSingleBlock**: Returns a prefix length for which the range of this segment grouping matches exactly the block of addresses for that prefix, if such a prefix length exists, and if it does, it will match the result of getMinPrefixLengthForBlock
- **getMinPrefixLengthForBlock**: Returns the smallest prefix length such that this address division series includes the block of addresses for that prefix, which is the bit count for individual addresses.

Iterators

Various iterators are available for iterating through subnet in different ways.

- **iterator**: Iterates through all addresses. Use getCount to get the iterated count.
- **nonZeroHostIterator**: Like iterator, but if the subnet has a prefix length, then this skips all zero-valued hosts. This can be useful because the zero-valued host in a subnet typically represents the subnet as a whole and is typically not used as an individual address. Use getNonZeroHostCount to get the iterated count.
- **prefixBlockIterator**: If the subnet has a prefix length, then this iterates through the prefixes, with each iterated item being the prefix block for each prefix. If no prefix length, same as iterator. Use getPrefixCount to get the iterated count.
- **prefixIterator**: If the subnet has a prefix length, then this iterates through the prefixes, with each iterated item including all the addresses in the original iterated subnet which have the same prefix. For all iterated items except the lowest and highest this is the prefix block, while the lowest and highest prefix might have fewer addresses than the prefix block. If no prefix length, same as iterator. Use getPrefixCount to get the iterated count.
- **blockIterator**: Iterates through the segment values of the higher segments up until a given segment index. If the segment index is 0, produces just a single value, the original address. If the segment index is the segment count, same as iterator. Like prefixIterator, for each segment prefix, includes all addresses in the original iterated subnet which have that prefix. Use getBlockCount to get the iterated count.
- **sequentialBlockIterator**: Iterates through the segment values of the higher segments up until a specific segment index. The segment index is chosen to be the largest segment index which allows for all iterated address blocks to be sequential. Use getSequentialBlockCount to get the iterated count.

Mask and Prefix Length Operations

The methods mask, maskNetwork, bitwiseOr, bitwiseOrNetwork, removePrefixLength, adjustPrefixBySegment, adjustPrefixLength, applyPrefixLength, and setPrefixLength allow you to apply and adjust prefix lengths, and to apply 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 be something representable with sequential segment ranges, or you will get IncompatibleAddressException (formerly 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);

        IPAddress maskedAddress = address.mask(mask);
    System.out.println("address " + address + " masked " + maskedAddress);

        //create the subnet
        IPAddress subnet = address.applyPrefixLength(prefixLength).toPrefixBlock();

        //mask
    IPAddress maskedSubnet = subnet.mask(mask);
    System.out.println("subnet " + subnet + " masked " + maskedSubnet);
    System.out.println(maskedAddress.equals(maskedSubnet));

    //getLower
        IPAddress lowestAddressInSubnet = subnet.getLower();
    System.out.println("lowest in subnet " + lowestAddressInSubnet);
        System.out.println("lowest in subnet no prefix " + lowestAddressInSubnet.removePrefixLength(false));
        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
true
lowest in subnet 1.2.0.0/16
lowest in subnet no prefix 1.2.0.0
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::
true
lowest in subnet a::/16
lowest in subnet no prefix a::
true
prefix removed a::
```

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(2, replacementSection));
        System.out.println(subnet);
```

Output:
207.0.68.0/22

Alternatively, you can use the prefixBlockIterator() method to get a list of subnets when adjusting the prefix:

```
        IPAddress subnet = new IPAddressString("192.168.0.0/28").getAddress();
    IPAddress newSubnets = subnet.setPrefixLength(subnet.getPrefixLength() + 2, false);
    System.out.println(newSubnets);
    TreeSet<IPAddress> subnetSet = new TreeSet<IPAddress>();
    Iterator<? extends IPAddress> iterator = newSubnets.prefixBlockIterator();
    while (iterator.hasNext()) {
      subnetSet.add(iterator.next());
    }
    System.out.println(subnetSet);
```

Output:
192.168.0.0-12/30
[192.168.0.0/30, 192.168.0.4/30, 192.168.0.8/30, 192.168.0.12/30]

IP Address Ranges

An IPAddress or IPAddressString instance can represent any individual address or any range of addresses in which each segment specifies its own range of sequential values. For IPAddress, the ranges are specified with the canonical number of segments for each address version or type, which is 4 segments for IPv4, 8 for IPv6, and either 6 or 8 for MAC.

Any prefix block of addresses can be specified this way (a block of addresses that includes all the hosts for a given prefix).

An IPAddressString can represent any such address string, even those that do not use the canonical number of segments. However, for ranges involving a non-canonical number of segments, converting to an IPAddress instance is not always possible (for example the IPv4 subnet 1.2-3.0-1000 cannot be expressed with 4 segments).

Not all IPAddress subnets are sequential. For instance, 1-2.3.4-5.6 is not sequential, since the address 1.3.4.6 is followed in the block by 1.3.5.6 while the next sequential address 1.3.4.7 is not part of the block. A sequential block is one in which the range is sequential. For a block to be sequential, the first segment with a range of values must be followed only by segments that cover all values. For instance, 1.2.3-4.* is a sequential block, as well as 1:a-f:*:*:*:*:*:*, also writeable as 1:a-f:*. Any prefix block is sequential and can also be expressed without the prefix length. For instance, the prefix block 1:2:3:4::/64 can be written without the prefix length as 1:2:3:4:*.

You can convert a non-sequential block to a collection of sequential blocks using the sequentialBlockIterator method of IPAddress. If you wish to get the count of sequential blocks, use the methods getPrefixCount or getSequentialBlockCount.

```
convertNonSequentialBlock("a:b:c:d:1:2-4:3-5:4-6");

static void convertNonSequentialBlock(String string) {
    IPAddressString addrString = new IPAddressString(string);
    IPAddress addr = addrString.getAddress();
    System.out.println("Initial range block is " + addr);
    BigInteger sequentialCount = addr.getSequentialBlockCount();
    System.out.println("Sequential range block count is " +
               sequentialCount);
    Iterator<? extends IPAddress> iterator =
               addr.sequentialBlockIterator();
    while(iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

Output:

```
Initial range block is a:b:c:d:1:2-4:3-5:4-6
Sequential range block count is 9
a:b:c:d:1:2:3:4-6
a:b:c:d:1:2:4:4-6
a:b:c:d:1:2:5:4-6
a:b:c:d:1:3:3:4-6
a:b:c:d:1:3:4:4-6
a:b:c:d:1:3:5:4-6
a:b:c:d:1:4:3:4-6
a:b:c:d:1:4:4:4-6
a:b:c:d:1:4:5:4-6
```

Not all sequential address ranges can be described by an instance of IPAddress or IPAddressString. One such example is the range of two IPv4 addresses from 1.2.3.255 to 1.2.4.0. One option is to represent the address range with just a single large segment covering the section of the address that has a range of values, such as in an instance of IPAddressLargeDivisionGrouping.

The better option is to use an IPAddressSeqRange instance, which provides a more general representation of address ranges and their associated operations. You can represent any sequential range of addresses with an IPAddressSeqRange instance.

Both IPAddress and IPAddressSeqRange implement the IPAddressRange interface. IPAddressSeqRange instances cover all ranges of addresses, while IPAddress cover all ranges of segments.

An IPAddressSeqRange instance can be converted to the minimal list of sequential blocks or prefix blocks that cover the same range of addresses, so that is a couple of ways to go from IPAddressSeqRange to IPAddress instances. To go in the reverse direction from IPAddress to IPAddressSeqRange, you can convert any IPAddress sequential block or prefix block to an IPAddressRange with the method toSequentialRange, and then you can join any collection of IPAddressRange instances with the join method in IPAddressSeqRange.

Here we show a code example of creating a sequential range, breaking it up into either prefix blocks or sequential blocks, and then merging those blocks to get the original sequential range.

```
testSpanAndMerge("2:3:ffff:5::", "2:4:1:5::");

static void testSpanAndMerge(String address1, String address2) {
    IPAddressString string1 = new IPAddressString(address1);
    IPAddressString string2 = new IPAddressString(address2);
    IPAddress addr1 = string1.getAddress();
    IPAddress addr2 = string2.getAddress();
    IPAddressSeqRange range = addr1.toSequentialRange(addr2);
    System.out.println("Original range of size " + range.getCount() + ": " + range);

    IPAddress result[] = range.spanWithPrefixBlocks();
    System.out.println("Prefix blocks: " + Arrays.asList(result));

    IPAddress result2[] = range.spanWithSequentialBlocks();
    System.out.println("Sequential blocks: " + Arrays.asList(result2));

    List<IPAddressSeqRange> rangeList = new ArrayList<>();
    for(IPAddress a : result) {
        IPAddressSeqRange r = a.toSequentialRange();
        rangeList.add(r);
    }
    IPAddressSeqRange joined[] = IPAddressSeqRange.join(
                rangeList.toArray(new IPAddressSeqRange[rangeList.size()]));
    System.out.println("Merged prefix blocks back again: " + Arrays.asList(joined));

    rangeList.clear();
    for(IPAddress a : result2) {
        IPAddressSeqRange r = a.toSequentialRange();
        rangeList.add(r);
    }
    joined = IPAddressSeqRange.join(rangeList.toArray(
                    new IPAddressSeqRange[rangeList.size()]));
    System.out.println("Merged sequential blocks back again: " + Arrays.asList(joined));
}
```

Output:

```
Original range of size 2417851639229258349412353: 2:3:ffff:5:: -> 2:4:1:5::
Prefix blocks: [2:3:ffff:5::/64, 2:3:ffff:6::/63, 2:3:ffff:8::/61, 2:3:ffff:10::/60, 2:3:ffff:20::/59, 2:3:ffff:40::/58, 2:3:ffff:80::/57, 2:3:ffff:100::/56, 2:3:ffff:200::/55, 2:3:ffff:400::/54, 2:3:ffff:800::/53, 2:3:ffff:1000::/52, 2:3:ffff:2000::/51, 2:3:ffff:4000::/50, 2:3:ffff:8000::/49, 2:4::/48, 2:4:1::/62, 2:4:1:4::/64, 2:4:1:5::]
Sequential blocks: [2:3:ffff:5-ffff:*:*:*:*, 2:3-4:*:*:*:*:*:*, 2:4:0:*:*:*:*:*, 2:4:1:0-4:*:*:*:*, 2:4:1:5::]
Merged prefix blocks back again: [2:3:ffff:5:: -> 2:4:1:5::]
Merged sequential blocks back again: [2:3:ffff:5:: -> 2:4:1:5::]
```

As you can see in the example above, you can generally describe a range with fewer sequential blocks than prefix blocks.

Parse String Representations 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-ff0fffffffff"
          };
    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.

Once a MAC address as an associated prefix length, that prefix length remains the same when any operations are applied to the address.

In summary, 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 0xffe. 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:

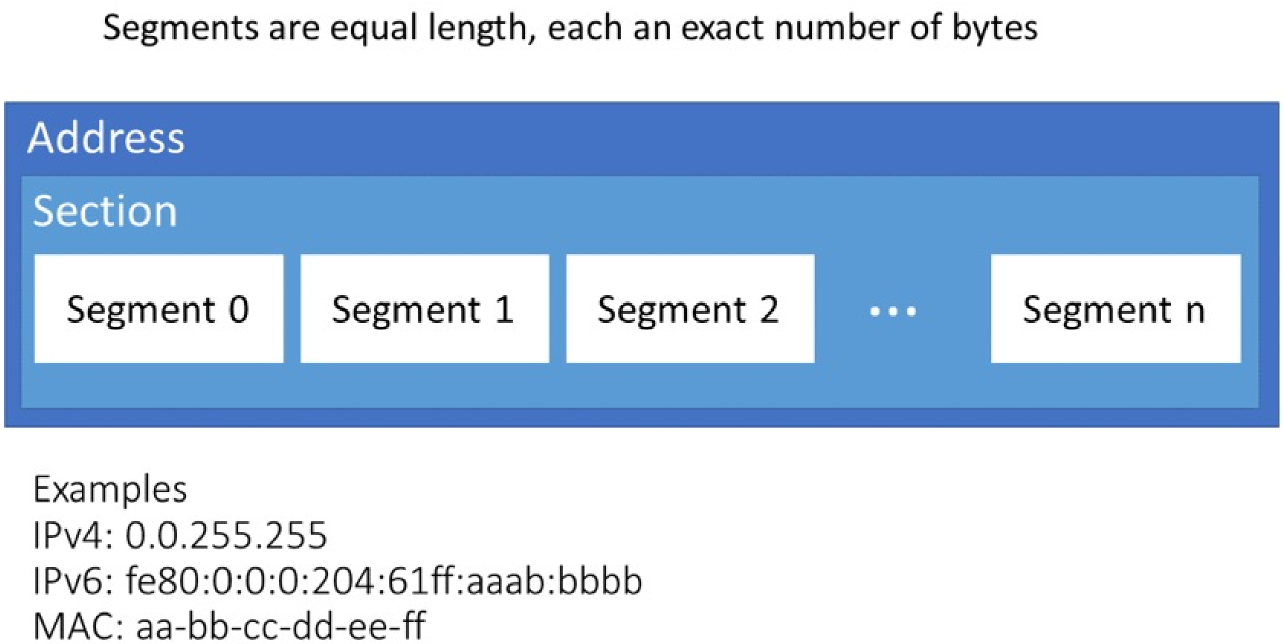
```
fe80::a8bb:ccff:fedd:eeff
1111:2222:3333:4444:a8bb:ccff:fedd:eeff/64
```

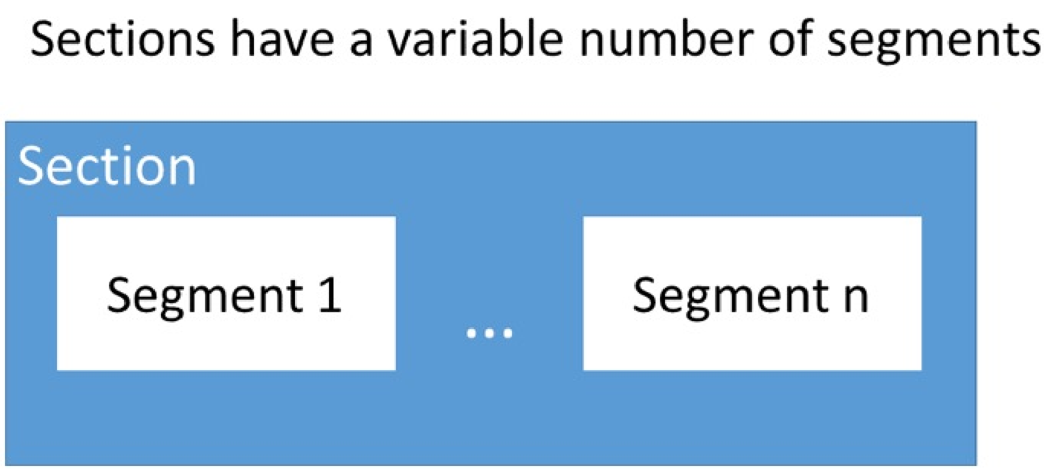
Address Framework

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.

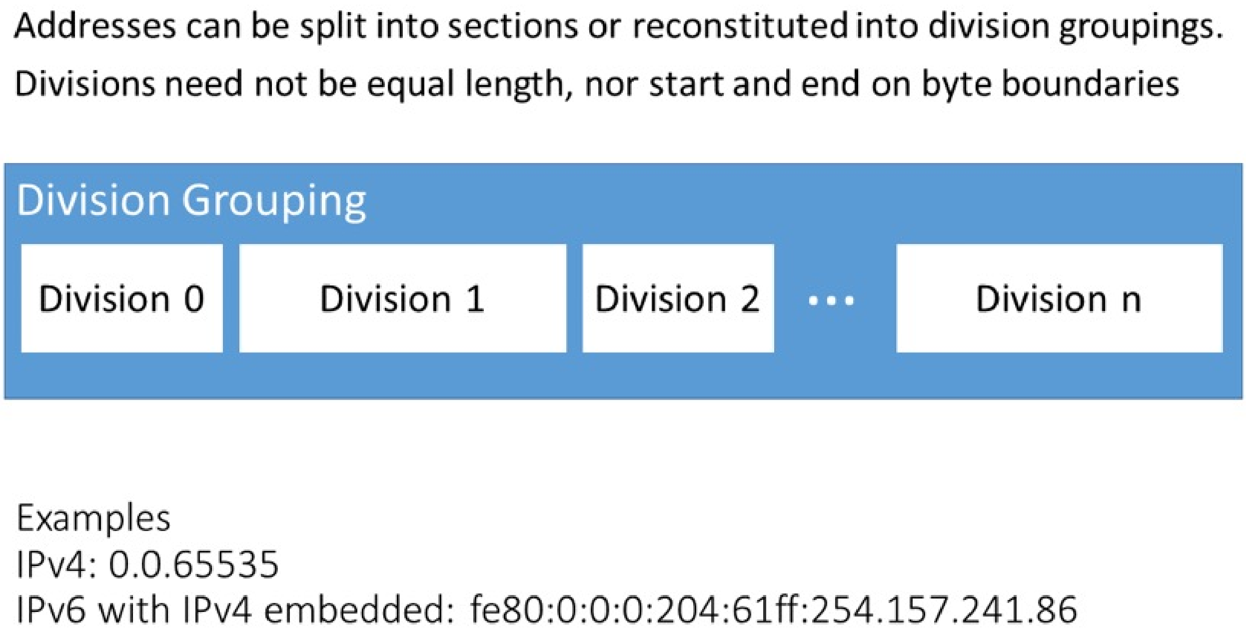
You might wish to manipulate address components of different shapes and sizes transparently, or you may wish to manipulate different types of addresses or different address versions transparently. One element of the interface is the ability to convert any address component to bytes, whether division,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

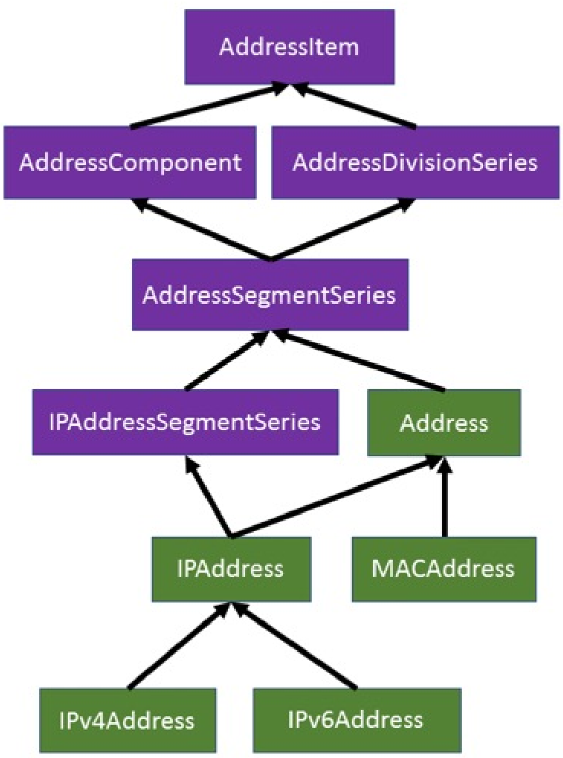




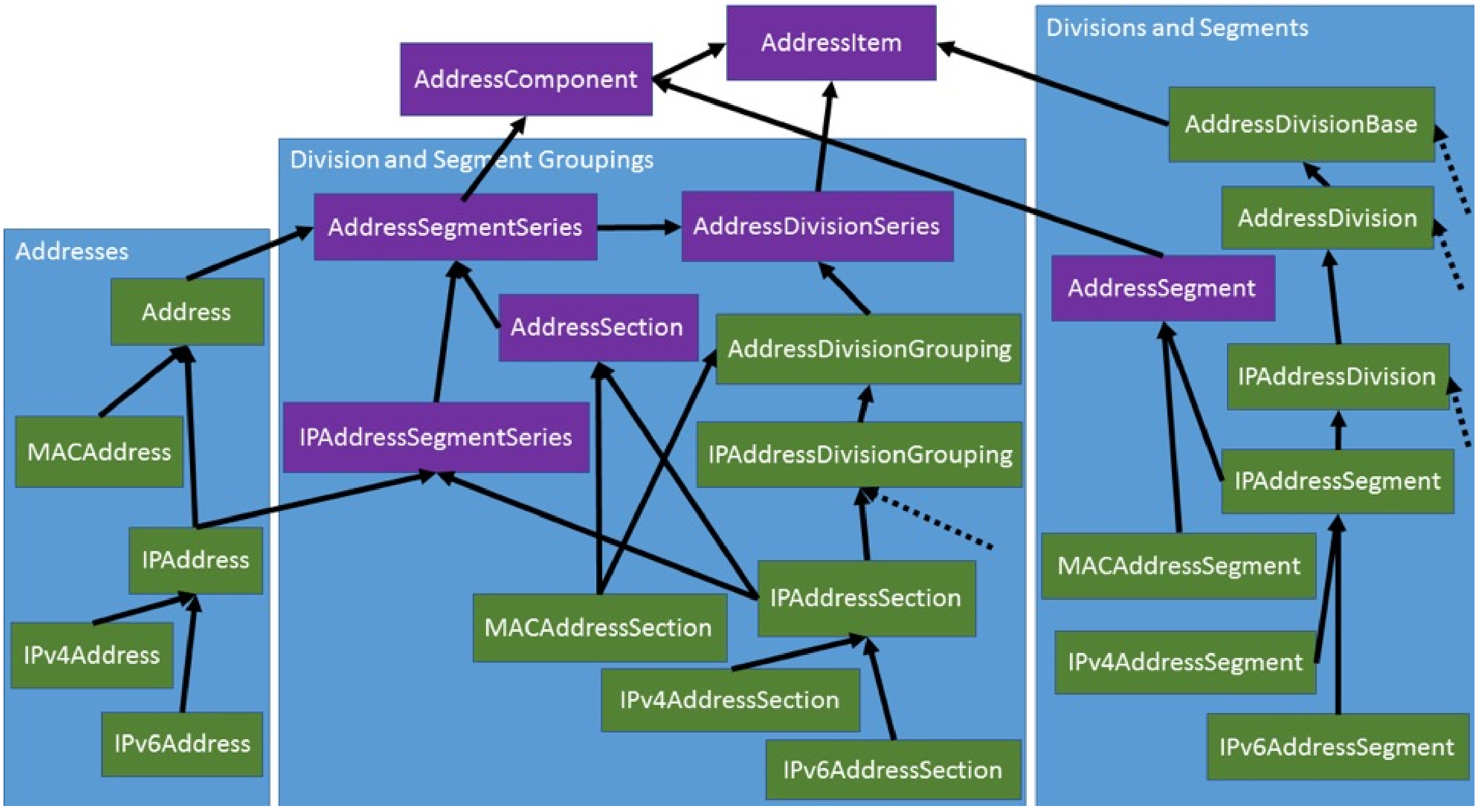
There is a more diversified hierarchy for non-standard address structures, in which addresses or address sections might be divided into divisions of unequal length, or of non-integral bytesize.



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

Here is a list of string methods, in no specific order.

Note that, for a given address, the string produced by one of these methods might match those of other methods. They are not necessarily distinct from each other for all addresses.

All address types, HostName and IPAddressString:

- **toNormalizedString**: Produces a consistent string. For addresses, a string that is somewhat similar and consistent for all address components of the same type. Uses no segment compression in IPv6. Uses xx:xx:xx:xx:xx:xx for MAC address. Prints HostName instances in standard formats as expected by URLs and as dictated by RFCs, including port or service name and using square brackets for IPV6 addresses. Prints a standardized format for IPAddressString instances that cannot be converted to addresses (such as '*'). For HostName and IPAddressString instances constructed from invalid strings, prints the original string used to construct.
- **toString**: same as **toCanonicalString** for addresses. For HostName and IPAddressString instances, prints the original string used to construct.

All address types:

- **toCanonicalString**: the string recommended by one or more RFCs or other common standard.
- **toCompressedString**: a short representation of the address while remaining within the confines of standard representation(s) of the address. For IPv6 compresses all compressible segments.
- **toHexString**: base 64

IP addresses and HostName:

- **toNormalizedWildcardString**: similar to **toNormalizedString**, but uses wildcards and does not print prefix length for addresses that have prefix lengths

IP addresses only:

- **toFullString**: a string which maintains a common length, a string with no compressed segments and all segments of full length, which is 4 characters for IPv6 segments and 3 characters for IPv4 segments
- **toCanonicalWildcardString**: similar to **toCanonicalString**, but uses wildcards and does not print prefix length for addresses that have prefix lengths
- **toCompressedWildcardString**: similar to **toCompressedString**, but uses wildcards and does not print prefix length for addresses that have prefix lengths
- **toSQLWildcardString**: similar to **toNormalizedWildcardString**, but uses SQL wildcards

- **toPrefixLengthString**: a string with CIDR prefix length if it has one, and compresses the host for IPv6. For IPv4 it is the same as **toCanonicalString**.
- **toSubnetString**: uses **toNormalizedWildcardString** for IPv4 and **toPrefixLengthString** for IPv6
- **toReverseDNSLookupString**: a string for reverse DNS lookup
- **toOctalString**: base 8, optionally with a '0' prefix to indicate octal
- **toBinaryString**: all ones and zeros
- **toConvertedString**: if an IPv6 address can be converted to IPv4 as determined by isIPv4Convertible(), produces the result of **toMixedString**. For IPv4 same as **toCanonicalString**.
- **toUNCHostName**: Microsoft UNC path component
- **toNormalizedString(IPStringOptions)** : your own customized string

IPv6 only:

- **toMixedString**: mixed IPv6/IPv4 string format like a:b:c:d:e:f:255.0.255.255
- **toBase85String**: RFC 1924

MAC address only:

- **toColonDelimitedString**: same as **toNormalizedString**, xx:xx:xx:xx:xx:xx
- **toDashedString**: same as **toCanonicalString**, xx-xx-xx-xx-xx-xx
- **toDottedString**: xxxx.xxxx.xxxx
- **toSpaceDelimitedString**: xx xx xx xx xx xx

More Details and Examples

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("1.2.0.0/16").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:0:0:0:0:0/64
a:bb:c::/64
0x000a00bb000c0000000000000000000-0x000a00bb000c0000ffffffffffffffff
lower: a:bb:c:0:0:0:0:0 bytes:[0, 10, 0, -69, 0, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
upper: a:bb:c:0:ffff:ffff:ffff:ffff bytes:[0, 10, 0, -69, 0, 12, 0, 0, -1, -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);

        /a:b:c:*:: cannot be represented as a range of two single values
    //so this throws exception in toBase85String(), toBinaryString(), and toHexString()
    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());
        }
        System.out.println(address.toBinaryString());
        System.out.println(address.toHexString(true));
    } catch(IncompatibleAddressException e) {}
    System.out.println();
}
```

Output:

```
a:b:c::e:f
000a:000b:000c:0000:0000:0000:000e:000f
a:b:c:0:0:0:e:f
a:b:c:0:0:0:e:f
a:b:c::e:f
a:b:c::0.14.0.15
00|N0s0$N0-%*(tF74+!
00000000000010100000000000001011000000000000110000000000000000000000000000000000000000000000000000000000000011100000000000001111
0x000a000b000c000000000000000e000f

a:b:c::
000a:000b:000c:0000:0000:0000:0000:0000
a:b:c:0:0:0:0:0
a:b:c:0:0:0:0:0
a:b:c::
a:b:c::
00|N0s0$N0-%*(tF5|-X
00000000000010100000000000001011000000000000110000000000000000000000000000000000000000000000000000000000000000000000000000000000
0x000a000b000c00000000000000000000

a:b:c:*::
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:*::
```

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-db8-0-0-0-0-0-1.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());

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

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

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

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

Output:

```
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa
0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa
1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa
0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa
*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa
0.0.0.0.0.0.0.0.8.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, including segments and sections.

Prefix Length Indicator in Strings

Typically prefix lengths will be added to IP 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:*:*:*:*
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:*:*:*:*:*
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.

Cache Classes

The IPAddressNetwork class defines IPAddressStringCache and HostNameCache. These cache classes allow you to cache identifier strings (hosts and/or addresses). Note that the identifier strings themselves will also 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 frequently. These caches do quick lookups using either bytes or strings, which can be ideal for some applications that handle many addresses or host names.